Characterizing the Relative Significance of a Test Smell

Bart Van Rompaey, Bart Du Bois and Serge Demeyer
Lab On Re-Engineering
University Of Antwerp
{bart.vanrompaey2,bart.dubois,serge.demeyer}@ua.ac.be

Abstract

Test code, just like any other code we write, erodes when frequently changed. As such, refactoring, which has been shown to impact maintainability and comprehensibility, can be part of a solution to counter this erosion. We propose a metric-based heuristical approach, which allows to rank occurrences of so-called test smells (i.e. symptoms of poorly designed tests) according to their relative significance. This ranking can subsequently be used to start refactoring. Through an open-source case study, ArgoUML, we demonstrate that we are able to identify those test cases who violate unit test criteria.

1 Introduction

Over the last few years, the rise of non-traditional methodologies such as eXtreme Programming and Test-Driven Development has resulted in an increasing awareness for software testing in development cycli [2, 3]. Unit testing especially is a cornerstone technique in these methodologies to ensure that the evolving system does not regress. Its effectiveness stems from the frequent and continuous application of the tests throughout the implementation phase.

Because these tests represent normal as well as exceptional paths through the unit under test, they add a considerable amount of test code to the system. This has a serious impact to the development cost of a software project. Moreover, when the unit under test evolves, the corresponding test code should be adapted to reflect those changes, amplifying the maintenance costs.

To make unit testing effective in the long run, the quality of the tests should not only be measured in terms of coverage (i.e. increase the likelihood of detecting bugs) but also in terms of maintainability (i.e. reduce the cost of adapting the tests). Traditionally, the maintainability of a piece of code is assessed by means of internal code metrics (size, complexity, coupling/cohesion) [16]. Unit tests, however, have a typical structure: (1) acquire the necessary resources; (2) send a number of stimuli to the unit under test; (3) verify that the unit responds properly and (4) release the acquired resources. Hence, to assess the maintainability of unit tests, we should exploit this so-called setup-stimulate-verify-tear down cycle.

In this paper we look for poorly designed unit tests which are likely to have a negative effect on the overall maintainability of a software system. We rely on the notion of *test-smells* (i.e. symptoms of poorly designed unit tests) introduced by Moonen and Van Deursen [8] and expanded upon by Meszaros [18]. Inspired by the work of Marinescu [17] we formulate heuristics which measure certain unit test properties and rank the unit tests according to the presence and significance of the test-smells.

After a review of unit testing and test smells in sections 2 and 3, we specify a meta-model for unit testing, based on the concepts of the xUnit family of testing frameworks in section 4. Heuristics for problematic constructs in test code using a metrics based approach are being introduced in section 5, which we then apply in a case study on an open-source system, ArgoUML (section 6). Next, we discuss our findings, compare them with related work in section 7 and formulate a conclusion.

2 Unit Testing Framework

A testing framework is a software library that helps to standardize test specification, execution and reporting. Beck introduced an object oriented unit testing framework based on a pattern system for Smalltalk [1]. Several implementations for other languages have been developed since then, being referred to as the xUnit testing framework family (JUnit 1998, PyUnit 1999, CppUnit 2000, RubyUnit 2001) [13, 18].

Over the years the xUnit testing framework has become a de facto standard. Most integrated development environments have xUnit integration via a plug-in. The framework is referred to in many programming, software development, maintenance, reengineering as well as software

22nd IEEE International Conference on Software Maintenance (ICSM'06)
0-7695-2354-4/06 $20.00 © 2006 IEEE

testing books [3, 7, 19, 14, 11, 15]. The xUnit framework is being adopted by industry as well, by means of JUnit itself or commercial derivatives such as Parasoft's JTest.

2.1 Unit Testing Criteria

Based on a literature survey [1, 7, 10, 11, 18], we present a list of desired unit testing criteria in Table 1. Some of them are taken care of by the testing framework, while the developer carries shared or full responsibility for others. The remainder of this paper will focus on the latter category, as poorly designed tests prevent satisfying one or more of these criteria.

Criteria	Responsible Frwk	Responsible Dev
Consistent – Tests should be standardized in their execution behaviour, reporting and error processing.	X	
Necessary – Every test is needed: it verifies a part of the system specification. There are no duplicate tests.		X
Maintainable – Tests should satisfy essential object-oriented design principles (e.g. abstraction, encapsulation and information hiding).		X
Repeatable – Tests should have repeatable outcomes.		X
Self-Checking – Tests should describe the expected outcome to compare the results against.	X	X
Isolation – The unit is tested in isolation of other units. Accesses to neighbouring units should be replaced by stubs.	X	X
Concise – Tests should conform to a clear and rigorous structure: (1) setup of unit into a desired state, (2) exercising the unit, (3) catching and checking the result and (4) teardown.	X	X
Robust – Test should produce the same result, independent of the environment in which they are run.		X
Fast – Tests should run fast to reduce the turn around time.		X
Automated – Unit tests should run autonomously.	X	X
Persistent – Tests should be written down.		X

Table 1. Unit Testing Criteria responsibility

We subdivide these criteria in three categories capturing the ease of their verification. The first category contains abstract criteria, which are difficult to measure. At the other side of the spectrum are the trivial criteria: these are immediately obvious. In the middle we list concrete criteria. The adherence to these criteria depends on the degree to which the unit testing principles and guidelines are being followed. In this paper, we will support the detection of test smells related to the category of concrete unit testing criteria. More specific, we will focus on the criteria independence, conciseness and performance.

3 Test smells

The concept of a *test smell*, denoting poorly designed tests, has been introduced in a paper by Moonen and Van Deursen in which eleven such constructs are discussed [8]. The proposed smells describe unit tests that make inappropriate assumptions on the availability of external resources, tests that are long and complex as well as tests exposing signs of redundancy. Such tests have a negative impact on both maintainability as well as criteria of test code: at one side complex tests are hard to understand and modify; at the other side, the presence of these smells harms the repeatability, independence and stability of the tests.

Meszaros describes the concept of test smells in a broader domain [18]. In addition to code-level test smells, he distinguishes between project, behavior and code-level smells. Higher level smells are often caused by a combination of lower level ones. Both [8] and [18] describe test smells informally, using a combination of object-oriented and software testing terminology.

In this paper, we contribute a formalization of existing test smells, and demonstrate its application to detect and rank test smell occurrences.

4 Towards a test meta-model

The specification of a meta-model, capturing entities and relationships of interest during unit testing, is facilitated through the consistent terminology in testing literature and supporting tools. Our analysis of existing test smells has indicated that such a model should describe entities like classes, methods and attributes, as well as relationships like containment, inheritance, method invocation and attribute reference.

The proposed meta-model is a refinement of the unified framework for coupling and cohesion measurement in object oriented systems introduced by Briand et al [5]. Amongst others, the meta-model enables to distinguish between test and production code. To be specific, we represent the software system as a set of types, which is decomposed in the following subsets (as depicted in Figure 1):

- *production code*: code developed by the project team that will end up in the released product.
- *external libraries*: the set of software libraries a system uses, but which are not developed by the project. One of the external libraries is the testing framework.

- *test code*: code developed by the project team to conduct developer tests. We subdivide it into test cases and other test types, such as suites, runners and helpers. This code typically does not end up in a production release.

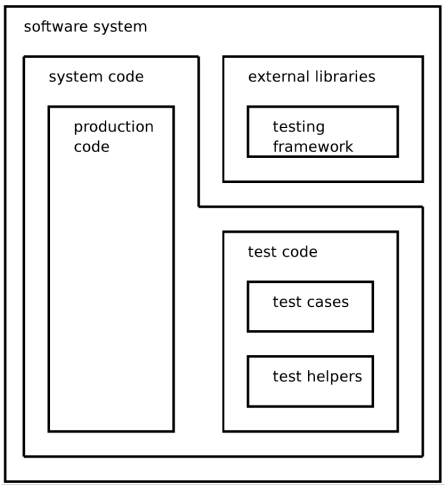

Figure 1. Software system decomposition

Evidently, this informal description still leaves room for different interpretations. Therefore, we formalize these concepts in the remainder of this chapter. In order to cope with minor differences between xUnit implementations, we will introduce some points of variation. Those will be clearly indicated.

4.1 Concepts

In this subsection, we provide a formal foundation consisting of 9 definitions. These definitions enable the formalization of test smells, avoiding potential misinterpretations. Our definitions are based upon concepts introduced by Briand et al. [5], enlisted in Table 2.

For a software system S, we define:

Definition 1: System code C and Library code L – A *software system* S is composed of *system code* C and *library code* L. $S = C \cup L$ is a set of classes.

Definition 2: Unit Testing Framework UTF – An object-oriented *Unit Testing Framework* UTF is an external library $UTF \subseteq L$.

There exists a *Generic Test Case* $gtc \in UTF$ which is a base class for all test cases of the system. $M(UTF) \subseteq M(L)$ is the set of all testing framework methods and is represented as $M(UTF) = \bigcup_{c \in UTF} M(c)$.

The *Generic Test Setup* method gts contains the basic functionality for setting up a system under test into the right state. Formally, $gts \in M_I(UTF)$. Every test case can override this method to add custom setup needs.

A testing framework also contains a set of *Test Framework Check Methods* $TFCM \subseteq M(UTF)$, used to check and report a test's outcome, by comparing the actual result with the expected one. Check methods vary in the expected result, the precision requirement or in the comparison mechanism.

Symbol	Entity
$M(C)$	the set of all methods in the system
$A(C)$	the set of all attributes in the system
$M_I(c)$	the set of methods implemented in c
$Descendents(c)$	the set of descendent classes of class c
$PIM(m)$	the set of polymorphically invoked methods of method m
$NPI(m_1, m_2)$	the set of polymorphically method invocations from m_1 to m_2
$SIM(m)$	the set of statically invoked methods of method m
$NSI(m_1, m_2)$	the set of static method invocations from m_1 to m_2
$AR(m)$	the set of attributes referenced by m
$M_{pub}(c)$	the set of public methods of class c
$uses(c, d)$	A class c uses a class d if a method implemented in class c references a method or an attribute implemented in class d

Table 2. Definitions from Briand et al.

Definition 3: Test Cases TC – Informally: a *test case* groups a set of tests performed on the same unit under test. In an object-oriented system, a test case is a class of the system.

Formally: The set of test cases $TC = Descendents(gtc) \subseteq C$.

- $M(TC) \subseteq M(C)$ is the set of all methods of test case classes and is represented as $M(TC) = \bigcup_{c \in TC} M(c)$.
- $A(TC) \subseteq A(C)$is the set of all test case attributes and is represented as $A(TC) = \bigcup_{c \in TC} A(c)$.

Definition 4: Test code $TEST$ and Production code $PROD$ – Informally: We define *test code* as the set of classes that are either test cases or that access methods or attributes of test cases. All other classes are considered *production code*.

Formally: Let $IM(c) = \bigcup_{m \in M(c)} PIM(m) \cup SIM(m)$ and $AR(c) = \bigcup_{m \in M(c)} AR(m)$. Then,

- Test Code is defined as the union of the set of test cases and all other types that access test case meth-

ods or attributes. $TEST = TC \cup \{c \in C \mid ((IM(c) \cap M(TC)) \cup (AR(c) \cap A(TC))) \neq \emptyset\}$
- Production code is defined as all system code that is not test code. $PROD = C \setminus TEST$.

Definition 5: Test Command (or Test Method) tm **–** Informally: For a system under test in a certain state, a *test command* is a container for a single test. It is typically implemented as a method of a test case containing one or more setup-stimulate-verify cycli. Its name stems from the fact that good unit testing practice advises to create independent tests: there are no parameters that influence the outcome of the test.

Formally: for each class $tc \in TC$ let
- $M_{pal}(tc) \subseteq M(tc)$ be the set of parameterless methods of tc. $M_{pal}(tc) = \{m \in M_{pal}(tc) \mid Par(m) = \phi\}$.
- $M_{pub}(tc) \subseteq M(tc)$ be the set of public methods of tc.
- $M_{tyl}(tc) \subseteq M(tc)$ be the set of typeless methods of tc. A typeless method is a method without a return type (indicated with *void* in programming languages such as Java and C++).
- f is a function expressing the adherence of a candidate test method to a set of imposed criteria.
- $TM(tc) = \{tm \in M(tc) \mid f(M_{pal}, M_{pub}, M_{tyl}, ...) > threshold\} \subseteq \bigcup_{tc \in TC} M(tc)$ is the set of test commands of a test case tc. The threshold in the formalism indicates the introduction of a variation point in our meta model, because the characteristics of a test command vary per implementation [4, 15, 9].

Definition 6: Invocations from Test Commands – Informally: Implemented as a method of a Test Case, it invokes several methods in its body: method calls to acquire the necessary resources, method calls to stimulate the unit under test as well as method calls to verify the status of the changed unit by comparing the actual outcome with the expected one. Figure 2 illustrates the categorization of method invocations in the context of software testing.

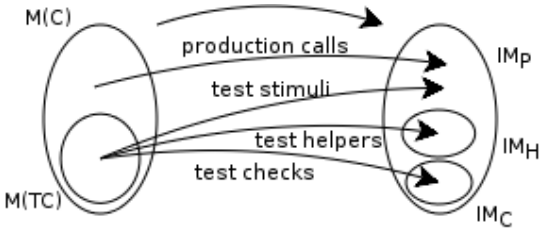

Figure 2. Types of method invocations

Formally: We define the set of methods invoked from within a test commands as $IM(tm) = SIM(tm) \cup PIM(tm)$. We divide $IM(tm)$into three partitions:
- The set of invoked test helper methods $IM_H(tm) = \{m \in IM(tm) \mid m \in M(TEST)\}$.
- The set of invoked production code methods $IM_P(tm) = \{m \in IM(tm) \mid m \in M(PROD)\}$.
- The set of referenced production code attributes $AR_P(tm) = \{a \in AR(tm) \mid m \in A(PROD)\}$.
- The set of invoked test framework check methods $IM_C(tm) = \{m \in IM(tm) \mid m \in TFCM\}$.

We define calls to production type methods method invocations *test stimuli*.

Definition 7: Test Case Fixture TCF **–** Informally: The Test Case Fixture is the set of attributes a test case requires to bring the unit under test into a desired initial state. It contains both instances of the units under test as well as some share data objects.

Formally: for every test case $tc \in TC$ and for every $tm \in TM$ let $TCF(tc) = \{a \in A(tc) \mid T(a) \in S \setminus TEST\}$ be the fixture of test case tc.

Definition 8: Unit under Test UUT **–** Informally: The Unit under Test of a test case is the set of production classes that compose the unit under test, and that as such is exercised by the test case.

Formally: $UUT(tc) = T(A(tc)) \cup PROD$ be the unit under test of test case tc.

Definition 9: Test Case Setup TCS **–** Informally: a Test Case Setup initializes a test case fixture into the desired state for testing. In xUnit, it is a method of the test case which overrides the setup method of the generic test case. This method is called before every test command to reinitialize the test case fixture, resulting in independent tests.

Formally: for each test case tc, let
$TCS(tc) = \{m \in M_{OVR}(tc) \mid m\, overrides\, gts\}$.

5 A detection technique for code-level test smells

In this section we use the definitions introduced previously to formalize two representative test smells: *General Fixture* and *Eager Test*. For each smell, we provide an informal description, discuss its presumed impact on test code maintenance and propose metrics that enable the characterization of a test case with regard to the specific smell.

The General Fixture and Eager Test smells were chosen because they (1) act on different levels (test case and test command) and (2) require a large part of our model in order to be specified (test case, test command, fixture, production and test types, unit under test, test stimuli, test case setup).

In our detection technique, we look for exceptional measurement values for the susceptible test entities, as in [7]. The lack of empirical experience prevents the definition of thresholds for the proposed metrics, which would allow us to compute precision and recall for the proposed technique. However we will show that our technique does yield an incremental way to addresses the worst test smell offenders.

5.1 General Fixture

Informal Description – It is considered good practice to gather test commands exercising the same unit under test together in a test case. This results in a minimal test case fixture which can be maximally reused by all test commands [8]. Therefore, a test case fixture starts to smell when it is larger than this limited set. We distinguish between two types of General Fixtures: (1) the first kind of fixture, a Large Fixture, initializes many objects during setup; (2) a Broad Fixture is a specific case of a Large Fixture: the fixture set consists of objects of many different production types.

Impact on maintenance and unit testing criteria –

- **Large Fixture.** A larger fixture makes it more difficult to understand the state of the unit under test and the purpose of the test commands exercising it. A test case with a large fixture results in more setup and teardown time, and thus negatively influences test performance.
- **Broad Fixture.** In addition to the properties described above, the code of occurrences of this smell risks to be modified frequently through dependencies on production types under test. These dependencies may be a sign that a unit is not tested in isolation or that tests do not logically belong together.

Motivation – The presence of a Large or Broad Fixture can be the consequence of several scenarios:

- When the original unit under test is refactored into multiple classes, the test needs to be adapted to reflect those changes. When the test case is not split up accordingly, the resulting fixture will grow.
- For a certain software project, developers can decide to write unit tests for groups of classes that logically belong together, resulting in fixtures containing a couple of production types.
- When a unit testing framework is used to perform other types of testing, such as integration testing, the fixture will also tend to be bigger.
- A test case may contain multiple instances of the unit under test, to verify the unit's behaviour with different data configurations. When a bug is detected and fixed, a test command using an additional instance can prove that the bug will not be reintroduced [6].

Characterizing metrics – To characterize the fixture of a unit test, we introduce four test case metrics.

For a Large Fixture, we define the following two metrics:

- *The Number of Fixture OBjects* $NFOB(tc)$ as the number of attributes (both implemented as well as visible inherited ones) of a test case tc. Formally, $NFOB =\mid \{a \in A(tc) \mid T(a) \in S \setminus TEST\} \mid$.
- *The Number of OBject Uses in Setup* $NOBU(ts)$ as the number of non-test object uses in the setup ts of a test case. We hereby include those objects used both directly and indirectly by the setup and its test helpers. Let the set of Direct OBject Uses from ts be $DOBU(ts) = (IM(ts) \setminus IM_H(ts)) \cup (AR(ts) \setminus AR(TEST))$; and the set of Indirect OBject Uses from ts be $IOBU_0(ts) = \bigcup_{h \in IM_H(ts)} (IM(h) \setminus IM_H(h)) \cup (AR(h) \setminus AR(TEST))$. In other words, $IOBU_0(ts)$ collects the set of non-test methods and attributes invoked or referenced by helper methods of ts.
  Formally, we define the set of direct test helpers $TH_0(ts)$ of a test setup method ts as $TH_0(ts) = IM_H(ts)$. The set of indirect test helpers with a level $i+1$ of indirection is then defined as $TH_{i+1}(ts) = \bigcup_{h \in TH_i(ts)} IM_H(h)$. This allows us to use induction for $IOBU_{i+1}(ts) = IOBU_i(ts) \cup \bigcup_{h \in TH_i(ts)} IOBU_0(h)$.
  Then, we finally define the number of non-test object uses in the setup ts of a test case as $NOBU(ts) =\mid DOBU(ts) \cup \bigcup_{i=0}^{\infty} IOBU_i(ts) \mid$.

Broad Fixtures can be characterized using the following two metrics:

- The *Number Of Fixture Production Types* $NFPT(tc)$ as the number of production types that the attribute set of a test case consists of. Formally, $NFPT(tc) =\mid UUT(tc) \mid$.
- In addition, we define the *Number of Production Type Uses* $NPTU(m)$ as the number of production type $uses$ (1) from a test method m or (2) from direct or indirect test helpers called from tm. Formally, we define the set of Direct Production Type Uses $DPTU(ts)$ as $\{c \in C \setminus TEST \mid M_I(c) \in IM_P(ts)\} \cup \{c \in C \setminus TEST \mid A_I(c) \in AR_P(ts)\}$. Also, we define the set of Indirect Production Type Uses as $IPTU_0(tc) = \bigcup_{h \in IM_H(tc)} \{c \in C \mid M_I(c) \subseteq IM_P(h)\} \cup \{c \in TEST \mid A_I(c) \subseteq AR_P(h)\}$.
  Then $IPTU_{i+1}(tc) = IPTU_i(tc) \cup \bigcup_{h \in TH_i(tc)} IPTU_0(h)$.
  For every Test Case Setup $ts \in TCS$, we compute $NPTU(ts)$ as $\mid DPTU(ts) \cup \bigcup_{i=0}^{\infty} IPTU_i(ts) \mid$.

Interpretation – Given the full set of test cases accompanied with values for the metrics introduced above, how

do we determine the test cases where the General Fixture smell occurs? We start by distinguishing between those test cases that have an explicit setup method and those that have not. In the former case, the setup method can be checked for a complex initialization involving many objects. A test case with a high $NOBU$ value is an indicator for a complex fixture containing multiple objects of production as well as library types.

For test cases without setup method, we rely on the metrics $NFOB$ and $NFPT$ to reveal a large fixture set. A high value for NFOB is an indicator for a large fixture. In addition to that, a Large Fixture with a high NFPT value is an indicator for a Broad Fixture.

5.2 Eager Test

Informal Description – An Eager test is a test command which checks several methods of the unit to be tested [8].

Impact on maintenance and unit testing criteria –

- Eager test commands are harder to understand because the method body contains more statements.
- Because it is less clear what a certain test command is checking, other tests that cover the same production methods may exist or risk to be introduced. This violates the necessary criterion.
- The sequence of stimulate-verify cycli within an Eager Test command results in dependencies between the cycli.

Motivation – Just like the case of a General Fixture, an Eager Test commands grows when a test command for a refactored production method has not been refactored accordingly. A scenario-based testing approach is another possibility that gives rise to Eager Tests.

Characterizing metrics – We define the *Production Type Method Invocations* $PTMI(tm)$ as the number of production type method invocations in a test command. Formally, $PTMI(tm) = \sum_{c \in PTU} \sum_{m_c \in M_I(c)} NSI(tm, m_c) + NPI(tm, m_c)$ with $PTU = DPTU(tm) \cup \bigcup_{i=0}^{\infty} IPTU_i(tm)$. Furthermore we compute the previously introduced $NPTU$ for every test command.

Interpretation – Test commands with a high value for $PTMI$ are Eager Smell candidates.

5.3 Hypothesis

As the introduced metrics describe aspects of the formalized test smells, comparing the results with a manual evaluation of the test suite under target does not validate the metrics's value. The manual evaluation would be based on the same premises as the automatic detection and would therefore, in that case, compare the formalism with itself.

In constrast, we state the hypothesis that *the proposed metrics are indicators for test cases violating the criteria a test smell has impact on*. Table 3 summarizes the influence of test concepts with high measurement values for the listed metrics on the unit testing criteria. A minus indicates that we hypothesize a negative effect based upon literature on unit testing principles and guidelines.

	NFOB	NFPT	NSPU	NPTU	PTMI
Isolation	-	-	-	-	-
Concise	-	-	-	-	-
Fast			-	-	

Table 3. Hypothesized effect of high metric values on unit test criteria

Therefore, in the case study presented in the next section, we evaluate whether the metrics detected those test cases having the most severe impact on these criteria.

6 The ArgoUML Case Study

In this section we report on a case study in which the proposed test metrics are used to automatically detect test smells. Herefore, we collect measurement results for the test suite of a software system, ArgoUML. Our preliminary validation is based on the manual evaluation of the test cases identified as containing test smells. This evaluation will learn us about the usefulness of the metrics regarding the characterization of test cases.

ArgoUML is an open source UML modeling tool. The system is developed in Java and accompanied by JUnit tests. We checked out the CVS HEAD version at the time of the case study (March 15, 2006). This version consists of 145 kSLOC of Java code, of which about 10 kSLOC is test code.

We selected ArgoUML for this case study because it (1) is a project of considerable size; (2) underwent a substantial evolution since its first release in 1998; (3) has been unit tested with JUnit since 2003; and (4) shares its repository as well as documentation on the Internet.

Using an implementation of the introduced test metamodel, we detected the following amount of test entities (Table 4).

6.1 Large Fixture

NFOB – First, we investigate the number of fixture objects. The right part of Figure 3 indicates the presence of some outliers for this measurement (i.e. test

Entity	ArgoUML
Nr. of System types	2835
Nr. of Production types	1495
Nr. of Library types	1213
Nr. of Test Types	127
Nr. of Test Cases	123
Nr. of Test Commands	332
Nr. of Test Setups	57
Nr. of Test Stimuli	2616
Nr. of Test Checks	332

Table 4. Test Model Entities in ArgoUML

cases with a NFOB value larger than 10, and to a lesser extent those with values 7 and 8). The former two test cases are *org.argouml.uml.generator.TestParserDisplay*, counting 48 fixture objects and *org.argouml.ui.targetmanager.TestTargetManager* with 15 fixture objects. We discuss them in detail.

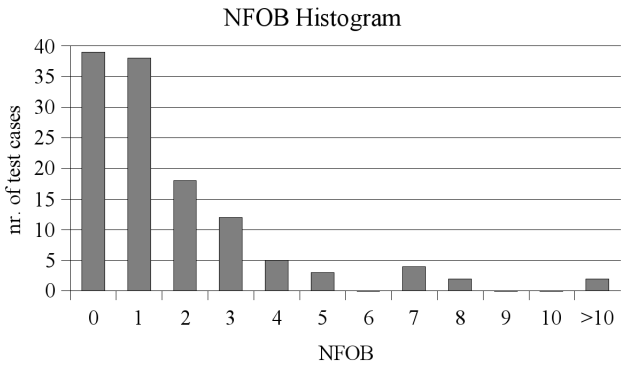

Figure 3. Histogram of the NFOB values for ArgoUML's test cases

A closer look at *TestParserDisplay* reveals that this test case contains a lot of constant strings in its fixture. This test case is also the one with the largest set of test methods (20) and test helpers(13) in ArgoUML's suite. In terms of standard code smells, this class qualifies for the God Class smell [20], and thus has the same maintenance consequences.

This test case violates encapsulation as it consists of three partitions of unrelated test commands, thereby leading to low cohesion. Therefore, we suggest splitting up this test case into smaller test cases, which would benefit the maintainability criterion.

TestTargetManager, the other outlier, has a fixture composed of *java.lang.Object* types. These are used by most test commands, but re-initialized in the individual test commands bodies. In order to improve the isolation of test commands, the initialization could be moved to a common setup, thereby stimulating the conciseness criterion.

Test Case	NOBU
#1	52
#2	46
#3	44
#4	40
#5	34

Table 5. Top scores for NOBU

NOBU – We compute NOBU for the 59 test cases that have a test setup. Table 5 presents the five test case with the highest score.

The first test case requires 52 object $uses$ to bring its unit under test in the desired state. This construct is then a.o. tested for the correctness of this initial state. All three test commands make use of the full fixture.

The four other test cases expose similar behaviour. These tests are a mixture of unit tests and GUI tests. At one side, they focus on one class of the production system and execute fully automatically, although without isolating this class. At the other hand, the units under test are parts of the GUI of ArgoUML, launching and displaying graphical components during test runs. This results in slow tests (more than 20 minutes for the whole suite).

The decision whether or not to rank such test cases as test smells and subsequently refactor them depends on their purpose. In case developers wants to use them for frequent regression tests, they need to be refactored for an increase in isolation and performance. But when they are indeed intended for verifying certain scenarios in the graphical interface, the question is rather whether specifying them in a unit testing framework is the right solution.

When looking into the CVS change history, we notice that the test cases in the *org.argouml.uml.ui.** package have on average been changing more frequently than other subpackages in *org.argouml.model.uml.** or the packages *org.argouml.cognitive.**, *org.argouml.ui.**, *org.argouml.model.**, ... (Table 6). Hence, from a cost perspective they at least increased the maintenance cost.

Test Case Packages	Avg. # of Changes
org.argouml.uml.ui	13.76
org.argouml.cognitive	7.43
org.argouml.ui	6.93
org.argouml.model	6.49
org.argouml.util	5.67
org.argouml.diagram	5.00
org.argouml.uml.[other]	4.24
org.argouml.i18n	3.60

Table 6. Average number of CVS changes for the test cases in ArgoUML's packages

6.2 Broad Fixture

For the detection of broad fixtures, we use two metrics similar to those for large fixtures: $NFPT$ for the test case fixture size and $NPTU$ for type uses in the test setup method.

NFPT – Unfortunately, the first metric does not yield us any information on the presence of broad fixtures (we only get values between 0 and 2 and therefore observe little variation): in ArgoUML, test case fixture attributes are consistently typed as *java.lang.Object*.

This illustrates a weakness of our static test modeling approach for this particular metric: we can not correctly determine the unit under test when statically declared as a supertype in the test case fixture:

1. In case the actual, intended unit under test is declared as a production supertype, the unit under test will not be detected correctly although the metric will report the right value (The supertype still counts as a fixture type).
2. The result is even worse when the unit under test is declared as a supertype that doesn't belong to production code, such as the *java.lang.Object* object hierarchy root class. In that case the $NFPT$ will not detect the fixture object as one of a production type.

Nevertheless, we still think that this metric is valuable, as we have not seen this fixture declaration behaviour elsewhere. Moreover, we do not see an advantage as (1) this manner of fixture declaration makes it harder for the tester to manually determine the unit under test; and (2) polymorphic calls don't rhyme with the isolation criterion. That's why we consider it as a separate type of test smell, which we call *Object Fixture*.

NPTU – This leaves us with the $NPTU$ metric for those test cases containing an explicit setup method. We indeed notice that broad fixtures are a subset of large fixtures, as the top 10 test cases for $NPTU$ are within the top 20 for NOBU.

6.3 Eager Test

Table 7 lists the top ten test commands for the PTMI metric.

PTMI – The test cases of the first three test commands have a common superclass *org.argouml.model.GenericUmlObjectTestFixture*. Each of them tests whether the UML entity that is created by a factory indeed carries the expected characteristics, which are specified in the test cases' setup and verified by a separate method *runTruthTests* in this test case superclass. The *runTruthTests* helper method, called from within every test method, extensively verifies the created UML entities. Therefore, these test cases are tagged as Eager Tests.

While this test design significantly reduces the amount of code that needs to be written, it adds complexity at the same time by introducing another layer of verification methods. The stimuli and the verification calls are situated at different locations in the code, violating encapsulation and therefore the maintainability criterion.

The fourth test command, *testCopyStereotype*, is confirmed to be a Eager Test upon code inspection. It checks whether a UML class can be copied, by verifying (1) that a second class instance can indeed be constructed; (2) that it is not connected to the original one nor its source code; and (3) that a copy from the copy can be made. The "Extract method" solution for an Eager Test, as described by Moonen en Van Deursen, would improve both maintainability, by stimulating abstraction, as well as independence of the stimulate-check cycles, by improving conciseness and isolation. The other test commands of *org.argouml.model.TestCopyHelper* have a similar structure and thus a similar solution.

6.4 Lessons Learned

During the ArgoUML case study, we were able to pinpoint severe offenders of the addressed test smells. Code inspection and developer documentation revealed that the ArgoUML JUnit test suite is not entirely composed of pure unit tests. Although the ArgoUML test suite targets one production class per test case, units are often not setup in isolation; the GUI tests even launch graphical components. When we computed the introduced metrics for these test cases, we ended up with measurements indicating severe violations of unit test criteria. The purpose of these tests must drive the solution: refactoring towards tests meeting these unit test criteria, or keeping them as a less frequently run regression suite. In the latter case the maintenance effort will stay high.

Besides the detection of actual test smells and the valuable insight in test code it yields, we encountered mainly two limitations of our current approach.

First of all, the static approach prevents us from detecting the correct dynamic type of all objects under test. E.g. in the case a test fixture consists of attributes declared of type *java.lang.Object*, parts of the actual unit under test can not be identified. Therefore, we are looking into ways to resolve this issue, e.g. through the incorporation of runtime information. Although the extensive use of such declaration characteristics appears, in a first comparison with some other open source projects (e.g. JUnit, PMD, etc.), not to be a general practice, we will investigate this limitation and its

Test Command Report	PTMI
org.argouml.model.TestUmlModel.testNamespace	70
org.argouml.model.TestUmlActor.testActor	70
org.argouml.model.TestUmlUseCase.testUseCase	70
org.argouml.model.TestCopyHelper.testCopyStereotype	36
org.argouml.cognitive.TestItemUID.testAssignIDsToObjects	36
org.argouml.model.TestCopyHelper.testCopyClass	35
org.argouml.model.TestCopyHelper.testCopyInterface	34
org.argouml.model.TestCopyHelper.testCopyDataType	33
org.argouml.model.TestCopyHelper.testCopyPackage	29
org.argouml.kernel.TestProject.testDeleteOperationWithStateDiagram	25

Table 7. Eager Tests

impact in future work.

A second limitation is that currently, we focus on violations of unit test criteria in test cases employing an xUnit test framework. However, as in our case study, such frameworks are also used for other kind of tests. For JUnit, there exist extensions for e.g. GUI testing (JFCUnit) and functional testing (JUnitScenario). We encountered test cases that balance on the boundary between unit tests and GUI tests.

Obviously, a single case study is not sufficient to make any hard claims about the general applicability of the approach. Nevertheless, our case study illustrates that it is worthwhile to further explore this research direction. As the next research step, we will address a more rigorous validation of the proposed test metrics, through describing essential mathematical properties which these metrics should satisfy, but also, through quantifying their effectiveness (in terms of precision and recall) in indicating test smell occurrences.

7 Related Work

Gaelli et al. [12] propose a taxonomy of test commands based on (1) the focus on either one or multiple methods; (2) method call frequency; and (3) expected outcome (optimistic or pessimistic). Their lightweight heuristical approach, using naming conventions and method call characteristics as guidelines, succeeds in automatically categorizing test commands with a high precision rate.

However, their work only studies the relation between test commands and unit under test, as they found that a majority of unit tests focus on a single method. In our case study, however, we noticed that only 24% of the 332 test commands focus on a single method. Moreover, for the detection of test smells via metrics-based heuristics, we need to look at other relations as well, e.g. the relation between test cases, their fixtures and test commands, etc.

8 Conclusion

In this paper we have proposed a heuristical approach which allows to assess the relative significance of two test smells (General Fixture and Eager Test), according to violations of unit test criteria. The heuristics are based on a number of metrics for measuring certain properties of test code, which have been precisely defined by means of a metamodel for unit tests. We validated the approach by means of a case study, showing that we do indeed identify test cases violating unit test criteria. However, some of the identified tests could be, depending on their purpose, considered false positives because they represented cases where the JUnit testing framework was not used for pure unit testing. Nevertheless, all identified tests have a negative effect on the maintainability of the system, because the test case structure makes it hard to understand and complex to change or CVS inspections shows that they have been changed more often.

Summarizing, in this work, we have provided a first step towards the quantitative analysis of test code. As indicated by our case study, test metrics like the ones presented in this paper have the potential to detect occurrences of relevant test smells. Unit test cases, exhibiting these test smells, are prime candidates for refactoring, thereby minimizing the maintenance costs for test code.

9 Acknowledgements

This work has been sponsored by Eureka $\sum$ 2023 Programme; under grants of the ITEA project if04032 entitled Software Evolution, Refactoring, Improvement of Operational & Usable Systems (SERIOUS).

References

[1] K. Beck. Simple smalltalk testing: With patterns. *The Smalltalk report*, 4(2):16–18, 1994.

[2] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
[3] K. Beck. *Test-Driven Development By Example*. Addison-Wesley, 2002.
[4] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 7(3):51–56, 1998.
[5] L. Briand, J. Daly, and J. Wuest. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.
[6] D. E. DeLano and L. Rising. Patterns for system testing. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pages 503–527. Addison-Welsey, 1998.
[7] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
[8] A. Deursen, L. Moonen, A. Bergh, and G. Kok. Refactoring test code. In M. Marchesi and G. Succi, editors, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, 2001.
[9] G. Doshi. Junit 4.0 in 10 minutes, 2006. online at http://www.instrumentalservices.com/index.php?option=com_content&task=view&id=45&Itemid=52 (accessed March 22, 2006).
[10] E. Dustin. *Effective Software Testing. 50 Specific Way to Improve Your Testing*. Addison-Wesley, 2003.
[11] M. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2005.
[12] M. Gaelli, M. Lanza, and O. Nierstrasz. Towards a taxonomy of SUnit tests. In *Proceedings of ESUG 2005 (13th International Smalltalk Conference)*, 2005.
[13] P. Hamill. *Unit Test Frameworks*, chapter Chapter 3: The xUnit Family of Unit Test Frameworks. O'Reilly, 2004.
[14] A. Hunt and D. Thomas. *Pragmatic Unit Testing in C# with NUnit*. The Pragmatic Programmers, 2004.
[15] J. Langr. *Agile Java Crafting Code with Test-Driven Development*, chapter Lesson 15: Assertions and Annotations. Prentice Hall, 2005.
[16] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, February 1993.
[17] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society, 2004.
[18] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, to be published in 2007. online version at http://xunitpatterns.com/ (accessed March 21st, 2006).
[19] M. Pelgrim. *Dive into Python*. Apress, 2004.
[20] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.