2013 IEEE Sixth International Conference on Software Testing, Verification and Validation

Automated Detection of Test Fixture Strategies and Smells

Michaela Greiler, Arie van Deursen
Delft University of Technology
{m.s.greiler||arie.vanDeursen}@tudelft.nl

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

***Abstract*—Designing automated tests is a challenging task. One important concern is how to design test fixtures, i.e. code that initializes and configures the system under test so that it is in an appropriate state for running particular automated tests. Test designers may have to choose between writing in-line fixture code for each test or refactor fixture code so that it can be reused for other tests. Deciding on which approach to use is a balancing act, often trading off maintenance overhead with slow test execution. Additionally, 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 show that test smells related to fixture set-up occur in industrial projects. We present a static analysis technique to identify fixture related test smells. We implemented this test analysis technique in a tool, called TestHound, which provides reports on test smells and recommendations for refactoring the smelly test code. We evaluate the tool through three industrial case studies and show that developers find that the tool helps them to understand, reflect on and adjust test code.**

***Keywords*-test code comprehension; maintainability; test fixture; test smells; software testing; test code refactoring;**

I. Introduction

Modern software development practice dictates early and frequent (automated) testing. While automated test suites written by developers are helpful from a (continuous) integration and regression testing perspective, they lead to a substantial amount of test code. Like production code, test code needs to be maintained, understood, and adjusted, which can become very costly. The long term success of automated testing is highly influenced by the maintainability of the test code [14]. To support easier maintainability of a system, test methods should be clearly structured, well named and small in size [7]. The duplication of code across test methods should be avoided.

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* [14]. Developers can adopt several strategies for structuring their 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*. A positive aspect of an *in-line* setup is the proximity of the setup code to the test itself. However, when several test methods require the same fixture, an *in-line* setup can lead to code duplication and high maintenance costs [5]. Also, configuring the SUT within the test method might hide the main purpose of the test and result in an *obscure test* [14].

An alternative approach is to place the setup code in helper methods that can be called by several test methods, which we refer to as a *delegate setup* [14]. With a delegate setup, the developer has to make sure the right methods are invoked at the right time (e.g. as a first statement in a test method).

In today's testing frameworks, such as the widely used xUnit family, there is a dedicated mechanism to manage setup code invocations [1], [8]. Therefore, helper-methods containing the setup code can be marked (e.g. using annotations or naming conventions) as specific setup methods, which we refer to as an *implicit setup*.[1] The advantage of an *implicit setup* is that the framework takes care of invoking the setup code at a certain point in time and for a specific group of tests, but also that the methods are explicitly marked as setup which helps with code comprehension. Often, *implicit* setups are invoked either before each test within a class, or once before all the tests within a class. One main drawback of this approach is that the tests grouped together (i.e. within one class) should have similar needs in the test fixture. Otherwise, tests might only access (small) portions of a broader fixture, which can lead to slow tests and maintenance overhead.

During the evolution of test code, developers have to make conscious decisions about how to set up the test fixture and adjust their fixture strategies, otherwise they end up with poor solutions to recurring implementation and design problems in their test code, so-called *test smells* [5]. Unfortunately, until now, no support has been made available to developers during the analysis and adjustment of test fixtures.

To address this shortcoming, we developed a technique that automatically analyzes test fixtures to detect fixture-related smells and guides improvement activities. We implemented this technique in *TestHound*, a tool for static fixture analysis. We evaluate our technique in a mixed methods research approach. First, we analyzed the test fixtures of three industry-strength software systems. Second, we eval-

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

978-0-7695-4968-2/13 $26.00 © 2013 IEEE
DOI 10.1109/ICST.2013.45

uated the usefulness of the technique with 13 developers. In the paper, we show that fixture-related smells exist in practice, and that developers find TestHound helpful during fixture management.

In Section II, we briefly summarize different test smells related to test fixtures. In Section III, our fixture analysis technique is presented, followed by implementation details in Section IV. Section V details our experimental design. In Section VI, the evaluation of our technique is presented, followed by a discussion in Section VII. In Section VIII, we present related work, and conclude in Section IX.

II. Test Smells

In earlier research [9], we interviewed 25 Java developers on information needs for test code understanding. We observed that the test structure is important for developers to navigate and retrieve tests within a code base. For example, to support easier retrieval of test code, it is a common practice in Java-based systems to organize tests similar to production code (i.e. class to test class, package to test package). Although this practice is chosen to facilitate maintenance, it might lead to groups of tests within one test class that have very different requirements on the system under test. This means that each test might need a different *test fixture* that initializes and configures the system under test and all its dependencies (to fulfill all preconditions of a test). As test code grows and evolves, this strategy can lead to *test smells* with respect to the test fixture.

The code smell metaphor has been introduced by Fowler [6] who describe a smell as a poor solution to a recurring implementation and design problem. 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 concepts of smells to test code [5]. The initial set of test smells has been extended by several researchers [14], [19], [17]. We further extend this set, in particular, with test smells related to test fixtures. Apart from the General Fixture Smell (introduced by van Deursen et al. [5]), we present five new test smells as well as possible refactorings to address these issues:

General Fixture Smell. The general fixture smell occurs if test classes contain broad functionality in the *implicit* setup, and different 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 outcome is less visible, and tests are harder to read and understand. This can cause tests to be fragile: a change that should be unrelated affects tests because too much functionality is covered in the fixture. Secondly, the test execution performance can deteriorate, and test execution times may eventually lead to developers avoiding to execute tests. *Refactoring.* A general fixture can be refactored by creating a minimal fixture, which covers only the setup code common for all test methods. Individual setups can be placed in delegate setups by applying an extract method refactoring. In the case where the test methods do not share too much setup code, an extract class refactoring can be applied.

Test Maverick. Based on the general fixture smell, we derived a related smell: the test maverick smell. A test method is a maverick when the class comprising the test method contains an *implicit* setup, but the test method is completely independent from the implicit setup procedure. The setup procedure will be executed before the test method is executed, but it is not needed. Also, understanding the effect-cause relationship between setup and test method can be hampered. Discovering that test methods are unrelated from the implicit setup can be time consuming. *Refactoring.* Test mavericks can be eliminated by the *extract class refactoring*, placing them in their own class.

Dead Fields. The dead field smell occurs when a class or its super classes have fields that are never used by any test method. Often dead fields are inherited. This can indicate a non-optimal inheritance structure, or that the super class conflicts with the single responsibility principle. Also, dead fields within the test class itself can indicate incomplete or deprecated development activities. *Refactoring.* Dead fields associated with the class should be removed. A possible refactoring for dead fields of the super class are splitting the super class into several classes.

Lack of Cohesion of Test Methods. Cohesion of a class indicates how strongly related and focused the various responsibilities of a class are [4]. Classes with high cohesion facilitate code comprehension and maintenance. Low cohesive methods are smelly because they aggravate reuse, maintainability and comprehension [6], [12]. The smell Lack of Cohesion of test methods (LCOTM) occurs if test methods are grouped together in one test class, but they are not cohesive. *Refactoring.* To reduce LCOTM, the extract class refactoring can be applied to split a test class with too many test responsibilities into different classes.

Obscure In-Line Setup. Meszaros introduced the smell obscure test to refer to a test that is difficult to understand [14] and thus is not suitable for documentation purposes. Based on this smell, we created the obscure in-line setup. An *in-line* setup should consist of only the steps and values essential to understanding the test. Essential but irrelevant steps should be encapsulated into helper methods. An obscure in-line setup covers too much setup functionality within the test method. This can hinder developers in seeing the relevant verification steps of the test. *Refactoring.* To conquer obscure in-line setups, the setup code can be moved into delegate setup methods, or if the in-line setup is common to all tests, one can use an *implicit setup*.

Vague Header Setup. A vague header setup smell occurs when fields are initialized in the header of a class, but not in *implicit* setup. We consider this a smell as the behavior of the code is not explicitly defined, and depends on the

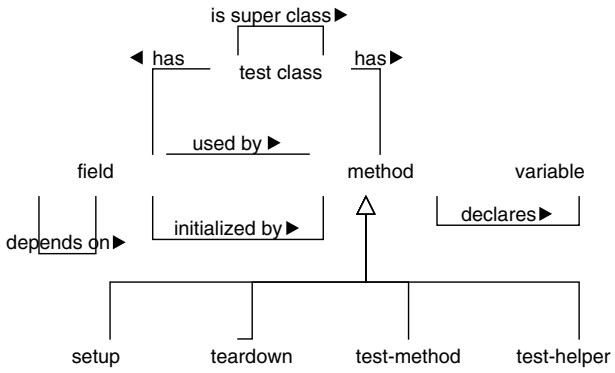

Figure 1. Meta-model Test Fixture Analysis

field modifier (static or member), as well as on the implementation of the test framework. Further, field declarations are not restricted to the header of a class, but can occur anywhere within the class. Vague header setups hamper code comprehension and maintainability. *Refactoring.* Field initializations should be placed in an *implicit* setup to specify the behavior and the places to inspect within a class.

III. Analysis of Fixture Usage

This section describes the technique we developed to analyze the test fixture organization, fixture usage and fixture smells, and suggest refactorings for the test code. Our reverse engineering technique follows the well-known reconstruction approach: fact extraction, abstraction, and presentation [18].

A. Fact Extraction

To determine fixture strategies and fixture-related smells we extract several facts for each test class. All relevant entities for our analysis are illustrated in the meta model in Figure 1. Firstly, we identify all methods in a class. We differentiate between test methods, setup methods, tear-down methods and test helper methods based on the method's annotation or naming conventions.[2] Further, we extract all global fields of the class, and all local variables for each of the test methods.

B. Analysis

The analysis consists of two steps. First, we derive indicators for smells based on the extracted facts as summarized in Table I. Second, we use those indicators to measure the existence of test smells based on our metrics (see Table II).

Implicit Fixture Usage Indicators. To determine how much a test class and its test methods use the *implicit* setup, we derive smell indicators `setupFlds`, `usedSetupFlds` and `deadFlds`. `setupFlds` are fields that are initialized in the *implicit* setup procedures or the class header. For example, in Listing 1, the fields $repository$, $repository2$, $gitDir$ and $store$ are seen as `setupFlds`. `usedSetupFlds` represents the number of `setupFlds` of the class that have also been accessed (i.e. read or write) by a test method. This access can happen directly in the test method or via (a chain of) helper methods invoked by the test method. A field only accessed in the setup, but by no test method is not seen as used. In Listing 1, test method $testRepository()$ uses fields $repository$ and dir, whereas the fields $repository2$ and $store$ are not used. Further, we *establish field dependency relationships* to determine whether a setup field is used by a test method. Setup fields depend on each other if one field f_a is used to set another field f_b (e.g. $f_b = f_a$). To extract these relations, we have to analyze the data flow. We detect direct assignments, but also whether a field f_a depends on another field f_b based on method calls (e.g., $f_b.set(f_a)$). Hereby, the field used for the method call (f_b) is seen as dependent on the fields (f_a) used as parameter. This means that in Listing 1, test method $testRepository()$ also uses field $gitDir$, as $repository$ depends on $gitDir$. Finally, `deadFields` are fields that are initialized in the *implicit* setup, but are never used by a test method. In our example, field $repository2$ is never used by any test method.

[2]This depends on the particular test framework.

Table I
Smell Indicators

Indicator	Description
`setupFlds`	Fields set in implicit setups or class header.
`usedSetupFlds`	SetupFlds used in test methods.
`adHocFlds`	Fields solely initialized in test methods.
`deadFlds`	SetupFlds fields never used in any test method.
`localVars`	Variables declared in a test method.
`headerInit`	Fields initialized in the class header.

Table II
Smell Metrics and Thresholds

Smell name	**Metric**
General Fixture	$\frac{usedSetupFlds}{setupFlds - deadFlds} \leq 0.7$
Test Maverick	$usedSetupFlds \equiv 0 \wedge setupFlds \geq 1$
LCOTM	$LCOTM \geq 0.4$
Dead Field	$\lvert deadFlds \rvert \geq 1$
Vague Header Setup	$\lvert headerInit \rvert \geq 1$
Obscure In-line Setup	$\lvert localVars \rvert \geq 10$

Measuring General Fixtures. To identify general fixtures, we calculate the ratio between how many `usedSetupFlds` a test method has, and how many `setupFlds` exist in the class. If this ratio is below a certain threshold, we identify it as a general fixture test method. In our current experiments, we set the threshold to 70%. We leave determining the optimal thresholds, for example through benchmarking, for future work.

Measuring Test Mavericks. If a class has an implicit setup, i.e. has `setupFlds`, and a test method does not use any of

the `setupFlds`, we identify a test method as detached from the test class setups.

Measuring Dead Fields. Dead fields are all fields that are initialized by the *implicit setup*, but that are never used by any test method. We differentiate between dead fields inherited by the super class (i.e. inherited fields) and dead fields declared in the class itself. Note that an IDE would not identify `deadFlds`, because an IDE only shows whether a field is never used within a class. Our analysis reveals whether a field is never used by a test method, even if it is accessed within a helper method or the implicit setups.

Cohesion Indicator. To address how cohesive test methods are in a class, we need another smell indicator: fields solely initialized in test methods but not in the setup procedure. We call these fields `adHocFlds`, as they are only created when they are needed. In Listing 1, the field *dir* is an `adHocFld`.

Measuring Lack of Cohesion in Test Methods. To measure how cohesive the test methods of a test class are, we adjusted the Henderson-Seller Lack of Cohesion of Method metrics [10]. Differing from the original metric, we exclusively calculate the cohesion between the test methods in a class and exclude any other methods (e.g. helper or setup methods). For our analysis, we consider all fields of the test class (i.e. `setup` and `ad hoc` fields) that have been used (i.e. we exclude dead fields). We calculate the Lack of Cohesion of Test Methods as the following:

$$\text{LCOTM} = \frac{\frac{1}{|F|} * \sum_{i=1}^{|F|} r(f_i) - |M|}{1 - |M|}$$

Where M is the set of test methods defined by the class, F is the set of `setupFlds` and `adHocFlds` (without `deadFlds`) of the class, and $r(f_i)$ is the number of test methods that access field f_i and f_i is a member of F. As we do not consider `deadFlds`, the metric reports a value between 0 and 1, with 0 indicating no lack of cohesion and 1 highest lack of cohesion. We choose 0.4 as an indicator for a smelly test class.

The LCOTM complements the metric for test mavericks and general fixtures, as it also addresses `adHocFlds` and thus reflects on how strongly test methods differ from each other independent of the fixture.

Obscurity Indicator. The counterpart to the implicit setup, is the *in-line* setup. We measure the obscurity of an in-line setup based on the number of local variables directly defined within a test method (i.e. `localVars` indicator).

Measuring obscure in-line setup. We detect an obscure in-line setup if the number of `localVars` exceeds a certain threshold (i.e. 10 variables per method). The rationale behind this threshold is that with the increasing length of the test method, the primary focus of the test may be hidden. The chosen threshold follows the best practices for the length of a method.

Header Indicator. Finally, the last smell indicator is the fields initialized in the header of the class (i.e. `headerInit`).

Measuring Vague Header Setup. We report this when at least one field is initialized in the header of the class.

Listing 1. Test Class Example

```
class BlobStorageExampleTest extends GitTestCase {
  //setup field
  Repository repository;
  Repository repository2;
  //header initialization
  Storage store = new Storage();
  //ad hoc field
  Directory dir, gitDir;

  @Before public void setUp() throws Exception {
    super.setUp();
    gitDir = new Directory(".");
    //repository depends on gitDir
    repository = new FileRepository(gitDir);
    repository2 = new FileRepository(gitDir);
  }
  ...
  @Test public void testRepository() {
    dir = new Directory(".");
    loadFile();
    ...}

  private void loadFile(){
    repository.getFile("testfile");
    ...}}
```

C. Presentation

This section explains how we present the information gathered in the analysis. We chose to use a navigable hypertext report to present the outcome to the developers, thus supporting a seamless navigation between overviews and details. The report is split into three parts: the fixture classification, the smell overview and the detail improvement report.

Fixture Classification Report. This report provides a list-based overview of the fixture strategies and used framework mechanisms of all test classes. Further it highlights the inheritance structures.

Test Fixture Smell Report. This report provides an overview of the test smells, also in the form of a list, as illustrated by Figure 2. The smells are indicated by an icon and, where relevant, a number showing how often the smell occurred within the test class. To get detailed information about the test class, the developer can click on the test class name and drill into the detail improvement report.

Detail Improvement Report. This report provides detail information on the analysis outcome for a single test class. In the first part of the report, a summary of all smells of the class is given, including a detailed description of the cause. Further, each smell description is enhanced with refactoring suggestions, as illustrated in Figure 3. The second part of the report outlines how fields and helper methods are used within each test method of the class. This part details

Testcase class	No. Tests	No. Flds (Setup/ All)	Dead Flds	LCOM	General Fixture	Test Maverick	Obscure Inline	Vague Header
Test Suite org.eclipse.egit.core--All-Tests								
org.eclipse.egit.core.test.op.AddOperationTest	6	5/ 5		0.0				
org.eclipse.egit.core.GitMoveDeleteHookTest	19	4/ 8		0.04				
org.eclipse.egit.core.test.indexDiff.IndexDiffCacheTest	1	5/ 5	1 deads super: 1/ 3	0.0				
org.eclipse.egit.core.test.op.RemoveFromIndexOperationTest	5	5/ 5		0.0				
org.eclipse.egit.core.securestorage.EGitSecureStoreTest	13	2/ 2		0.0				
org.eclipse.egit.core.synchronize.ThreeWayDiffEntryTest	9	9/ 9	5 deads super: 5/ 8	0.0				
org.eclipse.egit.core.test.GitProjectSetCapabilityTest	4	3/ 3		0.0			2 methods	
org.eclipse.egit.core.test.op.CommitOperationTest	7	7/ 7		0.12				
org.eclipse.egit.core.internal.storage.BlobStorageTest	6	4/ 4		0.5	2 methods	1 detached	1 methods	

Figure 2. Excerpt of the Test Fixture Smell Report for eGit

General Fixture

2 test methods use less than 70% of the fields set during test setup.
Methods affected: testFailNotFound, testFailWrongType,

Minimal Fixture

After removing detached methods from class, refactor the large fixture to a minimal fixture.
Minimal Fixture: repository gitDir
Hint: Only half of the fields are shared. Consider an Extract Class refactoring.
Extract Class 1: testOk
Extract Class 2: testFailNotFound testFailWrongType testFailCorrupt testFailCorrupt2

Setup Detail 4 member fields, 0 static fields

Fields set in Setup	Usage	Inherited
repository	5 out of 6 test methods use this field	
project	3 out of 6 test methods use this field	GitTestCase
gitDir	5 out of 6 test methods use this field	GitTestCase
testUtils	1 out of 6 test methods use this field	GitTestCase

Figure 3. Excerpt Detail Improvement Report for eGit - BlobStorageTest

information on the fixture usage, which is hard to obtain from the IDE and the code alone. It is designed to guide refactoring decisions and to support the developer during the smell assessment.

IV. Implementation and Tool Architecture

TestHound is implemented in Java and supports languages which compile to Java byte code by using the Apache BCEL library to extract facts. *TestHound* supports the JUnit and TestNG test frameworks, but can easily be extended to support other frameworks. Although *TestHound* supports only Java, the analysis is language and framework independent and only the facts extraction aspect is language specific. For the generation of the hypertext report, we use the StringTemplate engine.[3] *TestHound* is available for download[4] and we are in the process of making the source code available on GitHub. In a future release, the tool will be available as a Maven[5] plug-in to facilitate integration with the continuous integration process.

[3]http://www.stringtemplate.org/

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

V. Experimental Design

This section outlines the experimental design of the study, including the research questions, case studies, interviews and questionnaires.

A. Research Questions

To evaluate the applicability and helpfulness of our technique, we set out to investigate the following research questions:

RQ1 What do the structure and organization of test fixture look like in practice?

RQ2 Do fixture-related test smells occur in practice?

RQ3 Do developers recognize these test smells as potential problems?

RQ4 Does a fixture analysis technique help developers to understand and adjust fixture management strategies?

To answer our four research questions, we applied a mixed methods research approach. To answer RQ1 and RQ2, we applied case study research and investigated the code bases of three different Java-based software systems. To answer RQ3 and RQ4, we used interviews and a questionnaire.

B. Case Studies

We use three different subject systems in our experimental design - one closed and two open source systems.

HealthCare: Closed Source Health Care System. The first subject system is developed by a company based in Canada, that offers health care related software solutions. Part of the system is a Java back-end, which provides an API to other

[5]http://maven.apache.org/

systems. This back-end comprises 750K lines of code and has 945 test methods, using the *TestNG*[6] framework.

eGit: Open Source Eclipse-Integrated Version Control System. eGit[7] integrates the Git version control system into the Eclipse IDE. It consists of 130K lines of code and has 479 test methods, all written in *JUnit*[8].

Mylyn: Open Source Task Management System Mylyn[9] provides task management functionality within the Eclipse IDE. It consists of 500K lines of code and has 1644 test methods, written in JUnit.

C. Interviews and Questionnaire

We set out to evaluate our tool and technique by presenting it in a one hour session to a group of 13 professional software developers. These developers worked for the company of the HealthCare system. All developers have experience in writing and maintaining test code, and approximately half of the participants have been working on the code base of the HealthCare system. In this session, we covered the general functionality and purpose of TestHound, as well as the report produced for the HealthCare system. After the presentation, we interviewed five software developers who had contributed to the code base, with each interview taking 30 minutes. During these interviews, the participants could browse through the report produced by TestHound, ask questions and express their opinions on TestHound in depth. We recorded and transcribed each interview.

To capture the opinions of all participants of the presentation, we designed a questionnaire addressing the perception of the audience on software maintenance and the helpfulness of "TestHound". The questionnaire was filled in by all 13 developers and is available online.[10]

Pilot Sessions. To improve the experimental design of the interviews and questionnaire, we conducted three pilot sessions with experienced testers. Two pilot participants were co-workers, and the third participant was the second author of this paper.

VI. Evaluation

A. RQ1: What do the structure and organization of test fixture look like in practice?

This section highlights the basic structure and organization of the test code we analyzed. The results are summarized in Table III.

Package Structure. In all three case studies, the package structure of the test code closely followed the package structure of the system under test. In eGit and the HealthCare system, test code and production code is not separated by an additional package (e.g. test). In contrast, for the Mylyn system, the package "org.eclipse.mylyn.commons.core" is mapped to the package "org.eclipse.mylyn.commons.tests.core". In all three systems, the test code is often mapped to classes. For example, in Mylyn, the class "CoreUtil" is tested by the class "CoreUtilTest".

Table III
Fixture Management Strategies

Project	#test classes	#test methods	Implicit setup		No setup	Tear down
			member	class		
eGit	87	479	56	47	5	79
HealthCare	36	933	26	25	9	25
Mylyn	232	1644	164	0	68	152

In the HealthCare system, this mapping is followed rigorously, and this design decision has a significant impact on the modularity of the test code. The test code of this system consists of only 36 test classes that comprise 933 unique test methods (two of which are parameterized tests). Some of the test classes have more than 4,000 lines of code. For example, one test class comprises 112 test methods and approximately 4,500 lines of code. The Mylyn system consists of 232 test classes that comprise 1,644 test methods. Three of these test classes comprise more than 100 test methods, with a maximum of 172 test methods in the TextileLanguageTest (i.e. more than 1,500 lines of code). In eGit, 87 test classes comprise 479 test methods. The test class with the most tests has 19 tests and 600 lines of code.

Framework Fixture Functionality. In all three systems, the majority of the tests use the *implicit* setup mechanisms of the test frameworks. Interestingly, in the HealthCare system, only the functionality to automatically invoke *implicit* setups, either before one class or one test method, is used. The more fine-grained directives which TestNG offers are not used. In the eGit system, several separate test suites exist and the usage pattern of the *implicit* setup constructs differs: the test suite addressing the core of the system often invokes the setups before each test method, and the fields are mostly non-static. On the other hand, in the test suites addressing the user interface functionality, setups are most often invoked before each class and the fields are static. This design decision is probably due to performance considerations. User interface-related tests often need more setup and involve more expensive resources. In Mylyn, only the *implicit* setups that are executed before each test are used. In all systems, the tear down mechanisms of the test frameworks are used less frequently than the setup mechanisms.

B. RQ2: Do fixture related test smells occur in practice?

Table IV summarizes all smells detected in the three projects, whereby showing the absolute number and the percentage of entities affected by a smell. Each of the smells occurred several times in practice. In the following, we will present some highlights.

General Fixture Smell. The general fixture smell occurred for 32% of the test methods in the HealthCare system, for

[6]http://testng.org
[7]http://www.eclipse.org/egit/
[8]http://www.junit.org
[9]http://www.eclipse.org/mylyn/
[10]http://swerl.tudelft.nl/bin/view/MichaelaGreiler/TestHound

Table IV
FIXTURE PROBLEMS

Project	General Fixture		Test Maverick		LCOTM		Dead Field		Obscure In-line		Vague Header	
	#methods	pct.	#methods	pct.	#classes	pct.	#fields	pct.	#methods	pct.	#classes	pct.
HealthCare	299	≈ 32%	84	≈ 9%	7	≈ 19.4%	180	≈ 33%	100	≈ 10.7%	26	≈ 72%
Mylyn	377	≈ 23%	82	≈ 5%	36	≈ 15.5%	66	≈ 12.1%	17	≈ 1%	35	≈ 15%
eGit	65	≈ 13.5%	17	≈ 3%	12	≈ 13.8%	110	≈ 23.6%	8	≈ 1.6%	79	≈ 91%

23% of the test methods in Mylyn, and for 13.5% of the tests in eGit. An example from the eGit system is *ProjectReferenceTest*. In this test class, none of the setup fields are used by all test methods (i.e. the fixture is disjointed). To improve the class design, an extract class refactoring is recommended. In the HealthCare system, only a few classes contribute the majority of general fixture methods. In particular, three classes comprise 172 of the 299 general fixture methods (i.e. ≈58%). In Mylyn, the largest test class with 172 tests contributes 168 general fixture methods. This class has only two fields, whereby one is only used by three test methods. In eGit, fewer general fixture methods are detected and they are more distributed among classes, as compared with the other systems.

Test Mavericks Smell. Test mavericks occur less frequently than general fixture methods. Also, they are more distributed among classes. In the HealthCare system, the largest class (with 112 tests) contributes the largest set of detached methods (23 methods). In Mylyn, the class TaskListExternalizationTest has the largest number of test mavericks (10 out of 28). In eGit, 4 of the 8 methods in ChangeTest are test mavericks.

Dead Fields Smell. All three systems contains many dead fields, and most of them are are inherited by super classes and not needed. In case fields declared in the actual test class are dead, it often seems to be because of obsolete functionality or open issues. In the HealthCare system, more dead fields exist than in the other systems. There are two main reasons: first, as discussed most tests inherited from only two large super classes and inherit fields that are never used. Second, in this system many static methods are access via the fields, which is unnecessary and often even not recommended. For example, via a field $context$ the static method getBean() is invoked (i.e. context.getBean()), whereby getBean() should be access via the class.

Lack of Cohesion of Test Methods Smell. In the three systems, 14-19% of the classes have a LCOTM value greater or equal 0.4. In Mylyn, a class with high LCOTM (0.8) is the EncodingTest. Each of the test methods in the class uses different combinations of the `setupFlds`. In eGit, an example of a test with high LCOTM is ProjectReferenceTest. Here, all test methods share one field, and in addition, each test method addresses an additional field. In the HealthCare system, the test class with the highest LCOTM (0.89) has two used `setupFlds` that are only used by 4 out of 23 test methods.

Obscure In-line Setup Smell. In the HealthCare system, 10% of the methods contain an obscure in-line setup. The average number of variables declared within these tests is 14.4, with a maximum of 29 variables. In the Mylyn and eGit system, less than 2% of the test methods are reported to have an obscure in-line setup. In terms of test size, for example in Mylyn, a test method testSynchChangedReports in Class BugzillaRepositoryConnectorTest with 24 `localVars` has 113 lines of code.

Vague Header Setup. In the HealthCare system, header initializations occur in 72% of the test classes, and in eGit, in 91% of the test classes. In Mylyn, this smell occurs in only 15% of the test classes.

C. RQ3: Do developers recognize these test smells as potential problems?

During the tool demonstration and interviews, it became clear that developers do indeed recognize the reported test smells as potential problems, and that they see a strong connection between smelly tests and maintenance overhead. In the questionnaire, as illustrated in Figure 4, 12 of 13 developers agreed with the statement that wrong fixture management can lead to code quality problems, and all indicated that improving test quality is important. Only three indicated they they regularly engage in maintenance of test code, whereas four indicated that they do not regularly maintain test code. In the interviews, we investigated why test code is not regularly maintained. All of the interviewed developers said that they had expected their test code base to be very messy. Interviewee number two (i.e. P2) says: *"We know our classes are too large and wrongly focused. We start to write test code and then, next step, we improve."* Soon it became clear that time for the next step is limited, as P1 says: *"We do not have the option to say 'Oh, that's ugly, I'll spend a day to clean it up' if it does not give us immediate business."*

On the other hand, developers express that they are slowed down by the smelly classes (that our tool also identified). P1 says *"If you have to debug a bug revealed by one of these large test classes, then it takes 5 minutes to run and it makes you think 'I don't want to do that anymore'."*

How do these test code quality problems emerge? P1 says that *"people think: 'Ah, this test has to do with a ContainerType', and then add the test to the ContainerTypeTest class, even though it has nothing to do with the other tests in this class."* He adds *"I skip the implicit setup because often the initial setup is made by the first person who created*

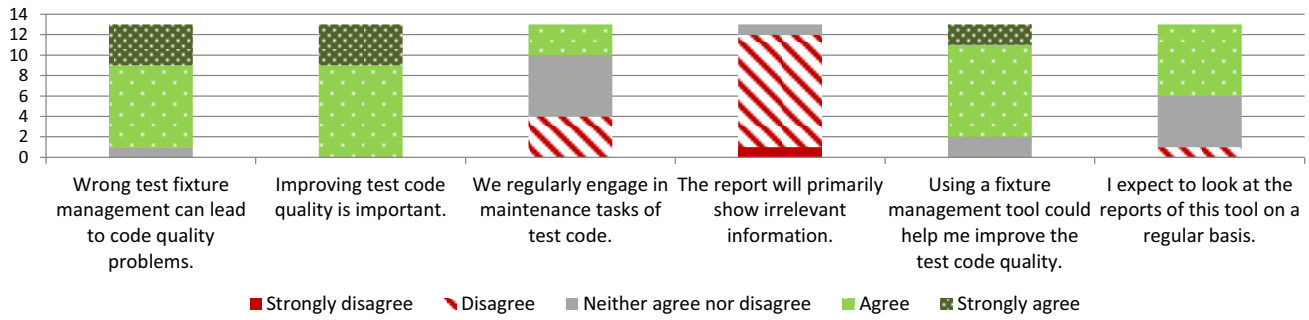

Figure 4. Answers of the questionnaire about maintenance attitude and tool expectations

the class. Maintainers are adding stuff to this setup as they go along, but maybe this is not as common for the other methods and some methods are even unrelated. And then, you make it just in-line, ugly in-line, instead of using the provided framework functionality." Also, other developers explain that they do not look at the implicit fixture because their experience tells them it is often not related to the test methods in a class.

Two interviewees did not see the value of detecting the dead fields smell for their system. P2 explains: *"I am not bothered by the inherited fields of the super class. If two classes have the same functionality, we immediately move this in the super class so they can share it."* Also P3 does not see a problem with this design and says *"I find the information on fields misleading. In our code base, inheritance is used as a convenient way of accessing helper methods."* Even though two participants are not concerned with their design decisions, systems which use inheritance instead of composition to allow code reuse are known to be vulnerable to the fragile base class problem, which hinders maintainability [15]. The other three interviewees refer positively to the identification of the dead field smell.

D. RQ4: Does a fixture analysis technique help developers to understand and adjust fixture management strategies?

The results of the questionnaire show that developers expect a fixture management tool to be helpful during understanding and adjusting fixture management strategies, as illustrated in Figure 4. All participants agree that a fixture management tool could help improve the test code quality and 12 developers think the tool shows relevant information. In the interviews, all developers were positive, as demonstrated by P4: *"I really like the tool. I think it presents a lot of useful information. I think it can definitely be very beneficial for our company."*

Not all developers are sure to look at the test fixture smell reports regularly. In the interviews, developers strongly felt that to allow adoption, the tool must be integrated with the regular build infrastructure. P1 says: *"It's not enough to have the tool run. It should be part of the infrastructure and result in a failed build."* Because time for quality improvement is limited, P3 suggests: *"We are not actively looking for opportunities to improve the code, but one gets the lucky one, when the threshold is exceeded."* He then adds: *"If the tool calls our attention to these problems, we would schedule blocks of time to make the internal quality better."* P5 likes that: *"I cannot get others to review my code all the time. So, a tool that tells me that things look odd, that's good."*

Regarding refactoring test code, P1 says: *"You want to assess how much risk is involved with a refactor. Sometimes you come to that point that you are less likely to change a test because it is smelly. But with the tool telling you that this test did not use anything, then you are more trusting to refactor test code."* P1 mostly appreciates refactorings that can be done easily and quickly: *"It is all about low-hanging fruit: what can you do easily and quickly."*

TestHound is designed not only to indicate smells in the test code, but also to guide the developer during refactoring by providing information on the test fixture usage profile, which is hard to obtain manually. This tool characteristic is also valued by the developers, as P1 says: *"I would look at the test methods and they might look unrelated, but without a lot of digging and work I might not know that they were not relying on anything from the class or the super class."* And P3 especially liked the detail report: *"The summary report is good to get an idea why the build failed, but then I want to go and see these variables in the detail report."* Also, P5 likes the refactoring suggestions: *"The tricky part is to actually understand why something is indicated as smelly. That's the learning part. The refactoring suggestions help. That's interesting to see."*

We also asked participants to list the additional features they would like to see. P4 expresses that the different smells should be integrated in one high-level metric: *"This would gives us an overall assessment, so that if you make some improvements you should see it in the metric."*

Another positive outcome of the evaluation is that developers state the tool makes them think differently about their test code. P5 said: *"I found particularly interesting that the tool made me think about how the tests that I write*

might not be good tests. They do their job, but they may not be maintainable." P3 said: *"The report was definitely very useful. It triggered a lot of ideas to improve and discussions."*

VII. Discussion and Threats to Validity

In the following section, we discuss some of the key findings, observations and threats to validity.

Test Class to Class Convention. A simple way to start organizing test suites is to adopt the *Testcase Class per Class* pattern [14], a commonly used approach that is supported by IDEs like Eclipse which can generate test class templates for a given class. This is at odds with a different pattern, *Testcase Class per Fixture*, in which "we organize Test Methods into Testcase Classes based on commonality of the test fixture." [14]. Our empirical study shows that this pattern is not followed as often as it should, resulting in the smells and maintainability problems we detected. Based on this, we believe it is necessary to rethink traditional mapping strategies and to develop further grouping recommendations and naming conventions, which take into account the evolution of a class as it starts to require more test fixtures.

Frequency of Vague Headers. One might argue that because vague headers occur frequently, they might not be a potential problem. During the interviews, we asked developers to explain the behavior of vague headers. Even though developers are familiar with the test framework and did place vague headers themselves, for several incidences they were uncertain or wrong about the concrete behavior.

Violation of the Single Responsibility Principle. Another observation we made is that the problem of not being able to have a non-smelly test class for a class can indicate a problem with the class under test, such as having too many responsibilities. Sometimes the solution can be not only to split the test class but also to split the class under test into several classes.

Inheritance Structure. In the test code we analyzed, we saw that some super classes are inherited by many test classes. This leads to dead inherited fields, because inherited setup functionality may not be needed. In all three systems, the dead fields are often the same ones (from certain super types), but repeatedly dead for many subclasses. While unused inherited fields are not a problem per se, the large superclass may become fragile (conform [15]), making developers reluctant to adjust it.

Performance Improvement. Based on our case studies, we believe that refactoring of test mavericks and general fixtures can lead to interesting performance improvements, especially considering that with continuous integration, test suites might run several times a day. In a future study, we want to gain a deep understanding of potential performance improvements associated with the application of the suggested refactorings of smelly test fixtures.

Threats to Validity. In terms of generalizability, in its current form, our implementation only works for Java-based systems that use JUnit or TestNG test frameworks. On the other hand, we believe that this technique is not only easily transferable to other xUnit testing frameworks, but also to other languages. Further, our evaluation is limited to three software systems. We chose three systems that are quite different in nature (domain, open versus closed source), and believe that similar results will occur in other software systems. We chose the closed source case study because of the availability of software engineers to take part in the study and its closed source nature. The two open source systems were selected because they are well-known and used software systems. Further, we were familiar with the systems through earlier studies and thus could more easily test that the analysis was accurate. The developers we interviewed also felt this tool could be used to analyze other systems they had worked with.

With respect to internal validity, the analysis may be incomplete or have bugs. To conquer this threat, we implemented many test cases. The developers also indicated the results were consistent with their understanding of the system. Finally, the developers may have been positively biased towards the tool due to the nature of the experimental design. We tried to offset this somewhat by collecting the responses to the questionnaire anonymously.

Our method has some limitations when establishing dependency relationships between setup fields. This can lead to false-positive dead fields. To mitigate the risk of wrong results, we manually inspected all dead fields, and found only a few false-positive cases. For example, in eGit, 3% of the fields could not be mapped to a field usage. For future work, we will enhance the recognition of field usages, and we plan to assess the accuracy of the results in additional case studies. The metrics designed for smell detection are based on field and variable declarations. Actions performed on persistence data storages (such as databases or files) are only detected when a handle (i.e., object reference) is used for access.

VIII. Related Work

Earlier work introducing test smells has been discussed in Section II. Scant research focuses on automatic *detection* of test smells. Among them, Van Rompaey et al. tried to detect the test smells *General fixture* and *Eager test* by means of metrics [20]. In a subsequent paper, they describe a tool which used well-known software metrics to predict a broader variety of potential problems and test smells [3]. Our study differs in several aspects. First of all, we focus on test fixture management and analyze the test code for specific fixture problems that are relevant in practice, and provide concrete refactoring suggestions. In contrast to our work, Borg et al. describe automated refactorings for acceptance tests based on the FIT framework [2]. To the best of our knowledge, fixture-related test smells and refactoring have not been studied in detail so far.

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 [16]. Lanza and Marinescu uses metrics to identify classes that might have design flaws [11], [13].

IX. Concluding Remarks

The goal of this paper is to understand the nature of fixture-related problems in developer test suites. To that end, the contributions of the paper are 1) five *new* test fixture smells, 2) a technique to analyze test fixtures and automatically detect six test fixture smells, 3) an implementation of the technique in a tool called *TestHound*, 4) an investigation of three industrial-strength case studies that shows that test fixture smells occur in practice and 5) an evaluation with 13 developers that shows that the tool is helpful to understand, reflect on and adjust the test fixture.

In our future work, we plan to further research the evolution of test smells and investigate in depth how test class-to-class mappings influence the emergence of test fixture smells. Furthermore, we intend to apply *TestHound* to a range of further systems, broaden the scope of our fixture analysis, and assess performance implications of the proposed refactorings.

Acknowledgments: We would like to thank all interview and questionnaire participants for their time and commitment.

References

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

[2] Rodrick Borg and Martin Kropp. Automated acceptance test refactoring. In *Proceedings of the 4th Workshop on Refactoring Tools*, WRT '11, pages 15–21, New York, NY, USA, 2011. ACM.

[3] Manuel Breugelmans and Bart Van Rompaey. TestQ: Exploring structural and maintenance characteristics of unit test suites. In *1st International Workshop on Academic Software Development Tools and Techniques*, 2008.

[4] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476 –493, jun 1994.

[5] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95, 2001.

[6] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[7] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 1st edition, 2009.

[8] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.

[9] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Test confessions: a study of testing practices for plug-in systems. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 244–254, Piscataway, NJ, USA, 2012. IEEE Press.

[10] Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

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

[12] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111 – 122, 1993.

[13] Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. In *In Proceedings of TOOLS*, pages 173–182. IEEE Computer Society, 2001.

[14] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, May 2007.

[15] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *Proceedings European Conference on Object-Oriented Programming (ECOOP)*, pages 355–382. Springer-Verlag, 1998.

[16] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20 –36, 2010.

[17] Helmut Neukirchen and Martin Bisanz. Utilising code smells to detect quality problems in ttcn-3 test suites. In *In Proceedings of the 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES 2007)*, pages 228–243. Springer, 2007.

[18] Scott R. Tilley, Dennis B. Smith, and Santanu Paul. Towards a framework for program understanding. In *Proceedings Workshop on Porgram Comprehension (WPC)*, pages 19–28. IEEE, 1996.

[19] Bart Van Rompaey, Bart Du Bois, and Serge Demeyer. Characterizing the relative significance of a test smell. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, ICSM '06, pages 391–400, Washington, DC, USA, 2006. IEEE Computer Society.

[20] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Trans. Softw. Eng.*, 33(12):800–817, December 2007.