JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 6, No. 9, 2007

Rule-based Assessment of Test Quality

Stefan Reichhart, Tudor Gîrba
Software Composition Group, University of Bern

Stéphane Ducasse,
LISTIC - University Annecy de Savoie

With the success of agile methodologies more and more projects develop large test suites to ensure that the system is behaving as expected. Not only do tests ensure correctness, but they also offer a live documentation for the code. However, as the system evolves, the tests need to evolve as well to keep up with the system, and as the test suite grows larger, the effort invested into maintaining tests is a significant activity. In this context, the quality of tests becomes an important issue, as developers need to assess and understand the tests they have to maintain. In this paper we present TestLint, an approach together with an experimental tool for qualifying tests. We define a set of criteria to determine test quality, and we evaluate our approach on a large sample of unit tests found in open-source projects.

1 INTRODUCTION

Testing is an important activity in the development of today's software projects [1, 2, 3]. Automated tests (*e.g.*, unit tests) help the developer to assure code quality and to detect possible bugs and flaws in the application code [13, 14, 22]. Furthermore, tests can also be seen as live documentation, and be used to understand foreign code [7].

Following an agile development process, the body of tests grows together with the source code. However, due to refactorings and changing requirements, code might start to erode [10, 25]. The same quality erosion also happens to test code: it becomes long, complex and obscure [31]. Although such tests might still serve the purpose of checking the correctness of the system at present time, they can easily break when further adaptations to the application code are required.

A large body of research has been carried out to assess the quality of tests from different perspectives:

- Code Coverage provides a quantitative measure [19, 27, 36].
- Mutation Analysis gives insights to code stability [17, 23, 24, 34, 35].
- Test Ordering shows the interconnection of tests [12, 15, 16, 26].

Cite this document as follows: Stefan Reichhart: *Rule-based Assessment of Test Quality*, in Journal of Object Technology, Special Issue TOOLS Europe 2007, 2007.

Although all those methodologies differently contribute in assessing the quality of tests, they all do it on a rather abstract level, and they do not focus on the actual test code.

We propose an approach to analyze the code found in the test implementations (*i.e.,* test-classes and test-methods[1]) to identify problems that influence their maintenance. The decayed parts of the application code are often referred to as Code Smells [11, 32]. In the same way, Test Smells refers to test code that is difficult to maintain [22]. As only little research has been spent on understanding and detecting Test Smells [8, 30, 31], we chose to systematically study a large set of tests and to learn the different characteristics that influence readability and maintainability.

The basis of our research is a case study consisting of 4834 test-methods and 742 test-classes taken from the Squeak[2] open-source community. Our study was conducted in three steps:

1. The first step was to harvest the tests and collect a list of problems found in the tests through manual inspection. Due the large number of tests we did not analyze all of them, but rather focused on a *sample* of approximately *500 test-methods* that were known to be good or bad based on input from the Squeak community.

2. In the second step we clustered the problems to identify commonalities and differences, and we distilled the lessons in automatic queries that we implemented in a tool called TestLint. We have employed several techniques: static analysis of the test code and dynamic analysis including code manipulation and instrumentation.

3. In the third step we have applied our queries on *all the tests* in our case study and manually inspected the detected Test Smells to identify false positives.

The result of our approach is a list of 27 abstract and fine-grained Test Smells that we empirically collected and checked against our case study. We believe that the automatic analysis of the quality of tests is an important activity in the development and testing process, and we see our list of Test Smells and detection rules as a starting point for a test analysis tool.

The rest of the paper is structured as follows. Section 2 briefly explains Test Smells, and Section 3 details our approach to detect them. Section 4 reveals our list Test Smells, and provides details for a selection of them. In Section 5 we present the results of our case study. In Section 6 we present the related work, and we conclude in Section 7.

[1]By test-class we refer to a class inheriting from TestCase, and by test-method we refer to a method that encodes a particular test (typically those that have a "test" prefix).

[2]Squeak is a Smalltalk dialect. For further details, please check: http://www.squeak.org/

2 Test Smells IN A NUTSHELL

Test Smells are, in general, signs of flaws in the design or code of a test. They describe tests that are too long, complex, include unnecessary redundancy, exposing or breaking encapsulation of the application code, run unnecessarily slow, or make inappropriate assumptions on external resources. The consequence of such tests is that they are hard to understand, hard to maintain, and they badly document the application. Furthermore, they typically become unstable, or even become unused or deprecated.

Test Smells mostly appear due to frequent changes in the application code. A factor that makes the test code brittle to changes in the application code is the duplication in test code, as this leads to developers not updating all the tests. Further aggravating factors for the maintenance of tests are the complexity of the code to be tested, and the lack of time of the developers.

Below we give three examples of Test Smells as can be found in the work of Meszaros [22]. A more detailed description together with code examples are given in [27]:

- An *Eager Test* is a test that verifies too much functionality. It is mostly, but not necessarily, a test with a large amount of statements and assertions. It is normally difficult to understand, and it offers a poor documentation.
- *Conditional Logic* breaks the linear execution path of a test, making it less obvious which parts of the tests get executed. This increases a test's complexity and maintenance costs.
- A *Large Fixture* provides a large amount of data to the tests, making it difficult to understand the state of a unit under test and also obfuscating the purpose of the tests. Furthermore setup and teardown require a large amount of time slowing down the execution of the tests.

Test Smells are a recent research topic but have attained interest, especially from researchers being active in research about refactoring code [6, 9, 11, 18, 29] and tests [8]. Test Smells have been categorized and described informally [8]. Meszaros [22] decomposed and further subdivided them, explaining the reasons for their appearance as well as their consequences. Some Test Smells were formalized using heuristics and metrics [20, 21] to gain insights about their significance [30, 31].

3 TestLint

Our approach to detect Test Smells started by first collecting a large set of tests from SqueakSource, an open-source code repository of the Squeak community. We have selected 67 packages containing 4834 test-methods and 742 test-classes.

In a second step we systematically analyzed the tests, starting with packages known to have very good and very bad code or tests. During this analysis we created an extensive list of common problems. We then categorized the detected problems into already known Test Smells, and also formalized new types of smells or extended existing ones. Table 1 presents a small selection of all Test Smells we gathered and checked in this paper's case study.

Based on our analysis we determined that most Test Smells consist of several diverse smelling aspects. For example, an *Eager Test* is a test referencing and executing many different methods, but it can also be long and include many comments. We also noticed that Test Smells are partially interconnected, as they share some of their characteristics with one another:

- *Obscure* tests are hard to understand and badly document the application due to their complexity.
- *Long* tests are most of the times *obscure*. However, *obscure* tests are not necessarily *long*.
- *Erratic* are those tests that alternate the result of the test. Erratic tests are also *obscure* tests, but rather seldom the other way around.

We implemented an approach to automatically detect a set of smells in a tool called TestLint. TestLint contains a rule-based engine, similar to Smalllint [9]. We decided for such a rule-based technique as it allows us to analyze Test Smells in a fine-grained and flexible way. Each rule in TestLint can either map to a complete Test Smell as known from the literature [8, 22] or just to a part of its characteristics. Furthermore, due the interconnection of Test Smells a rule can be reused in the detection of several Test Smells.

We characterize the rules according to different aspects. From the point of view of the analysis goal, a rule can be applied to either test-methods or test-classes. From the point of view of the analysis, a rule can encode a static analysis, can encode a dynamic analysis test-method (*i.e.,* they run the test to gather additional information during runtime), or can combine static and dynamic analysis.

4 TEST SMELLS

In this section we give a brief overview of Test Smells and the rules we define to detect them. A first section covers static smells, and the second gives examples of dynamic ones. We give a short description for each smell. In particular we summarize the smelling aspects and explain how they appear in the code, what consequences they might have and how we detect them using TestLint. As we performed our experiment in Squeak we give examples using Smalltalk syntax.

During our process of gathering and analyzing Test Smells we implemented approximately 70 partially overlapping rules to automatically identify Test Smells. Table 1 gives an overview of 27 smells for which we have formalized useful and robust detection rules, and which we validated in our case study. An extensive list of Test Smells and rules including more detailed descriptions is presented in [27].

Test Smell	**Description**
Guarded Test	conditional test including branches like ifTrue:aBlock or ifFalse:aBlock
Overreferencing	test creating unnecessary dependencies and causing duplication
Assertionless Test	pretending to assert data and functionality, but does not
Proper Organization	violating testing conventions by using bad organization of methods
Test-MethodCategory Name	method categories having a meaningless name
Anonymous Test	test-methods having a meaningless name
Long Test	tests including too many statements
Mixed Selectors	violating common organizational testing conventions by mixing up testing and non-testing methods
Likely ineffective Object-Comparison	objects comparisons which can never fail
Unclassified MethodCategory	methods not being organized by any method-category
Test-Class Name	test-class having meaningless name
Unused Shared-Fixture Variables	parts of the fixture that are never used
Early Returning Test	test returning a value and too early, maybe dropping assertions
Unusual Test Order	tests calling each other explicitly (unusual for unit tests)
Under-the-carpet Assertion	some assertions put into comments
Comments Only Test	all test-code put into comments
Overcommented Test	test having too many comments
Under-the-carpet failing Assertion	failing assertions put into comments
Control Logic	test controlling the execution flow by using methods like *debug* or *halt*
Max Instance Variables	large or oversized fixture
Teardown Only Test	test-suite only defining teardown (unusual for unit tests)
Abnormal UTF-Use	test-suite overriding the default behavior of the unit testing framework
Empty Shared-Fixture	fixture defined, but empty
Transcripting Test	test writing and logging to the console
Empty MethodCategory	empty method categories
Returning Assertion	assertions returning a value (unusual for unit tests)
Empty Test-MethodCategory	empty testing method categories

Table 1: A selection of Test Smells used in our analysis and case-study (see Figure 6)

Static Smells

TestLint handles static smells by scanning for specific patterns. It parses the code, analyses the source tree to detect specific node items, and also computes metrics on the test code. This analysis can be done without actually running or instrumenting the test.

Assertionless Test. This rule checks whether a test contains at least one *valid assertion*. A valid assertion is either one provided by the underlying unit testing framework or is a user defined one that is composed of valid assertions. The following code shows an example of a valid user defined assertion, containing an assertion provided by the unit testing framework:

```
UserDefinedTestCase>>userDefinedNotNilAssertion: anObject
  self assert: anObject isNil not
```

A test not containing at least one valid assertion is just a piece of executable code that can either succeed or throw an error but can never throw an assertion failure, unless thrown explicitly, which should not be done. Such a test is a weak one, because the only thing it tests and documents is that the code of the application does not throw an error for a particular run. The following example shows an *Assertionless Test*:

```
ICCreateCalendar>>TesttestCreatingSeveralCalendars
  self addCalendarWithName: 'new Calendar 1'.
  self addCalendarWithName: 'new Calendar 2'.
  self addCalendarWithName: 'new Calendar 3'.
  self addCalendarWithName: 'new Calendar 1'.
  self addCalendarWithName: 'new Calendar 2'.
  self addCalendarWithName: 'new Calendar 3'.
```

We can detect most of those tests by statically analyzing the parse-tree, including all referenced methods. If none of them is known as a valid assertion to the system, then we probably found an *Assertionless Test*. False positives might appear in a dynamic language like Smalltalk as we cannot retrieve the implementor of a method by doing a static analysis. A dynamic analysis could eliminate this uncertainty.

Guarded Test. This rule is detecting tests implementing conditional branches, *e.g.,* ifTrue: or ifFalse:.

The consequence of using branches could be that certain assertions are not executed. In the worst case an actual failing test would return a success letting the developer believe the test is green. Also, conditionals are problematic because they make the test less predictable, and harder to understand.

We identify *Guarded Tests* by scanning the abstract syntax tree of a test-method for all occurrences of any conditional logic. The following code-example would be detected as a *Guarded Test*:

```
testRendering
  self shouldRun ifFalse: [ ^ true ].
  self assert: ...
  ...
```

We further notice at this point that many projects we harvested use conditionals on purpose to drop slow tests, tests that are platform-dependent or tests that can only run in a special environment. As this is always done in the same way we can identify this need for *Conditional Tests* as a missing *design pattern* for unit

testing. Therefore *Guarded Test*, when used by such purposes, can be regarded as false positives. However because of the following reasons we regard the way these tests are realized as a Test Smell:

- Conditional code is repeated like a pattern all over the tests and pollutes the test code, making a test harder to comprehend.
- Standard unit-testing frameworks are not aware of the condition and its meaning and purpose. Therefore conditional tests always return a success although they did not run any code or assertions.

We therefore recommend to extend any unit testing framework to become aware of such tests and the condition under which they can be executed, without the need of polluting the test code with conditional branches. Like that a developer would be notified which and why tests were not executed.

Overreferencing Test. This rule is testing whether a test is referencing many times classes from the application code.

The main problem with an *Overreferencing test* is that it distracts from the goal of the test. In our experiments, such tests were rather long and obscure. We have also detected overreferencing as a source for subtle duplication in the test code: slightly different fixtures are present in different test-methods.

Figure 1 shows how overreferencing can be detected. The first condition checks for referenced types. The second one counts how often the same type is referenced in the code. From our experiment, we have found 3 to be a good value for the thresholds.

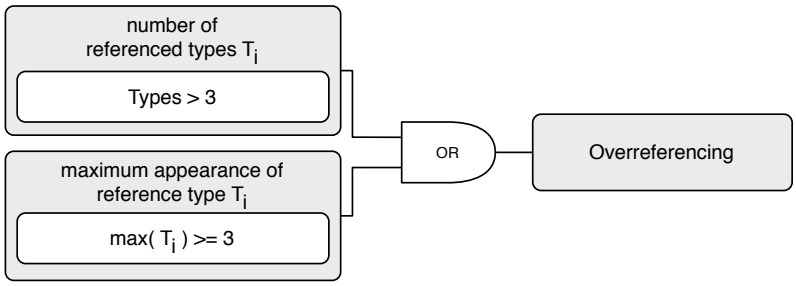

Figure 1: The Overreferencing Test rule.

Applying this rule on the example below, we would get the value 3 for the first condition and 3 for the second one. Therefore the test would have the problem of *Overreferencing*:

```
testLibrary
  track1 := MP3Track new.
  track2 := WAVTrack new.
  track3 := MP3Track new.
  track4 := MP3Track new.
  library := Library new.
  ...
```

Anonymous Test. This rule is analyzing the signature of a test-method to find out whether the test has a meaningful name within its context.

Anonymous Tests are Test Smells as they do not or only badly document the goal of a test, making the test-suite obscure. Furthermore such tests require the developer and tester to read all the code to understand its purpose, wasting unnecessary attention. Therefore all tests should have a good name documenting what they are about.

For that, we split the test-method name following sequences of numbers and literals including camel-case notation. We then check whether *all* the obtained tokens are found in the names of the application classes or methods.

As an example, the rule to detect *Anonymous Tests* rejects method names like test1 to test31, but might allow testSHA256 if its context defines it. The context could be the current package defining a class or method whose name is similar to SHA, 256 or SHA256.

False positives are inevitable as the analysis of names and their meaning heavily depends on the context and the algorithm used for calculating similarities. In our implementation we apply a basic pattern matching on each part of the name within the context. As context we use the package under test.

Dynamic Smells

Dynamic smells require the test to be run. For most rules it is already enough to run the test once, but there are also rules that require to run the test multiple times. Almost all dynamic smells also include static analysis. For these rules we instrument parts of the application code, the fixture or the tests. The instrumentation includes an uncertainty factor as not all code can be successfully instrumented.

Under-the-carpet Failing Assertion. This rule is checking whether a successful test might contain *hidden failures* that have been commented out. This rule parses the source-tree for assertions put into comments, safely removes any comment-tokens around valid code and asserting statements and runs the whole test again. If it fails a hidden failure has been detected (see Figure 2).

For example, the following test shows an *Under-the-carpet Assertion* that raises

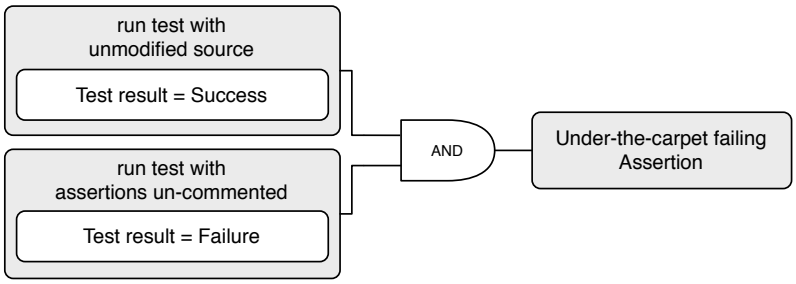

Figure 2: Schematics of Under-the-carpet failing Assertion

an Error if we removed the comment-tokens.

```
ICImporterTest >> #testImport
  ... self assert: eventAtDate textualDescription = 'blabla'."
  self assert: eventAtDate categories anyOne
        = (calendar categoryWithSummary: 'business').
  "self assert: ...
```

At a closer analysis we find out that the method categoryWithSummary: can throw an Error if aString is not detected in categories.

```
ICCIaendar >> #categoryWithSummary: aString
  ^ self categories detect: [ :each — each summary = aString]
```

This rule is important as developers obviously often comment out failing assertions due to several reasons. We have encountered assertions that were commented in the course of debugging activities, and later got forgotten to be removed. We have also detected commented assertions that were just obsolete and tests that were completely commented because they run slowly, require a special environment, or just for the sake of making the test green.

Badly Used Fixture. This rule is instrumenting all methods inside a test-class including tests and instance variables of a test-case to find out which method is actually reading or writing (directly and indirectly) which instance variable while a particular test is running. Figure 4 shows an example of a badly used *shared fixture*.

Having this information about the shared fixture it is possible to make a very precise analysis about its usage and necessary refactorings *e.g.,* to decide whether it is sufficiently used by the tests, used at all or whether a fresh fixture might be more appropriate.

The first simple condition in Figure 3 says that at least 75% of all tests should use the fixture. The second condition requires that on average all tests should use

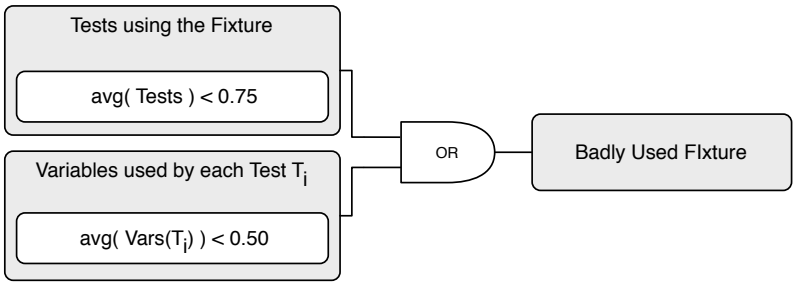

Figure 3: Schematics of Badly Used Fixture

at least 50% of the variables defined within the fixture. When we apply this to the example of Figure 4 then the first condition would evaluate to 0.25 (=1/4), same for the second condition (=(0+0+1+0)/4)

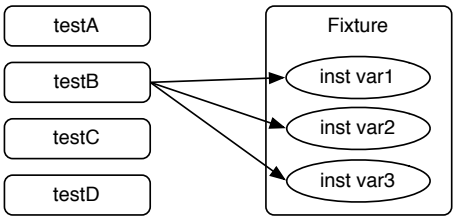

Figure 4: Example: Tests using the shared fixture

False positives cannot be totally excluded as the purpose of an instance variable is difficult to estimate, therefore this rule is quite sensitive to the context of the test. Furthermore the threshold we've chosen for both conditions is rather restrictive and might not work for all situations. For example we might encounter a false positive for very abstract high level test requiring a large fixture.

5 CASE STUDIES

The basis of the case study is a Squeak 3.9 distribution with Christo Code Coverage [27] and TestLint [27] installed. During the validation phase we analyzed *all the tests* methods that we extracted from 67 packages containing a total of about 4834 test-methods originating from SqueakSource, an open source code repository for Squeak.

Overview

To get a first overview on the detected Test Smells, we collected all smells for each test in the case study by applying all available TestLint-rules to the tests. We then clustered the tests by the number of smells detected (see Table 2).

Smells per Tests	0	1	2	3	4	5	6-9
Distribution	61 %	20 %	11 %	4 %	2 %	1 %	< 0.5 %

Table 2: Distribution of Smells per Test

The result shows that 39% of all tests are affected by at least one detected Test Smell – possible false positives included. Furthermore we discover that most tests have exactly one issue (20%), followed by tests having two (11%) and so on. Tests having more than 4 problems can be regarded as an exception.

The reason why a test can be affected by multiple smells is due to our fine-grained rules. For example, a test might have commented assertions, might be complex and it can contain guarding conditionals in the same time. The more problems show up in a test the more likely it has a serious problem. However, we could not conclude the opposite from our experiment.

From the manual inspection we identified the false positives. Figure 5 shows the success rate of a selection of rules, sorted descending by the total number of detections.

First, we determine that certain rules (*e.g., Anonymous Test*) work better than other ones (*e.g., Proper Organization*). However, most rules produce false positives. This can have multiple reasons: the rules are not well enough specified or the contextual sensitivity cannot appropriately be addressed.

We also notice that the clearer the goal of the smell, the more precise the rule. For example, *Comments Only Test* has no false positives as it is easy to make a very precise formalization. As another example, the detection of *Proper Organization* is less precise than for *Anonymous Test* as there is no clear understanding for what makes a good organization for tests.

Second, we discovered that Test Smells are not equally distributed. There are smells that appear more often than other ones. For example most test code implements more problems like *Assertionless Test* than *Under-the-carpet failing Assertion* which means there are more tests detected without assertions than such with a failing assertion put in comments.

Based on the data gathered by TestLint we can also map detected Test Smells to each package in the case study. Figure 6 shows a small selection of well known packages, sorted descending by the density of smells per test on the x-axis. As we are interested in a quality measure for the packages we cut the y-axis at one smell per test.

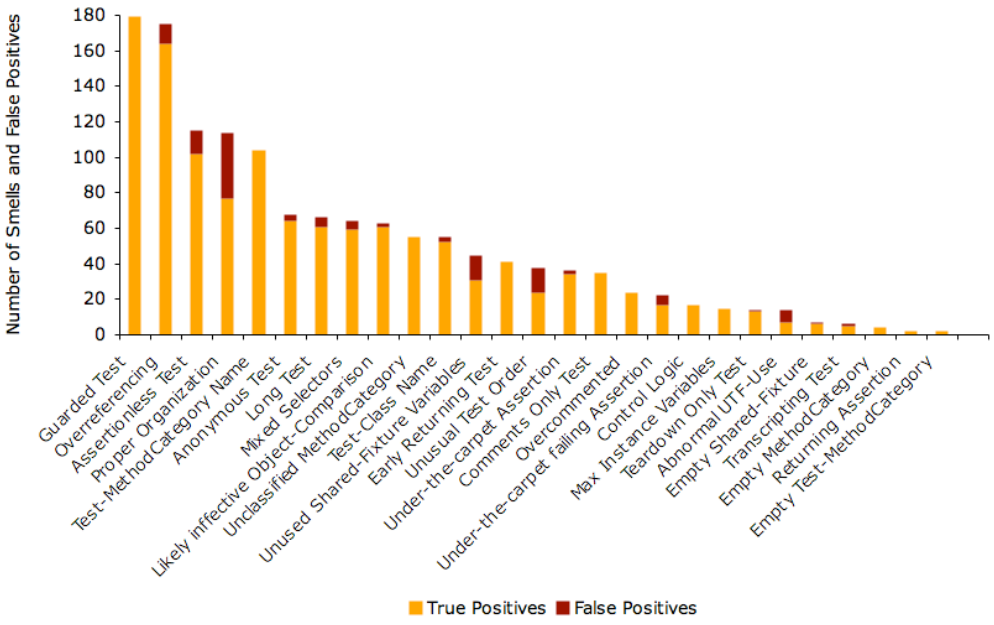

Figure 5: Detected Test Smells (orange) by rules including false positives (dark red). The list of Test Smells together with a short description can be found in Table 1.

We discover packages with qualitatively good tests (on the right side, below the x-axis), having less than one smell per test, and ones with rather problematic tests (left side, above the x-axis), having more than one smell per test. We determine that the results shown in Figure 6 mostly and in general conforms to our earlier manual analysis of the tests as well as the input from the community. As an example, the packages *Magritte* or *Aconcagua* are known to have good tests whereas packages like *ToolBuilder* or *SMBase* are rather regarded as hacks, not containing very good or reliable tests.

We go a step further and sort all packages by the number of tests and correlate this number with the number of totally detected Test Smells, shown in Figure 7. We enrich the graphics with exponential curves to globally approximate tests and Test Smells.

An interesting fact is that packages defining more tests also have more smells. However the exponential approximations show that the amount of smells per test is increasing less than the amount of tests written. Based on this we might conclude that developers writing more tests, therefore having and acquiring more testing experience, also tend to write better tests.

To check this assumption, we show in Figure 8 a similar graphic, only in this case showing the amounts of tests and Test Smells per author. The diagram of shows the relation between the number of tests (primary x-axis), number of smells per tests (secondary x-axis), and the authors writing them sorted ascending by

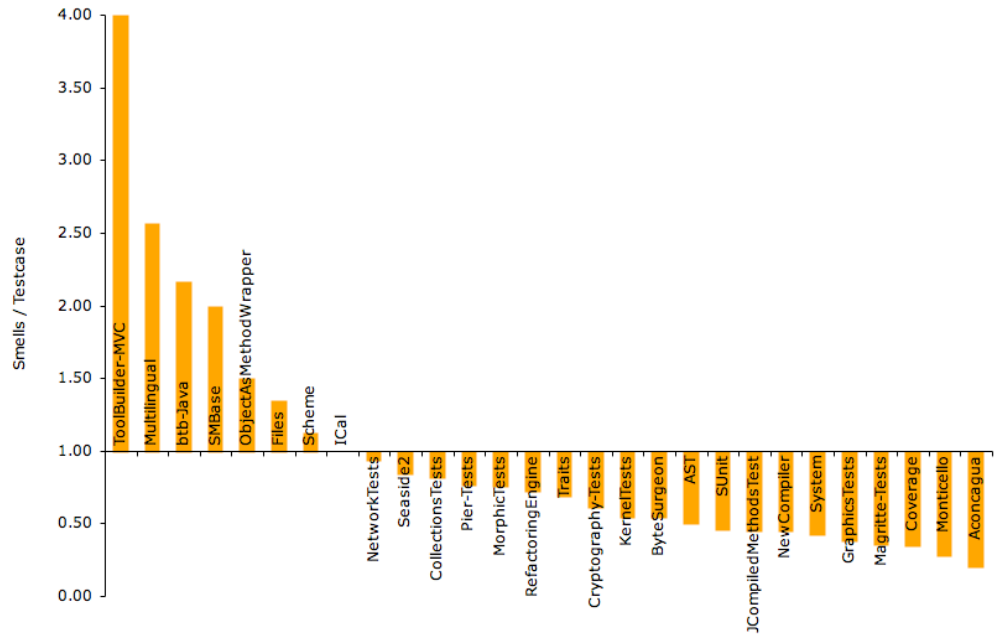

Figure 6: Test Smells per Test and Package

the number of tests. In this case too, tests and Test Smells are approximated using an exponential curves. We notice that the curve approximating Test Smells is monotonically decreasing whereas the one of the number of tests is increasing very strongly.

As a result, we conclude that testing does, at least in long terms, scale extremely well as the quality of tests increases with the experience of writing tests. Qualitatively good test in turn have again a positive effect on the application and so on.

In the following sections we discuss the interesting results of a small selection of well-known packages: Aconcagua, Magritte, Refactoring Engine, and Cryptography.

Aconcagua

Aconcagua is a project about reifying measures as first class objects whereas measure is a number with a unit. Aconcagua encounters about 549 tests which are mostly very good and short in general (Figure 6). The major Test Smells are *Overreferencing* causing a lot of code duplication and *Magic Literals* obfuscating the code a bit. However as Magic Literals have a stronger contextual dependency in tests, they often cause many false positives. We therefore expected many of them as Aconcagua depends a lot of numbers and units. The problem of Overreferencing could have been solved by using more example or factory methods instead of referencing classes

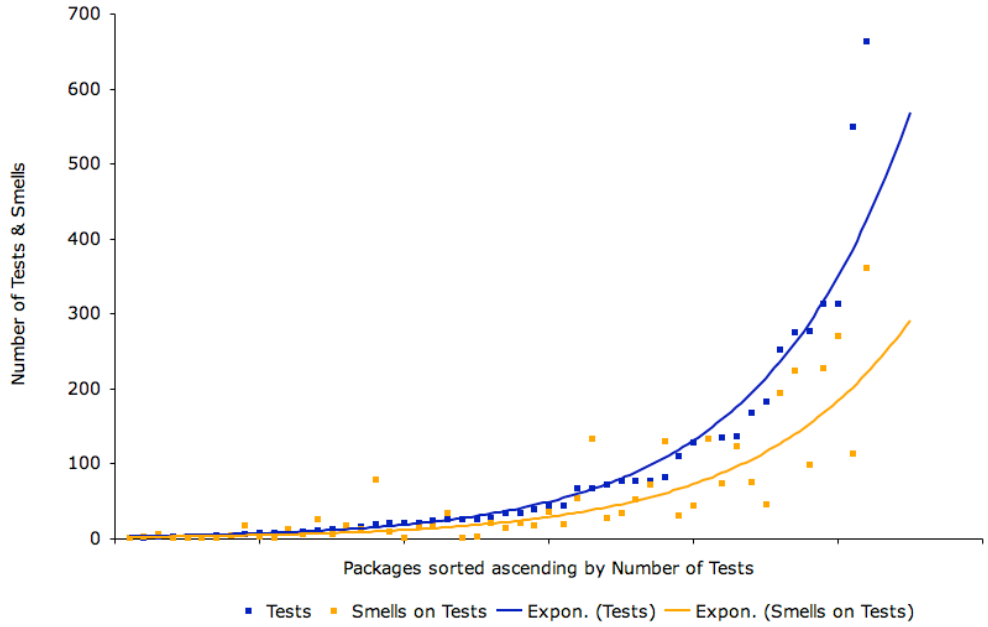

Figure 7: Packages sorted (ascending) by number of tests (blue) on the x-axis. The y-axis shows the number of tests (blue) and smells (orange)

for object creations all over the tests.

The automatic analysis using TestLint mostly agrees to our manual inspection. However the high quality of the tests produce more false positives for Aconcagua then for other packages. The reason for this is probably the fuzziness of the rules as well as the contextual sensitivity of Test Smells.

Magritte

Magritte [28] is a meta-description framework to build user-interfaces, reports, queries and persistency. It defines a large amount of tests (1778), most of them being very good and easy to understand. Using the results of our case study we encountered that only about 2% of all tests have Test Smells, false positives not counted.

Our analysis and the one by TestLint of Magritte concludes that the tests are in general very well designed. However, there are a several tests using *Conditionals* like `self shouldSkipStringTests ifTrue: [ ^self ]` to drop tests. The manual inspection also revealed that one test in the class `MAAutoSelectorAccessorTest` is overriding the default behavior of the underlying unit testing framework, however this is not regarded as a flaw. There are also some cases of *Overreferencing* which actually show some code duplications and missing generic methods.

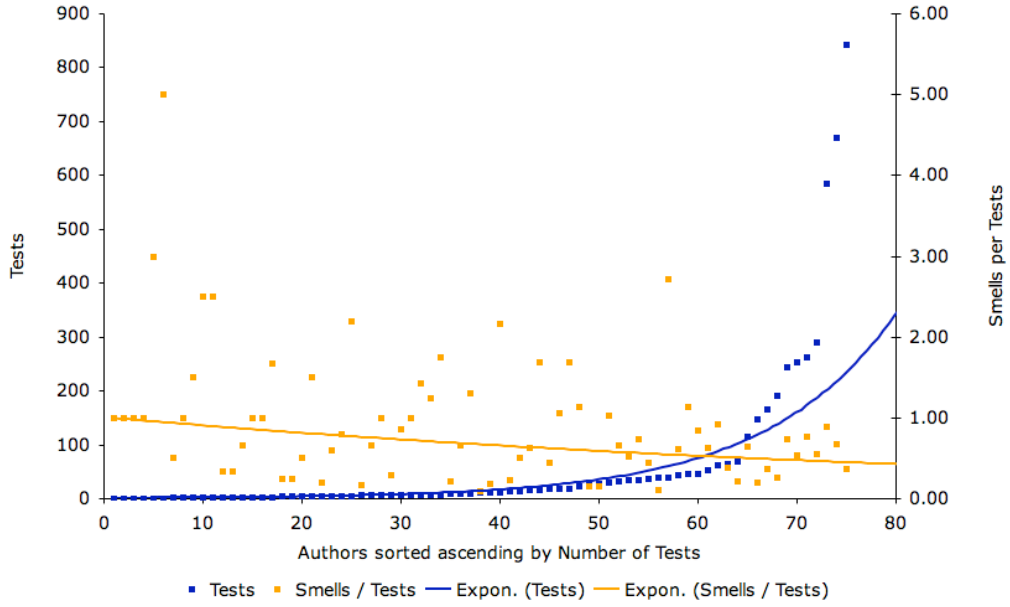

Figure 8: Authors sorted (ascending) by the number of tests (blue) on the x-axis. The primary y-axis shows the number of tests (blue) and the secondary y-axis the number of smells per test (orange)

Refactoring Engine

The Refactoring Engine is a package including Smalllint [9], Refactoring Browsers and other related tools. In general, it lies in the average of all projects. Nevertheless, we have found several problems (Table 3) using TestLint.

Test Smell	**Occurences**
Transcripting Test	2
Overreferncing Test	20
Guarded Tests	10
Long Tests	23
Overcommented	7
Code in Comments	11
Under-the-carpet Assertion	2
Under-the-carpet failing Assertion	1

Table 3: The Test Smells found in the Refactoring Engine.

TestLint further recommends to use explicit fresh fixtures in RefactoringTest instead of the large implicit and shared one. The shared fixtures in Refactoring Engine are not always fully used.

The results gathered by TestLint fully agrees to our previous manual analysis. Furthermore, TestLint found a number of problems which we did not detect by looking at the code (*e.g., Under-the-carpet failing Assertion*).

Cryptography

Cryptography defines a set of well known and frequently used Cryptographic algorithms and protocols. It is a rather large package (247 classes), but contains only very few tests (39). The main problem of Cryptography tests are *Magic Literals*. Although they are expected in such a package they appear far too often. Every single test is heavily "infected", making the tests difficult to understand, especially the purpose of the data. The following Smalltalk code gives an example.

```
CryptoRigndaelCBCTest >> #testRFC3602Case2
    | result |
    ((CBC on: (Rijndael new keySize: 16;
    key: (ByteArray fromHexString: '06A9214036B8A15B512E03D534120006')))
    initialVector: (ByteArray fromHexString: '3DAFBA429D9EB430B422DA802C9FAC41'))
    encryptBlock: (result ˙ 'Single block msg' asByteArray).
    self assert: result hex = 'E353779C1079AEB82708942DBE77181A'
```

A database, examples or factory methods for tests would clean up many of the test and would also document them and make them easier to understand. Furthermore, using TestLint we detected one *Under-the-carpet Assertion* in testSHA256, but not a failing one. TestLint also found out that most tests are badly organized, as it found missing method categories or ones with bad or meaningless names. Furthermore, several test methods are mixed up with non test methods.

Our manual analysis and the result by TestLint are very similar. Especially the bad organization of tests make it difficult to understand the model behind and how it is designed and structured. Furthermore we expected many *Magic Literals*, still we believe the amount could be reduced by a better design of the test data.

6 RELATED WORK

Moonen and Van Deursen [8] as well as Meszaros [22] started researching Test Smells by analyzing and describing them informally and in a much broader domain of software testing. Rompaey, Demeyer at al. [30, 31] formalize them and characterize their significance using software metrics.

There are many other methodologies trying to analyze and assess the quality of tests. Code Coverage [19, 4, 36] was introduced in the early 70s to measure the degree to which the source code of an application is tested. It is one of the first techniques being developed to more systematically and thoroughly test of a software systems.

Moor at al. [23] introduced Mutation Analysis [5, 17, 24] to measure the stability of tests and achieve a qualitatively better measure for the coverage of tests. They apply Mutation Operators on primitives like Boolean, String or Integer within the source to check whether the test still succeeds, in which the test would be a bad one. Yu-Seung at al. [33, 35] altered the original approach of Moor and further extended and improved it. They focus only on method- and class-based Mutation Operators as they argue and conclude that this is a better and more reliable way of of doing Mutation Analysis.

Gaelli at al. [12] propose a new taxonomy of tests [14]. They propose a way of automatically classifying and categorizing tests as well as analyzing their relations using partial ordering of tests [16, 15]. However their approach does not return a formal or concrete measure for the quality of a test respectively the code of a test, either.

7 CONCLUSIONS

In this paper, we have reported on our approach to detect Test Smells. We have started from the informal and rather abstract descriptions available in the literature, and extracted a set of automatic rules based on manually inspecting a set of known tests. Our rules make use of both static and dynamic analysis. We have evaluated our rules against a large body of tests, and we then manually identified the false positives.

The results of our case study show that Test Smells exist and appear quite often, also by people writing many of tests. Therefore it is important for developers to quickly detect Test Smells. Furthermore, we also revealed that developers that write more tests tend also to write better tests. Hence, we conclude that one way to increase testing quality is to write more tests.

The result of our approach is a first list of key TestSmell that we validated in a real and pragmatic setup. We believe that such a list can be the core of future test analysis frameworks.

During the experimentation process, we encountered several problems. First, finding a large enough test set was not a trivial task. Second, Test Smells are defined in the literature on an abstract and informal level. However, at a fine-grained level the Test Smells are interconnected. When we found characteristics of one Test Smell, we might also find evidence for another, and that makes it hard to distinguish between the different encountered smells. Third, many Test Smells depend on contextual information that is hard to formalize. Fourth, there is no clear consensus on what makes a Test Smell, as some smells are regarded differently by different people.

However, our experiments showed promising results as only few false positives have been found. In the future, we intend to continue our research on Test Smells.

We plan to extend the list of smells we can detect. For example, we will target at Test Smells like *Test Duplication* or *Eager Tests*.

We have performed our experiments on Smalltalk code. We expect the rules we built to need adaption for other languages, but we believe that their backbones are language independent. That is why, in the future, we plan to implement and apply our detections to Java.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "NOREX: Network of Reengineering Expertise" (SNF SCOPES Project No. IB7320-110997), and "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008); and of the French ANR (National Research Agency) for the project "COOK: Réarchitecturisation des applications industrielles objets" (JC05 42872).

REFERENCES

[1] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.

[2] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[3] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, 1999.

[4] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[5] B. J. Choi, R. A. DeMillo, E. W. Krauser, R. J. Martin, A. P. Mathur, A. J. Offutt, H. Pan, and E. H. Spafford. The mothra tool set (software testing). In *System Sciences*, volume 2, pages 275–284, January 1989.

[6] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of 15th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 166–178, 2000. ACM Press.

[7] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[8] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95. University of Cagliari, 2001.

[9] Stéphane Ducasse. Refactoring browser et smalllint. *Programmez! Le Magazine du Développement*, 1(46), September 2002.

[10] Stephen Eick, Todd Graves, Alan Karr, J. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[11] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[12] Markus Gaelli. PhD-symposium: Correlating unit tests and methods under test. In *5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2004)*, volume 3092 of *LNCS*, page 317, June 2004.

[13] Markus Gaelli, Orla Greevy, and Oscar Nierstrasz. Composing unit tests. In *Proceedings of SPLiT 2006 (2nd International Workshop on Software Product Line Testing)*, September 2005.

[14] Markus Gaelli, Michele Lanza, and Oscar Nierstrasz. Towards a taxonomy of SUnit tests. In *Proceedings of 13th International Smalltalk Conference (ISC'03)*, September 2005.

[15] Markus Gaelli, Michele Lanza, Oscar Nierstrasz, and Roel Wuyts. Ordering broken unit tests for focused debugging. In *20th International Conference on Software Maintenance (ICSM 2004)*, pages 114–123, 2004.

[16] Markus Gaelli, Oscar Nierstrasz, and Roel Wuyts. Partial ordering tests by coverage sets. Technical Report IAM-03-013, Institut für Informatik, Universität Bern, Switzerland, September 2003. Technical Report.

[17] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.

[18] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *Proceedings of the International Conference on Software Maintenance*, pages 736–743, November 2001.

[19] Brian Marick, John Smith, and Mark Jones. How to misuse code coverage. International Conference and International Conference and Exposition on Testing Computer Software, June 1999.

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

[21] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 350–359, 2004. IEEE Computer Society Press.

[22] Gerarde Meszaros. *XUnit Test Patterns - Refactoring Test Code*. Addison Wesley, June 2007.

[23] I. Moore. Jester – a JUnit test tester. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*. University of Cagliari, 2001.

[24] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Mutation analysis testing for model transformations. In *ECMDA-FA*, volume 4066/2006, pages 376–390, July 2006. IRISA, Campus Universitaire de Beaulieu.

[25] David Lorge Parnas. Software aging. In *Proceedings 16th International Conference on Software Engineering (ICSE '94)*, pages 279–287, 1994.

[26] Allen Parrish, Joel Jones, and Brandon Dixon. Extreme unit testing: Ordering test cases to maximize early testing. In Michele Marchesi, Giancarlo Succi, Don Wells, and Laurie Williams, editors, *Extreme Programming Perspectives*, pages 123–140. Addison-Wesley, 2002.

[27] Stefan Reichhart. Assessing test quality — testlint. Master's thesis, University Bern, April 2007.

[28] Lukas Renggli. Magritte – meta-described web application development. Master's thesis, University of Bern, June 2006.

[29] Don Roberts, John Brant, Ralph E. Johnson, and Bill Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96*, April 1996.

[30] Bart Van Rompaey, Bart Du Bois, and Serge Demeyer. Characterizing the relative significance of a test smell. *ICSM*, 0:391–400, 2006.

[31] Bart Van Rompaey, Bart Du Bois, and Serge Demeyer. Improving test code reviews with metrics: a pilot study. Technical report, Lab On Re-Engineering, University Of Antwerp, 2006.

[32] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proc. 9th Working Conf. Reverse Engineering*, pages 97–107. IEEE Computer Society Press, October 2002.

[33] Jeff Offut, Yu-Seung M, and Yong-Rae Kwon. An experimental mutation system for Java. *ACM SIGSOFT Software Engineering Notes, Workshop on Empirical Research in Software Testing*, 29(5):1–4, September 2004.

[34] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for Java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, pages 352–363, November 2002.

[35] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.

[36] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.