# Assessing Test Quality

## TestLint

**Masterarbeit**
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

**Stefan Reichhart**

April 2007

Leiter der Arbeit

Prof. Dr. Oscar Nierstrasz
Prof. Dr. Stéphane Ducasse

Institut für Informatik und angewandte Mathematik

Further information about this work, the used tools as well as an online version of this document can be found at the following places.

Stefan Reichhart
stefan_reichhart@students.unibe.ch
http://www.squeaksource.com/TestSmells
http://www.squeaksource.com/Coverage

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
http://www.iam.unibe.ch/~scg/

# Abstract

With the success of agile methodologies, Testing has become a common and important activity in the development of software projects. Large and automated test-suites ensure that the system is behaving as expected. Moreover, tests also offer a live documentation for the code and can be used to understand foreign code.

However, as the system evolves, 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 becomes 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.

While testing has grown to be popular and well supported by today's IDEs, methodologies and tools trying to assess the quality of tests are still poorly or not at all integrated into the testing process. Most important, there has been no attempts yet to concretely measure the quality of a test by detecting design flaws of the test code, so called Test Smells.

We contribute to the research of testing methodologies by measuring and assessing the quality of tests. In particular we analyze Test Smells and define a set of criteria to determine test quality. We evaluate our results in a large case-study and present TestLint, an approach to automatically detect Test Smells. We provide a bundle of tools that tightly integrate source-code development, automated testing and quality assessment of tests.

# Acknowledgements

First of all, I thank Prof. Dr. Oscar Nierstrasz, head of the Software Composition Group, for giving me the opportunity to work in his group.

I thank Stéphane Ducasse for giving me the freedom, support and motivation of composing the topic of this thesis. Thanks for all the constructive and encouraging feedbacks on my work.

I thank Markus Gälli for helping me elaborating the final topic of this thesis, giving me many references to related works and creative advices and ideas to spice up the thesis as well as inspiring me for venturesome experimentations.

I thank Marcus Denker and Philippe Marschall for keeping me up-to-date about the latest changes of their work on code instrumentation.

Special thanks to Tudor Gîrba for his mentoring and tenacity of motivating and encouraging me all the time and working out and inspiring me for interesting visualizations on code quality as well as giving me valuable feedbacks.

Thanks also to all the other members of the Software Composition Group and the students working at the SCG student pool. I enjoyed all the discussions, funny intermezzos and lunches I had with you.

My apologies to the Software Composition Group for accidentally having messed up the thoroughly maintained bibliography database. :)

Finally, I want to thank my family for their patience and support they gave me during all these years doing my study.

Stefan Reichhart
April 2007

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

Unit Testing is a common and important activity in today's software projects. Automated tests help the developer to assure code quality and detect possible bugs and flaws in the application code. Furthermore, tests can also be seen as live documentation, and be used to understand foreign code.

Due the continuous propagation of the methodologies *Agile Development* [Cock02a], *Extreme Programming* [Beck00a, Beck01a] or *Test-Driven Development* , many tools supporting automated Unit Testing [Beck98a, Mads02b] have become real and have been integrated into several IDEs, making it easy to create Unit Tests.

However Unit Tests, especially their number and size, neither reveal insights into the amount of functionality or data being tested, nor about how well tested those are or whether the tests themselves are "good" and trustworthy.

Tests might not check each single functionality or requirement of the application code, leaving certain features or code patches untested. Furthermore, tests might not cover all the possible boundary situations, not assuring the stability of the source-code.

Moreover, 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 [Eick01a, Parn94a]. The same quality erosion also happens to test code [Romp06b] – it becomes long, complex and obscure. 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 [Zhu97a, Mari99a] of functionality being tested.
- *Mutation Analysis* gives insights to code stability [Mott06, Howd82a, YuSe02a, Moor01a, YuSe05a].
- *Test Ordering* shows the interconnection of tests [Parr02a, Gael04b, Gael03b, Gael04a].

Although all those methodologies contribute differently in assessing the quality of tests, they all do it at a rather abstract level, and they do not focus on the test or actual test code. Besides, techniques like Code Coverage or Mutation Analysis are missing ease, integration and automation in today's testing tools and IDEs. Test Ordering and visualization of tests is only used in research about test quality, but completely missing in current testing frameworks. Furthermore, the results of all those techniques are mostly static, have no connection to the code, are complex to read and comprehend, and miss the focus for the testing and development process, for example refactorings.

Finally, there has been very little research into identifying and understanding problems and design flaws that influence the maintenance of tests. As the decayed parts of the application code are often referred to as Code Smells [Fowl99a, VanE02a], Test Smells [Deur01a, Romp06a, Romp06b] refers to test code that is difficult to maintain [Mesz07a].

## 1.2 Approach

Our solution to the previously described problems is to enhance and substantiate current methodologies trying to assess the quality of tests. We introduce new features into testing tools that have not yet being implemented but which are necessary or show evidence to be useful for an efficient Test-Driven Development, refactoring and reengineering process.

In our research activities we primarily focus on the analysis and automatic detection of Test Smells. We believe that this methodology will help developers to:

- *detect problems early*, fast and without much effort or spending time in harvesting test code or debugging.

- write *simpler and better understandable tests* that more easily and concisely document the features and requirements of the application code as well as the aim of the test
- *design* tests to make them robust and stable to changes
- *ease refactorings* and get a better focus on what parts of the tests have to be refactored, why and how
- learn to *write qualitatively good tests* and stick to standard coding and testing conventions

We also believe that this methodology will greatly improve the quality of tests in general and maybe also of the application code.

We present TestLint [Reic07a], an approach to detect design flaws in tests automatically. We conduct an empirical study of a larger set of tests, gather an extensive list of all kinds of Test Smells through manual inspection and analyze them in detail. We cluster the problems, identify commonalities and differences and distill the results in automatic queries to precisely detect design flaws in test-code. We conduct a large case study to show the need and significance of detecting Test Smells and conclude that writing tests can really improve the quality of tests.

We implement the methodology of Test Smells in a tool called TestLint. By integrating it into the development and testing environment we take the first steps toward Quality-Driven Development. That is supposed to seamlessly and tightly integrate testing and quality-testing methodologies into the evolution of application and test code. We believe that this will not only enhance the quality of tests but also the documentation of the code and the software in general.

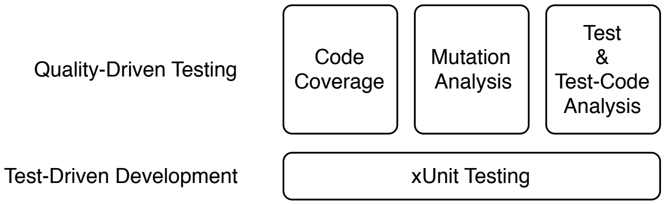


Figure 1.1: The 3 best known quality assessing methodologies on top of Unit Testing, forming the methodology of Quality-Driven Development

## 1.3 Outline

- Chapter 2 gives an introduction about the different kinds of testing methodologies. In particular, we make an introduction to the recent research of Test Smells, briefly explaining and describing them in general. Furthermore, we present well known and highly used tools that go beyond simple Unit Testing and discuss the advantages, problems and limitations of current implementations and their methodologies.
- Chapter 3 presents TestLint, our approach and contribution to the analysis of detecting design flaws in tests. We conduct an extensive empirical study and analyze Test Smells at an abstract and lower code level. We combine our results with the existing research results, enhance them and draw conclusions.
- Chapter 4 presents a selection of the most commonly found Test Smells based on our empirical study. We discuss, explain and categorize them, and formalize concrete rules to detect them automatically. Furthermore we give hints to fix the smells in the test-code.
- In Chapter 5 we discuss a large case study performed with TestLint, conclude the need for the detection of Test Smells and give example results on a selection of projects analysed with TestLint.
- In Chapter 6 we conclude our results and experience with TestLint and give further insight into the future work.

In the Appendix we detail the implementation of TestLint and give a full list of Test Smells we analysed. Furthermore we present Christo, an approach to enhance standard Code Coverage to gather insights about the quality of tests and reuse the dynamic data to make further conclusions about the application and test code.

## Chapter 2

# State of the Art in Testing

Testing application code, in particular automated *Unit* and *Regression Testing* is an integral part of today's development processes as proposed by *Extreme Programming* [Beck00a, Marc02a, Marc03a]. This methodology proved to greatly increase the design and quality of the application code. In the following sections we present various known testing methodologies and techniques and discuss a selection of current tools supporting the testing activity.

## 2.1 Code Coverage: Test Quantity

*Code Coverage* [Corn96a, Chil94a, Corn06a, Zhu97a], first based on a static analysis later also on dynamic analysis, is a common and very well known technique 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 a software systems.

After the first official publication of the rough idea by Miller and Maloney [Mill63a] at ACM[1] in the early 70s, Code Coverage obtained much attention in the following years in which the technique was more thoroughly researched and elaborated. Code Coverage was implemented in many software testing tools in all major and most other well known languages, for example C. Large software projects use Code Coverage to measure test coverage of the entire application code or the kernel functionality only. Safety critical projects, for example in avionics and astronautics, even require such tools and demand at least full coverage.

[1] ACM (Association for Computing Machinery) is the world's first scientific and educational computing society

Although Code Coverage proves to be an useful technique and tool for testing, it can only serve as an indicator of how much and which parts of the application logic are being covered by a test. It cannot tell the developer whether the covered code works appropriately nor can it objectively assess the quality of the code or the test itself [Corn99a, Mari99a].

Today, software testing distinguishes between 5 major aspects of quantity-based Code Coverage [Corn96a]. We notice that other or more fine-grained differentiations could also be done:

- *Method Coverage* is the simplest form of coverage only checking whether a method has been called during the runtime of a test
- *Statement Coverage* focuses only on whether a line of sourcecode, *e.g.,* a statement is executed or not
- *Condition Coverage* is analyzing conditional branches within source code, checking that each branch is executed
- *Path Coverage* is checking a possible route through the code or a given part of it
- *Entry Coverage* deals with any possible call and return of a method during the current execution

Depending on the coverage strategy chosen the results might vary in precision and density. For example Condition Coverage implies Statement Coverage, but not the other way around. Therefore Condition Coverage induces more knowledge about the system than Statement Coverage.

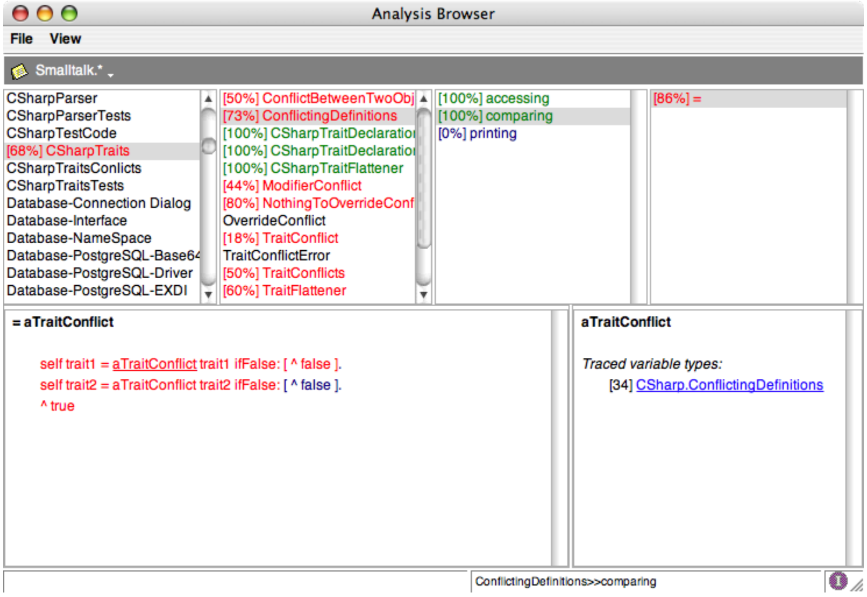
Most Code Coverage tools focus on Method Coverage, Statement Coverage or Condition Coverage as they are fairly simple to check. Full Path Coverage is mostly impractical or even impossible, especially for medium and large projects, as its complexity and the data collected is growing exponentially with every branching or enumerating code structure. Moreover the results gathered by Path Coverage are mostly complex and hard to understand.

The following sections will discuss each of a Code Coverage tool being developed and used in VisualWorks and Java projects.

### Zork Code Coverage

Zork is the only known project in Smalltalk that deals with Code Coverage. It is implemented in VisualWorks and is based on John Brant's Method Wrappers [Bran98a], providing a kind of Statement Coverage and Condition Coverage.

Although wrapped code runs approximately 6 times slower, Zork is a fast and sophisticated tool, providing a lot of very fine-grained information about coverage. It provides an intuitive system-browser-like browser, called the Analysis Browser (Figure 2.1), which offers the opportunity to browse that information.



```
CSharpParser
CSharpParserTests
CSharpTestCode
[68%] CSharpTraits
CSharpTraitsConlicts
CSharpTraitsTests
Database-Connection Dialog
Database-Interface
Database-NameSpace
Database-PostgreSQL-Base64
Database-PostgreSQL-Driver
Database-PostgreSQL-EXDI
```

```
[50%] ConflictBetweenTwoObj
[73%] ConflictingDefinitions
[100%] CSharpTraitDeclaratio
[100%] CSharpTraitDeclaratio
[100%] CSharpTraitFlattener
[44%] ModifierConflict
[80%] NothingToOverrideConf
OverrideConflict
[18%] TraitConflict
TraitConflictError
[50%] TraitConflicts
[60%] TraitFlattener
```

```
[100%] accessing
[100%] comparing
[0%] printing
```

```
[86%] =
```

```
= aTraitConflict

    self trait1 = aTraitConflict trait1 ifFalse: [ ^ false ].
    self trait2 = aTraitConflict trait2 ifFalse: [ ^ false ].
    ^ true
```

```
aTraitConflict

Traced variable types:
    [34] CSharp.ConflictingDefinitions
```

```
ConflictingDefinitions>>comparing
```

Figure 2.1: A standard view of the Zork Analysis Browser

However, Zork lacks automation, ease, usability and safety. A developer needs to go through many manual and time-consuming steps to set up and obtain coverage. For example, the developer must first select each class or method separately from a flat list of all classes and methods within the environment, and install the wrappers by clicking through multiple dialogs. To browse the results, one must explicitly open the Analysis Browser and search for the processed items manually.

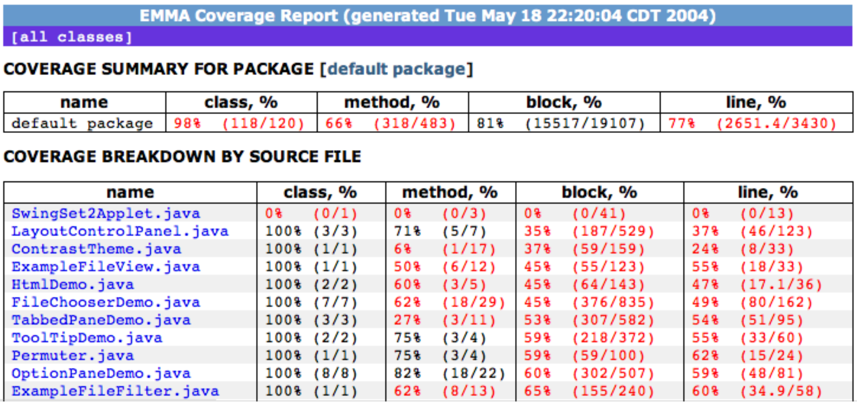
Furthermore, the developer needs to know which methods are savfe to wrap. Installing a wrapper on an unsafe element might corrupt the image or even cause it to crash. Finally, the wrappers need to be manually uninstalled before continuing to work in the environment. When running the analysis again, the developer has to go through all steps again.

Although Zork provides an easy to use browser, coloring well-covered nodes in green, others in red, and annotating the code with additional type information, it completely misses the connection between covered sources and executed tests. It is therefore not possible to find out which method was actually covered by which test, or the other way round. However this information is absolutely crucial to extend the Test-Suite with the right tests to obtain a higher and better coverage.

We further notice that the coloring and accumulation of coverage values are wrong. The selected method = in Figure 2.1 declares a coverage of 86%. However all nodes, except the last block, are colored in red, assuming rather 0%. On the other hand, the method-category containing this method (and only this method) shows 100%.

### Emma Code Coverage

Emma is one of many Code Coverage tools for Java. However in contrast to other Code Coverage tools, Emma is open-source and therefore available for free. It provides many features like offline and online code instrumentation, partial coverage, full integration into build systems like Ant or Maven, HTML- and XML-report generation and many more. Due its rich features and simplicity compared to other tools, it gained a lot of interest and is therefore one of the best known and most often used Code Coverage tools in the Java community.



EMMA Coverage Report (generated Tue May 18 22:20:04 CDT 2004)

[all classes]

COVERAGE SUMMARY FOR PACKAGE [default package]

| name | class, % | method, % | block, % | line, % |
|---|---|---|---|---|
| default package | 98% (118/120) | 66% (318/483) | 81% (15517/19107) | 77% (2651.4/3430) |

COVERAGE BREAKDOWN BY SOURCE FILE

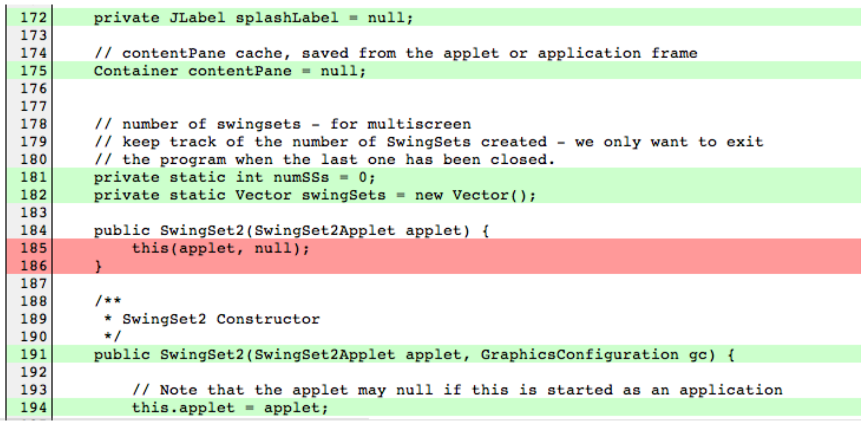
| name | class, % | method, % | block, % | line, % |
|---|---|---|---|---|
| SwingSet2Applet.java | 0% (0/1) | 0% (0/3) | 0% (0/41) | 0% (0/13) |
| LayoutControlPanel.java | 100% (3/3) | 71% (5/7) | 35% (187/529) | 37% (46/123) |
| ContrastTheme.java | 100% (1/1) | 6% (1/17) | 37% (59/159) | 24% (8/33) |
| ExampleFileView.java | 100% (1/1) | 50% (6/12) | 45% (55/123) | 55% (18/33) |
| HtmlDemo.java | 100% (2/2) | 60% (3/5) | 45% (64/143) | 47% (17.1/36) |
| FileChooserDemo.java | 100% (7/7) | 62% (18/29) | 45% (376/835) | 49% (80/162) |
| TabbedPaneDemo.java | 100% (3/3) | 27% (3/11) | 53% (307/582) | 54% (51/95) |
| ToolTipDemo.java | 100% (2/2) | 75% (3/4) | 59% (218/372) | 55% (33/60) |
| Permuter.java | 100% (1/1) | 75% (3/4) | 59% (59/100) | 62% (15/24) |
| OptionPaneDemo.java | 100% (8/8) | 82% (18/22) | 60% (302/507) | 59% (48/81) |
| ExampleFileFilter.java | 100% (1/1) | 62% (8/13) | 65% (155/240) | 60% (34.9/58) |

Figure 2.2: A standard Emma HTML-report showing the summary of a processed package

The instrumentation is done either on single class files, packages or entire JAR-libraries using direct Byte-Code transformation, resulting in a very low runtime overhead.

Although Emma provides rich functionality and is also highly customizable, it lacks – like many other Code Coverage tools – ease and usability. Emma is not yet well integrated into today's IDE's, causing the developer to spend time writing scripts or plugins to simplify and automate the gathering of coverage information. Switching between the IDE and web browser is inevitable as only the features of Emma are accessible through the IDE; the results on the other hand have to be inspected elsewhere.

Furthermore, Emma-reports – like any other coverage reports – generally contain a lot of information and are rather complex. This requires the developer to spend a lot of time in understanding and harvesting valuable information and getting used to the amount of data displayed. Figure 2.2 shows the overview, called the package summary of a usual Emma-report.



```
private JLabel splashLabel = null;

// contentPane cache, saved from the applet or application frame
Container contentPane = null;


// number of swingsets - for multiscreen
// keep track of the number of SwingSets created - we only want to exit
// the program when the last one has been closed.
private static int numSSs = 0;
private static Vector swingSets = new Vector();

public SwingSet2(SwingSet2Applet applet) {
    this(applet, null);
}

/**
 * SwingSet2 Constructor
 */
public SwingSet2(SwingSet2Applet applet, GraphicsConfiguration gc) {

    // Note that the applet may null if this is started as an application
    this.applet = applet;
```

Figure 2.3: Emma HTML-report, showing coverage details

Hyperlinked reports and visualized covered and uncovered lines of code help browsing and identifying the uncovered pieces of the code (Figure 2.3). However the focus on the interesting and important parts is missing and also Emma misses the link between sources and tests.

## 2.2 Mutation Analysis: Test Stability

Test stability can be measured using a variation of Code Coverage which is based on fault tolerance and boundary checking rather than quantity. This kind of coverage is also known as *Mutation Testing* or *Mutation Analysis* [Mott06, Howd82a, Choi89a, Baud06a] and was originally introduced by Moor at al. [Moor01a]. The original approach of Mutation Analysis was recently altered and further extended and improved by Yu-Seung at al. [YuSe04a, YuSe05a]. We discuss and give examples for both approaches.

The basic idea of Mutation Testing is to systematically modify pieces of the program code by applying a *Mutation Operator* and to run the tests on the mutated sources. If the tests pass then we can conclude either of two possibilities.

- A test is missing because there is a mutation *a.k.a.* "mutant" that lets the test pass, for example an unchecked boundary.
- There is some redundant code in the program.

A set of tests whose mutants all fail – or get "killed" –, i.e., there is no succeeding test after a mutation, is said to be effective or having a good fault coverage or high stability.

The major difference to quantity based coverage is that Mutation Testing focuses more on the test quality and whether the tests behaviorally cover all the specified requirements in the tests. Furthermore it cannot be fooled by tests simply executing code, but not doing any assertions as could be possible in Code Coverage.

Mutation Testing differentiates between weak [Howd82a] and strong Mutation Coverage [Moor01a], whereas Strong Mutation implies Weak Mutation but not the other way round.

- *Weak Mutation Coverage* assumes that the application code is correctly implemented according to the requirements. Therefore it can only tell you whether there is a test missing for a certain mutant or not.
- *Strong Mutation Coverage* rather enforces the correct specification of a test and mutation, trying to figure out whether the tests really check what the code suggests.

The following example demonstrates the fine difference between the weak and strong aspect. Assume the following piece of application code containing the simple fault <= instead of <:

```
(A <= B) ifTrue: [ ... ]
```

If the tests runs and A<=B is always executed as A<B, Weak Mutation Coverage only indicates a missing test for the condition A=B. However Strong Mutation Coverage would lead to the mutant A=B not being killed, causing the test to pass, and therefore lowering the coverage value and the quality of the test.

Although Mutation Testing adds valuable information to Code Coverage, the number of all possible mutations and their combinations cause a very high computational effort, rising exponentially with the amount of code and mutants as a Test-Suite has to run for every single mutation seperately. This causes Mutation Coverage to run extremely slow, even with a fast Test-Suite on a small codebase. As a consequence of this, it is not very often used or only to check simple boundaries, for example array indexes or boolean conditions.

Another drawback of Mutation Testing is the sensitivity of the Mutation Operators towards the data and subject of application – not every mutation makes sense on all kind of data and application. As an example, mutating a string which contains the name of a person or a number representing the current year will very likely lead to mutants not being eliminated by the tests, supposing a lower coverage. Because of that and despite the computational effort, Mutation Analysis might not return any useful result, causing many false positives, in the worst case even unnecessarily prolonging the testing-process.

### Jester

Jester [Moor01a] is probably the best known Strong Mutation Coverage tool. It is open source, available for Java and ported to other languages like Python or C#.

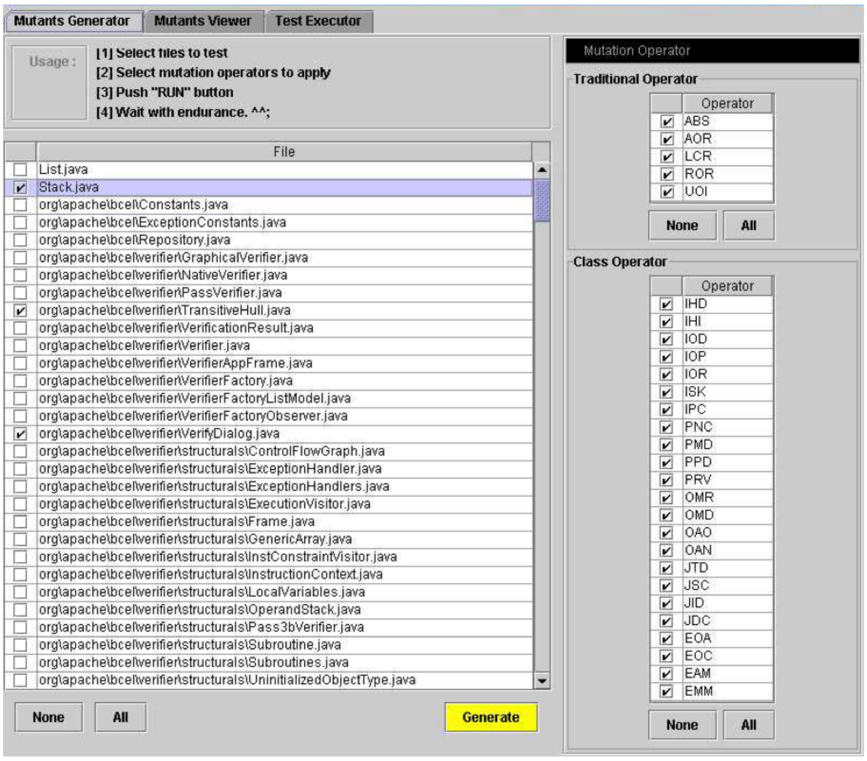
Although Jester exists for several years it has not yet been integrated into any common IDE, for example Eclipse. Therefore using Jester is quite labor intensive as someone has to specify all the classes and libraries by hand on the command line. It is necessary to write an Ant script transforming the necessary paths automatically.

Jester runs extremely slowly – as expected for a Mutation Testing tool. For each mutant it has to duplicate and recompile the sources before it can actually run the tests. Due to this performance limitation it is almost impossible to analyze more than one class at a time. Furthermore, the reports generated on the console or as XML are hard to decipher. There is no support to browse between sources and tests. In general, developers have a hard time harvesting the results.

As the methodology of Mutation Analysis is not foolproof due the partial sensitivity to the context, it is very likely that mutants get created that are either obvious to fail or can never fail. Therefore, Jester reports likely contain many false positives, depending on the Mutation Operators used.

### $\mu$Java

$\mu$Java [YuSe05a, YuSe04a] is another mutation system for Java that already comes with a complete and reasonably complicated user interface. However it has only little in common with tools like Jester. $\mu$Java is capable of strong as well as weak Mutation Analysis making it more flexible to testing needs. Furthermore $\mu$Java introduces class-level mutation [YuSe05a, YuSe02a].



[1] Select files to test
[2] Select mutation operators to apply
[3] Push "RUN" button
[4] Wait with endurance. ^^;

| | File |
|---|---|
| □ | List.java |
| ☑ | Stack.java |
| □ | org\apache\bcel\Constants.java |
| □ | org\apache\bcel\ExceptionConstants.java |
| □ | org\apache\bcel\Repository.java |
| □ | org\apache\bcel\verifier\GraphicalVerifier.java |
| □ | org\apache\bcel\verifier\NativeVerifier.java |
| □ | org\apache\bcel\verifier\PassVerifier.java |
| ☑ | org\apache\bcel\verifier\TransitiveHull.java |
| □ | org\apache\bcel\verifier\VerificationResult.java |
| □ | org\apache\bcel\verifier\Verifier.java |
| □ | org\apache\bcel\verifier\VerifierAppFrame.java |
| □ | org\apache\bcel\verifier\VerifierFactory.java |
| □ | org\apache\bcel\verifier\VerifierFactoryListModel.java |
| □ | org\apache\bcel\verifier\VerifierFactoryObserver.java |
| ☑ | org\apache\bcel\verifier\VerifyDialog.java |
| □ | org\apache\bcel\verifier\structurals\ControlFlowGraph.java |
| □ | org\apache\bcel\verifier\structurals\ExceptionHandler.java |
| □ | org\apache\bcel\verifier\structurals\ExceptionHandlers.java |
| □ | org\apache\bcel\verifier\structurals\ExecutionVisitor.java |
| □ | org\apache\bcel\verifier\structurals\Frame.java |
| □ | org\apache\bcel\verifier\structurals\GenericArray.java |
| □ | org\apache\bcel\verifier\structurals\InstConstraintVisitor.java |
| □ | org\apache\bcel\verifier\structurals\InstructionContext.java |
| □ | org\apache\bcel\verifier\structurals\LocalVariables.java |
| □ | org\apache\bcel\verifier\structurals\OperandStack.java |
| □ | org\apache\bcel\verifier\structurals\Pass3bVerifier.java |
| □ | org\apache\bcel\verifier\structurals\Subroutine.java |
| □ | org\apache\bcel\verifier\structurals\Subroutines.java |
| □ | org\apache\bcel\verifier\structurals\UninitializedObjectType.java |

Traditional Operator

| | Operator |
|---|---|
| ☑ | ABS |
| ☑ | AOR |
| ☑ | LCR |
| ☑ | ROR |
| ☑ | UOI |

Class Operator

| | Operator |
|---|---|
| ☑ | IHD |
| ☑ | IHI |
| ☑ | IOD |
| ☑ | IOP |
| ☑ | IOR |
| ☑ | ISK |
| ☑ | IPC |
| ☑ | PNC |
| ☑ | PMD |
| ☑ | PPD |
| ☑ | PRV |
| ☑ | OMR |
| ☑ | OMD |
| ☑ | OAO |
| ☑ | OAN |
| ☑ | JTD |
| ☑ | JSC |
| ☑ | JID |
| ☑ | JDC |
| ☑ | EOA |
| ☑ | EOC |
| ☑ | EAM |
| ☑ | EMM |

Figure 2.4: $\mu$Java user interface to select and configure the Mutation Operators

Class-level mutation applies only to object-oriented languages that provide features like encapsulation, inheritance, polymorphism and others. $\mu$Java is organizing the mutators in those categories giving them a description and a 3 letter shortcut. We list a few examples of $\mu$Java 's class-level mutators:

- *A*MC (access modifier change): the operator changes the access modifier (public, private, protected) of a method or constructor. The purpose of the AMC operator is to guide testers to generate test cases that ensure that accessibility is correct.

- *I*SI (super keyword insertion): the operator inserts a super keyword to reference the overridden method or variable from the super class. The ISI operator is designed to ensure that hiding/hidden variables and overriding/overridden methods are used appropriately.
- *J*ID (Member variable initialization deletion): the operator removes the initialization values of member variables. This is designed to ensure correct initializations of instance variables

Contrary to Jester, $\mu$Java only mutates operators but not literals at the method-level [YuSe05a]. This means, it does not mutate values of integers, strings or booleans. Therefore $\mu$Java is less likely to be caught by the contextual sensitivity which might produce many false positives, making Mutation Testing a very expensive way of testing.

However it has similar but not as distinctive time-consuming computations as Jester and requires a lot of knowledge to set up an appropriate mutation set. As $\mu$Java is not integrated into any IDE, nor provides any set-up scripts, the developer need a considerable amount of time to set this tool up and run it.

## 2.3 Test Smells: Test-code Quality

The systematic analysis of Test Code is a rather new topic in measuring the quality of a test. It is known under the keyword *Test Smells* [Deur01a, Mesz07a, Romp06a, Romp06b] and focuses on the detection of bad code fragments and design flaws in tests.

The idea of Test Smells is directly derived from Lint [Duca02v], a static code analyzer for application code, and the research in reengineering. Whereas Lint focuses on the low-level analysis of source-code, for example by scanning for suspicious code elements, reengineering [Deme02a, Duca00a, Lanz99a] tries to capture the problems at a more abstract level by detecting general design flaws in the model design and evolution.

As an overview we give three simple introductory examples of Test Smells as they can be found in the work of Meszaros [Mesz07a].

- An *Eager Test* is a test that verifies too much different 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 obscures the purpose of the tests. Furthermore setup and teardown require a large amount of time slowing down the execution of the tests.

Test Smells have been summarized, categorized and described informally by Deursen at al. [Deur01a] and Meszaros [Mesz07a]. Meszaros further decomposed and subdivided them, explaining the reasons for their appearance as well as their consequences in detail. Metrics and heuristics [Mari04a, Mari01a], rather abstract and mathematical approaches to generally detect design flaws are also being adapted to tests to make them able to detect suspicious code and design in tests, and to gain insights into their significance [Romp06a, Romp06b].

Although the current research proves the significance and usability of detecting design flaws and bugs in tests, all the approaches are still at a very high level of abstraction or provide an informal description only. They cannot detect concrete smells at the lower code level, nor their reasons due to their abstractness.

As there are no tools available to statically or dynamically analyze tests and test code we give a representative example of Lint, a static source-code analyzer and briefly discuss its applicability to Test Smells.

### Smalllint

Smalllint is the Smalltalk version of Lint, doing a static code analysis. It implements a dedicated and special query language on the parse tree to find design flaws or possible bugs.

Although it is not designed or meant to be applied to tests, some of its rules could actually be used to analyze test code, however with a reduced information value like the following examples demonstrate.

- *Long Methods* apply also to tests, however tests are normally much longer than non-testing methods causing too many tests to appear when using this rule with the same parameters.

- *size=0 instead of isEmpty* and similar code structures might reveal some intention for source code. When used in tests it is rather a sign of an encapsulation being broken in of the application code or obfuscating the test code with symbols and literals.

- *Method implemented but not sent* completely fails for tests as test methods are not referenced – unless tests are calling each other. However *Chained Tests* represent rather a bad or at least unusual design for unit tests.

Whereas Smalllint is great in detecting Code Smells, it is not suitable for the detection of Test Smells. Although it is basically able to detect certain design flaws in test-code, the results are mostly too general or fail due the missing dedication of the rules towards tests. The last example above even shows that some rules would need to be inverted to be useful fo tests.

We notice that other tools offering similar functionality to Smalllint, for example *FindBugs* or *Lint4J* in Java, have the same limitations.

## 2.4 Other testing methodologies

Gaelli et al. propose a new taxonomy of tests [Gael03a, Wamp06a]. They analyze the different types of tests, automatically categorize and decompose them into smaller units, called commands [Gael04c]. They use partial ordering [Gael03b] to debug and comprehend unit tests [Gael04b] and application code and present the *Eg* meta-model [Gael06b, Wamp06a] to link unit tests to their method under test and compose unit tests to form higher-level test scenarios. The basic concepts of the *Eg*-meta-model has been implemented in a prototypical testing-browser in VisualWorks.

*Delta Debugging* , proposed by Zeller at al. [Zell01a, Zell05a], is an approach to *Automated Debugging*. They analyze the cause-effect chain of a failure to automatically find the failure-inducing statements [Zell02a, Zell02b], supporting the developer in the development and debugging process. Due to the complexity of Delta Debugging it is only available for a small number of languages so far. As an example there is a Eclipse-plugin for Java.

The members of the *Dynamix Group* at the Software Composition Group are developing multiple approaches to dynamic analysis and representing it using various meta-models. Denker [Denk06a, Denk06b] introduced dynamic bytecode transformations using higher abstraction levels. Marschall [Mars06a] is further abstracting this approach by introducing reflection into the sub-method level. On top of this framework, Lienhard [Lien06a] has built ObjectFlow to capture the life cycle of objects during execution.

## 2.5 Conclusion

Most techniques trying to assess additional information about quantity and quality of tests have several drawbacks. Neither Code Coverage nor Mutation Testing are able to reveal concrete design flaws in the test-code. Partial

Ordering of tests can only help one to understand the interconnection of the tests. Similar for Flow Analysis and Delta Debugging.

Furthermore, tools representing those methodologies are strictly separated from each other and take a considerable amount of time to set up. They are complex to use because either the methodology is complex or the testing interface is badly designed. Moreover, they completely miss the automation which is essential for an efficient testing process. Also, the results are mostly static, for example in the form of a browsable, maybe hyperlinked HTML- or XML-report, and are lacking connection between sources and tests. Such reports are complex to understand and miss the essential focus as they don't point the developer to the really relevant parts of the code and tests; in particular there is no support for refactoring.

Other techniques like Delta Debugging or Flow Analysis are still heavily in research and barely implemented in tools usable for real applications. As an example, there are currently no tools available to either dynamically or statically analyze test code for possible design flaws or bugs within tests – despite the existing advantages taken by Lint.

The following chapters present an approach to assess Test Quality. We contribute to the analysis of Test Smells, define first automated detection mechanisms and discuss the results in a large case study.

Although the topic of this thesis is about assessing code quality, we did not focus on Mutation Analysis. That's basically because of the complexity of this methodology and its dependency towards the language and its features. In particular there has been no attempt of introducing Mutation Testing to dynamically typed languages like Smalltalk. This research will remain to the future work on assessing test quality.

In the Documentation of the Appendix we give insights into Christo, an approach to enhance the Code Coverage methodology to gather and extract quality-based information from the tests and visualize the results in a more effective and comprehensive way.

# Chapter 3

# TestLint: Measuring Test Quality

During the evolution of a software system [Parn94a], code starts to erode. It becomes complex, fragile, susceptive to bugs and hard to maintain. The code and the design of the system starts to "smell" [Fowl99a, VanE02a]. The same happens to the tests and their code.

Test Smells [Deur01a] describe tests that are too long, complex, include unnecessary redundancy, exposing or breaking encapsulation of the application code, run unnecessarily slowly, or make inappropriate assumptions on external resources. Furthermore they might frequently change their results, leading to fragile and untrustworthy tests.

There are many reasons for Test Smells [Mesz07a]. One factor that makes the test code brittle to changes in the application code are frequent changes in the application code. Another is 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 for testing.

The consequence of such tests is that they are hard to understand, difficult to maintain, and badly document the application, its features and requirements. Furthermore, they typically become unstable, or even become unused or deprecated.

As only little research has been conducted on the detection of Test Smells so far, mostly at a high and rather abstract level or describing them informally, there is no approach or knowledge for the automatic detection of Test Smells at the code level.

In the following sections we describe TestLint, our contribution to the methodology of Test Smells. We analyze Test Smells in a fine-grained fashion and combine and synthesize our results with existing knowledge. We formalize a model and rules to automatically detect Test Smells and explain a selection of those rules in detail.

## 3.1 Analysis and Synthesis of Test Smells

The basis of our research is a case study consisting of 4834 test-methods and 742 test-classes taken from the Squeak[1] open-source community. Our study was conducted in three steps:

1. The first step (Section 3.1.1) 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 tests, 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 (Section 3.1.2) we clustered the problems to identify commonalities and differences, and we distilled the lessons in automatic queries (Chapter 4) that we implemented in a tool called TestLint.
3. In the third step we have applied our queries on *all the tests* in our case study (Chapter 5) and manually inspected the detected Test Smells to identify false positives.

### 3.1.1 Analysis

Based on our preparations we started to systematically analyze the test samples by manually inspecting each test. We set up a list of Test Smells including their causes and effects already known by other research projects [Deur01a, Romp06a, Romp06b] and literature [Mesz07a].

During our analysis we heavily extended that list with more details and added many new abstract design and very fine-grained concrete code problems found in the tests. Table 3.1 gives a small overviewof the 10 most common and re-appearing problems. A full list of Test Smells including descriptions for each is given in the Appendix.

First, we determine that some of the smells we captured in our analysis are *partially subjective*. For example not everyone agrees to regard *Chaotic*

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

| |
|---|
| Magic Literals |
| Bad test data, location or reuse |
| Code Duplication, Multiplication |
| Conditional Logics (ifTrue:, ifFalse:) |
| Missing Generics, Building Blocks |
| Complex and obscure tests |
| Too many and too large comments |
| Valid code (or complete test-methods) put into comments |
| Meaningless class and method names |
| Chaotic or no organization |

Table 3.1: The 10 most common problems in tests and test-code

*Organization* or *Large Comments* as smells because they're not directly related to the test-code or the runtime of a test. However we regard them as smells because they obfuscate and therefore decrease the quality of documentation or even prevent the comprehension of it. Besides, we discovered that such smells often appear in conjunction with other smells, sometimes even causing them.

In a next step, by taking into account knowledge from the literature [Mesz07a], we analyzed all Test Smells within their context to gather further insights into the following problems and questions:

- *Cause.* What is causing a Test Smell, and why and how ?
- *Consequences.* How do Test Smells manifest themselves in the code ? What effects or consequences do they have, for example on the code, tests and other Test Smells?

We notice that some Test Smells are extremely *context sensitive.* For example, a parser or compiler framework handles a lot of primitive data like strings and numbers. Therefore *Magic Literals* in application and test code appearing as Test Smells are inevitable. However respecting the context and the matter of such a project we cannot treat *Magic Literals* as a design flaw because there is no other way of doing such a project. Therefore it is important to know the scope of a project when analyzing its tests for Test Smells.

Furthermore identifying the concrete origin and cause of one particular Test Smell or multiple smells is difficult and sometimes impossible. That's because of the contextual sensitivity of the tests and smells. Therefore too many different and possible reasons might apply. We give two examples. The first one is describing causes at a very abstract level whereas the second one describes a rather concrete Test Smell:

- Tests might be badly designed, having lots of Test Smells because the project is lacking time for testing. Maybe testing was not part of the development process in the first place, and done at the end of the project, leaving only a small amount of time for testing. The environment or language the project is made with provides no testing platform fitting the needs of the project. ...
- A test-method has too many and large comments because the matter of the project is complex. Maybe the code-under-test or its interface is badly designed and became complex. Another reason for a design flaw could be because the test was already badly designed, lacking generics and no dedicated place for data. ...

We conclude for the cause of Test Smells that the contextual sensitivity influences the analysis and very likely has a negative impact on the automatic detection of Test Smells. Therefore the automation will be fuzzy and heavily depend on human arbitration. Furthermore, the same concrete detection formalisms might work very well for some tests, but might completely fail for others.

When analyzing tests we immediately become aware of the amount and *diversity of consequences* Test Smells can have. We notice that the type of Test Smell and the number and combination of smells found in a test can lead to more or less severe consequences. For example, we regard a long test with many conditional branches to be worse than an much longer test with many and large comments. Others might not agree to that or even regard the latter as not being a smell. Besides, the consequences of Test Smells can be regarded as subjective as not everyone agrees to them. For instance not everyone regards a slow test as a serious Test Smell. Nevertheless we noticed in our analysis that slow tests are often put into comments, guarded (Section 4.1.9) and therefore likely to be ignored.

In the following list, we present a selection of regularly reoccurring consequences we've found in our analysis:

- Test becomes *unsuitable* for documenting the model as it is more complex than the model itself.
- Lack of test *comprehension*. It is unclear what the test is doing, which requirements, features or date it actually covers or what's the purpose of the test.
- The tests are or seem to be *unmaintained* and *unused*. The code is old or failing as the model doesn't implement the specified methods any more. Maybe the test-data doesn't fit either.

- Tests run extremely *slowly*. Slow tests often fail as they're less often executed. Many slow tests are put into comments, hiding the test and pretending a good result.

Finally, the ultimate consequence of all Test Smells is that the test becomes hard-to-maintain, maybe untrustworthy and obsolete. In the worst case the test is not used any more or thrown away without an appropriate replacement.

One other important thing we learnt about Test Smells taken from the literature and our own experience from the analysis is that they consist of multiple *smelling aspects* or characteristics. A smelling aspect is a part of a Test Smell, either another Test Smell or a more fine-grained and concrete unit of a design problem. For example, an *Eager Test* is by definition a test referencing and executing many different methods while the test is running. But it can also be long test or include many comments the different behaviors of the test, consists of building blocks, code duplication or simply tests a lot of data.

Moreover, Test Smells and their causes and consequences *overlap* and are *interconnected* to each other. We give three short examples to illustrate that:

- *Obscure* tests [2] are hard to understand and badly document the application due to their length and complexity. However a *complex* test does not necessarily need to be *obscure* as maybe only the code-under-test is complex; the test itself is not hard to understand.
- *Long* tests are most of the times *obscure*. However, *obscure* tests are not necessarily *long*.
- *Erratic* tests are those that produce an alternating test result. *Erratic* tests are always *obscure* tests. However *obscure* test are rather seldom *erratic*.

To get a better understanding of this interconnection we visualized this fact in Figure 3.1 and Figure 3.2 using a network diagram implementing causes, effects, smelling aspects and Test Smells.

Figure 3.1 shows a strongly demagnified overview of the network to give an impression of the interconnection. Dark-orange boxes are abstract Test Smells like described in literature [Mesz07a], light-green rounded-boxes are possible and suggested refactorings and white and light-gray boxes represent the possible effects, consequences and items causing the problems. The

[2] *Obscure* tests are often referred as *Long* or *Complex* Tests in literature. However we regard them not as equivalent but rather as a combination of those.

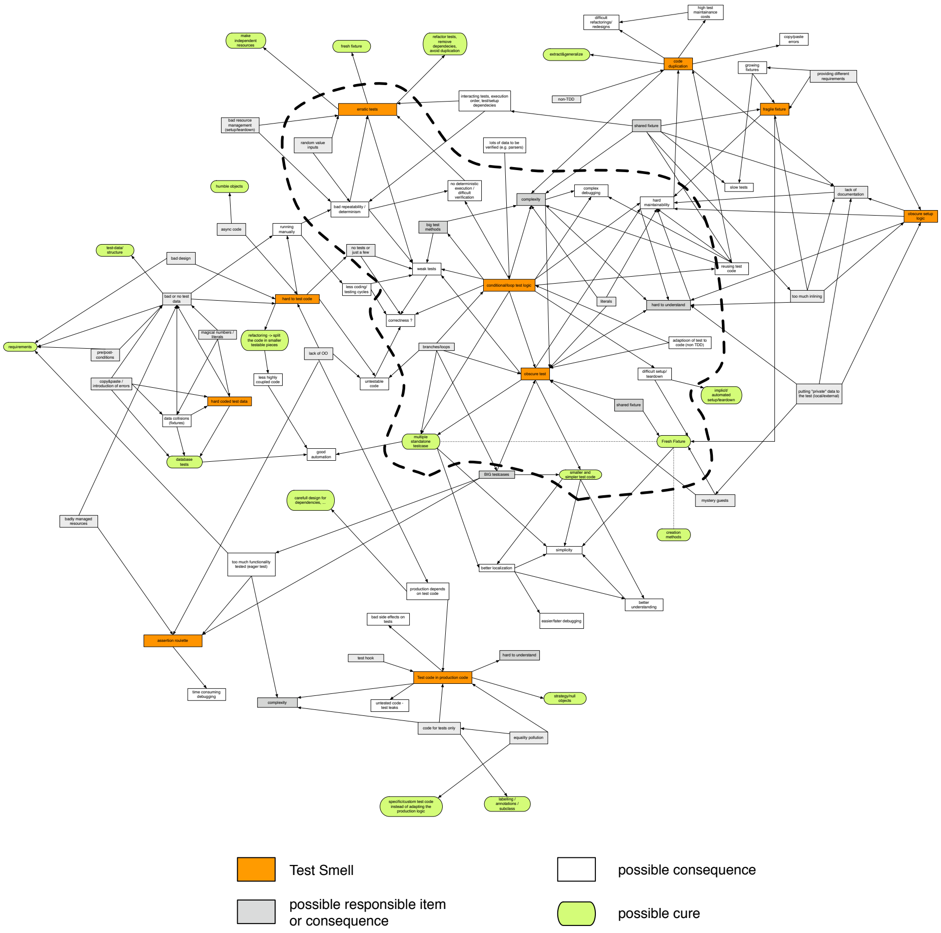

difficult refactorings/ redesigns
high test maintainance costs
make independent resources
fresh fixture
refactor tests, remove dependecies, avoid duplication
extract&generalize
code duplication
copy/paste errors
growing fixtures
providing different requirements
non-TDD
interacting tests, execution order, test/setup dependecies
erratic tests
fragile fixture
bad resource management (setup/teardown)
shared fixture
random value inputs
lots of data to be verified (e.g. parsers)
humble objects
no deterministic execution / difficult verification
slow tests
complex debugging
complexity
hard maintainability
lack of documentation
bad repeatability / determinism
big test methods
obscure setup logic
async code
running manually
test-data/ structure
no tests or just a few
bad design
weak tests
reusing test code
less coding/ testing cycles
conditional/loop test logic
too much inlining
bad or no test data
hard to test code
literals
hard to understand
correctness ?
requirements
magical numbers / literals
refactoring -> split the code in smaller testable pieces
branches/loops
adaptioon of test to code (non TDD)
pre/post-conditions
lack of OO
obscure test
difficult setup/ teardown
less highly coupled code
untestable code
copy&paste / introduction of errors
implicit/ automated setup/teardown
putting "private" data to the test (local/external)
hard coded test data
shared fixture
data collisions (fixtures)
multiple standalone testcase
Fresh Fixture
good automation
database tests
BIG testcases
smaller and simpler test code
mystery guests
carefull design for dependencies, ...
badly managed resources
creation methods
simplicity
too much functionality tested (eager test)
better localization
production depends on test code
better understanding
easier/fater debugging
bad side effects on tests
assertion roulette
hard to understand
test hook
Test code in production code
time consuming debugging
strategy/null objects
complexity
untested code - test leaks
code for tests only
equality pollution
specific/custom test code instead of adapting the production logic
labelling / annotations / subclass
Test Smell
possible consequence
possible responsible item or consequence
possible cure

Figure 3.1: Overview of the Test Smell network. The dashed line marks the extract taken for Figure 3.2

dotted circle marks the extract taken for Figure 3.2. An arrow can be interpreted as “leads to”.

When we look at the extract of the network it becomes clear that by finding one Test Smell or smelling aspect we also have evidence for another design problem or even more than one. Furthermore consequences are revealing smells and give evidence for further smells.

We notice that we visualized only a very small selection of Test Smells collected and analyzed during our study in Figure 3.1 and Figure 3.2. By including other smells into the network, it would become larger and more complex. Furthermore the density of interconnections would rapidly increase.

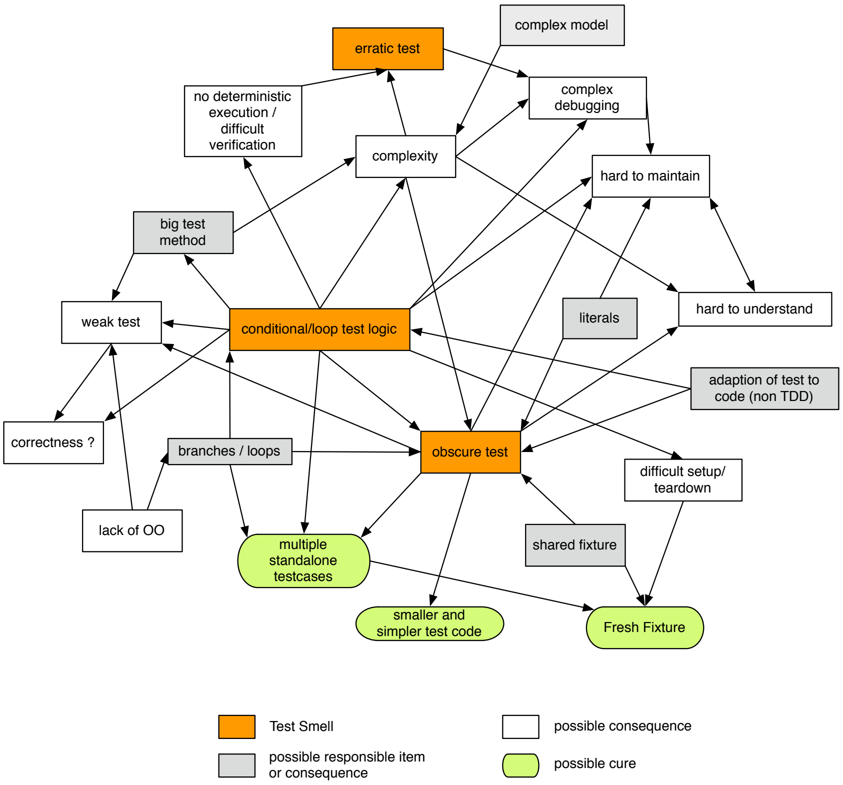


Figure 3.2: A small part of the network from Figure 3.1 showing the interconnection of Test Smells, their likely consequences, items that cause them and possible cures

### 3.1.2 Synthesis

As a first step of our synthesis we mapped the various fine-grained smells and their consequences to already known abstract Test Smells, and distilled the results gathered by the analysis (Figure 3.1) by removing redundancy and clearing out causes, effects and consequences.

We also organized each smell for which we have clear results, taking categories from literature [Mesz07a] and introducing new ones.

We notice that a clear categorization is impossible to achieve as certain smells have characteristics of multiple categories. For example the smell *Under-the-carpet failing Assertion* (explained in details in Section 4.2.1)

| **Category** | **Description** |
|---|---|
| Code | design flaws within the code |
| Behavior | issues that only arise when executing tests |
| Organization | problems about organizing tests and keeping suites tidy |
| Convention | abuse or disregard of common coding or testing conventions |
| Sensitivity | problems caused by changing the environment or its parameters |
| Project | smells that appear as a consequence of an insufficient project planning or leading |

Table 3.2: Test Smell Categories

has code and behavioral smells. The first one because it contains relevant assertions in comments, second because those assertions fail on execution. We give a full list of all smells and their assigned categories in the Appendix.

At the end of our synthesis we prepared various possible formalizations for each smell for an automatic detection. During this process we gathered new insights about Test Smells:

First, we can reuse one fine-grained formalization to detect multiple smells as smells are interconnected and share some of their characteristics. Therefore finding one smell might reveal evidence or even find another ones. We can use this circumstance to set up a list of primitive detection rules and combine them to detect fine-grained and abstract Test Smells.

Second, depending on the smell we have to choose and apply a different detection methodology, or to combine multiple different ones. For example, we cannot use *Pattern Matching* to decide whether or not a method is or contains an assertion to detect an *Assertionless Test* (Section 4.1.5). For this we have to query the interface of the Unit Testing Framework– because that one is supposed to know how an assertion looks like – and analyze the executed methods during the runtime. Although this approach works fine for *Assertionless Tests* it is not efficient or suitable for detecting *Magic Literals* (Section 4.1.4). For those we need another technique, for example a parse-tree analysis.

Third, behavioral smells require additional information which a static analysis can not deliver as it would be too imprecise and unreliable. Such smells require a runtime analysis, for example with a previous code instrumentation. Therefore we can also categorize Test Smells into static and dynamic smells whereas dynamic smells often also include a static analysis.

Finally, we notice that not every smell can be detected by simply applying an algorithm to the test. For example there is no way to detect *Context-Sensitivity* smells. Such smells let a test suddenly fail although there was neither a change to the source nor to the test. Therefore it would be necessary to supervise and analyze the context during the development process to make any assumption related to Test Smells. However in many cases this might not be efficient or possible. Furthermore, quality analysis is mostly done near the end of a project but not in-between.

We give details of possible formalizations on a selection of Test Smells in Chapter 4. The full list of processed Test Smells is given in the Appendix.

## 3.2 Detection Model

As TestLint was basically inspired by Smalllint, but doing the analysis on tests, we decided to take over some of its conceptual ideas. Therefore we also implemented a *rule*-based model similar to the one in Smalllint. However there are some major differences between the model of Smalllint and the one of TestLint.

First, TestLint does not define its own dedicated query language, but rather implements all rules in plain Smalltalk. The main reason for this is because TestLint is an evolutionary and experience-based implementation and prototype. Therefore developing a dedicated language would have been difficult as not all knowledge was a priori defined and fully known. Besides, using plain Smalltalk has several advantages when analyzing tests:

- *Complexity* is highly reduced as no special language needs to be learnt. This increases understanding and comprehension.
- *Flexibility* is increased as each rule has a different context and requirements. Furthermore the goal of a rule depends only on its implementation and not on the capability of a query language.
- *Dynamic*. Many rules are dynamic or context-sensitive. They require additional information by running or instrumenting the tests. However this is extremely hard to specify in a static rule or query.
- *Fuzziness*. Most smells are fuzzy. We can utilize this circumstance as an advantage to formalize fuzzy and context-sensitive rules implementing various heuristics to increase the detection accuracy.

Furthermore the results of our analysis and synthesis (Section 3.1.2) showed that a dedicated methodology like a query language would likely not be appropriate for the detection of all kinds of smells. Due the fuzziness and context sensitivity of certain flaws, they need their own dedicated technique to more precisely detect them.

Second, TestLint represents all *rules* as instances of classes, organized in a class-hierarchy whereas Smalllint defines each rule as a method in the Rule class. This further increases the flexibility for the rule-implementations and reduces the load of a single class, enhancing a rule's documenting character. Finally, we introduce different types of *nodes* which allow us to analyze more than just test-methods.

### 3.2.1 Rules and Nodes

A *rule* is basically an object that takes a *node* as argument, applies some detection strategies to that node and finally should return a result. The result can be a boolean, deciding whether the node has the smell being detected by the rule, the *reasons* why it has smells or a state notifying whether and why the analysis was dropped or not successful. Figure 3.3 shows the basic concepts of the rule-node model.

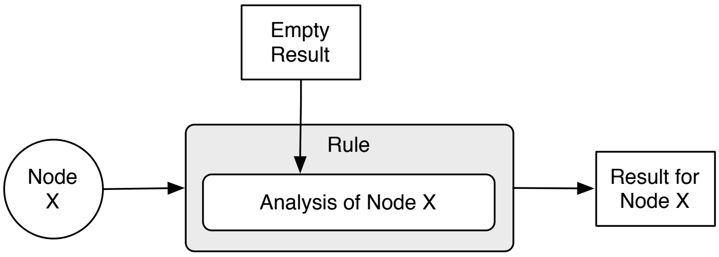


Figure 3.3: A rule takes a node as argument, produces an empty default result and applies its analysis to the node, saving the results in the result object.

To reduce the complexity of each rule and based on the results of Section 3.1.1, concluding that Test Smells are interconnected, overlapping and have different smelling aspects, we decided to map rules to the previously analyzed concrete and low-level design problems. Therefore a TestLint-rule might only cover one or multiple smelling aspects of a Test Smell, but not all of them. Furthermore a rule could even partially cover multiple Test Smells at once.

We organize rules following the categories (Table 3.2) resulted from our analysis and formalizations. We notice that some rules might by organized by multiple categories as most smells cannot be assigned to one single category. Furthermore there are two basic types of rules:

- Static rules to detect static Test Smells (Section 4.1) based on the analysis of the test-code or its abstract syntax tree.

- Dynamic rules that allow the detection of static and dynamic Test Smells (Section 4.1) by instrumenting or analyzing the tests at runtime.

A *node* is a simple abstraction of a standard language entity, for example a package, class or method. Each type of node can have an arbitrary number of Test Smells, depending on the abstractness and number of rules implemented. Furthermore each node can have different types of Test Smells. Table 3.3 maps the different kinds of Test Smells that could apply to a particular node.

| **Node** | **Types of Test Smells** |
|---|---|
| Test Method | Static and Dynamic Smells, Testing Conventions, Naming |
| Test Class | Hierarchy, Testing Conventions, Naming |
| Test Suite | Structure, Test Order |
| Package, Class- and Method-Categories | Organization, Testing Conventions, Naming |

Table 3.3: Nodes and their possible types of Test Smells

We give further details about our contribution of formalizing automated rules and detecting Test Smells in Chapter 4 and Chapter 5. The Appendix includes in-depth information about the model and implementation of TestLint as well as an extensive list of Test Smells we detected in our study.

## Chapter 4

# Test Smells

In this section we present an overview of our contribution of formalizing Test Smells based on our empirical analysis. We declare a selection of Test Smells and the rules to detect them using TestLint. A first section covers static smells, and the second gives a list of dynamic ones.

The section-templates about Test Smells are structured as follows: Each smell is given by a unique, simple and meaningful *name*. We declare the *node* it applies to, respectively what kind of node the detection rule expects. Furthermore we define a set of smelling aspects or characteristics the Test Smell belongs to or *concerns*.

We describe all smells in details and explain how the smells shape, what smelling aspects they show and which problems and consequences they cause. Furthermore we give code examples for the smells and the formalization we used to detect them. All code examples use Smalltalk syntax as we implemented TestLint in Squeak. A comprehensive list of Test Smells can be found in the Appendix.

## 4.1 Static Smells

A static Test Smell is the simplest form of a smell and can be analyzed without actually running or instrumenting the test or the test-suite. As we've chosen not only test-methods for the subject of our analysis, but also test-classes, we define two types of static smells:

- *Static Class Smell.* Class smells are rather abstract and focus on the correct organization and categorization of the tests as well as naming convention. Besides, we can for example analyze whether the test-class is accordingly used based on the Unit Testing Framework, or the test-suite is correctly setup.

- *Static Method Smell.* Method smells can basically be detected by applying metrics to the source-code. Another way is to parse the source-code and analyze the resulting parse-tree by scanning for specific nodes and tokens, or to detect patterns in the source-code or parse-tree. Finally it is also possible to make assumptions about applied coding conventions, in particular analyzing the name of a test-method.

We notice that in the literature static Test Smells refer only to the analysis of test-methods, and mainly focus on parsing and analyzing the source-code as well as applying metrics to it.

### 4.1.1 Improper Test Method Location

***Node:*** *Test Class*
***Concerns:*** *Test Conventions, Organization, Documentation*

Smalltalk-developers normally organize their source-code in packages, class-categories and method-categories. Although there is no common convention for doing so, most projects we encountered during our case study are organized in a very similar fashion. However we also noticed that tests are sometimes excluded from that.

Badly organized tests tend to obfuscate the purpose of a test-suite as for example it is not obvious at the first glance what features or requirements they test or how abstract they are. Furthermore it is hard to find out which functionality of the application code is not (yet) tested. In short, the test-suite is harder to understand and badly documents the application code.

Therefore we claim that developers should apply the same or similar organization to their tests as they do for their sources. We claim that a good organization of tests can help in finding and understanding them more quickly. Furthermore as tests are well organized and the understanding is simplified they also help in better understanding the features and requirements of the application code. A tidy organization can also speed up the development and future refactoring processes as tests have an *obvious location.* Finally well organized code and represents a certain *quality standard.*

As there are no declared *testing conventions*, in particular how tests should be organized, we propose a fuzzy formalization, shown in Figure 4.1, based on testing patterns we've found in our case study.

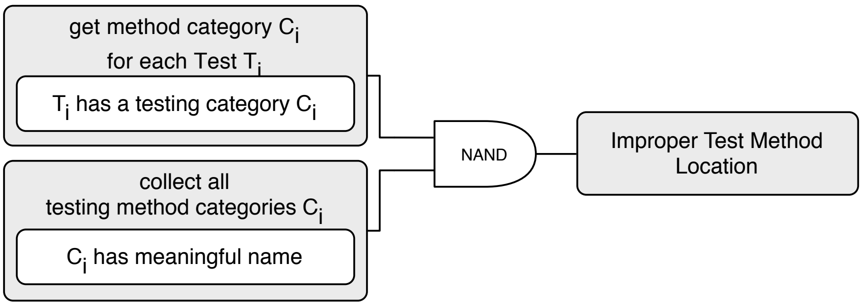

get method category $C_i$
for each Test $T_i$
$T_i$ has a testing category $C_i$
collect all
testing method categories $C_i$
$C_i$ has meaningful name
NAND
Improper Test Method Location

Figure 4.1: Schematics of *Improper Test Method Location*

The first condition in Figure 4.1 checks whether each test is contained by a testing method-category. That is a method-category that contains only test-methods and the name of which matches the patterns `test*` or `run*`. The second condition makes sure that the name of the category is meaningful. For example condition two would reject names like `test1`. As both conditions combined represent a *Proper Location* we concatenate them with `NAND` to get an *Improper Location.*

We notice that the analysis of names is fuzzy, subjective and the purpose of names strongly context sensitive. Therefore this rule could produce false positives.

### 4.1.2 Mixed Selectors

***Node:*** *Test Class*
***Concerns:*** *Test Conventions, Organization, Documentation*

The problem of mixing up all the methods of a test-class is that it is harder to allocate and differentiate accessors, fixtures, utilities and test-methods. By putting each type of method into a different method category, especially strictly separating test-methods from other methods we get a better structure of the test-class. A better and cleaner structure helps in understanding the test-suite, the fixtures and all the test-methods.

As it is hard to estimate or analyse the purpose of a method our rule for detecting *Mixed Selectors* differentiates only between test-methods and non test-methods.

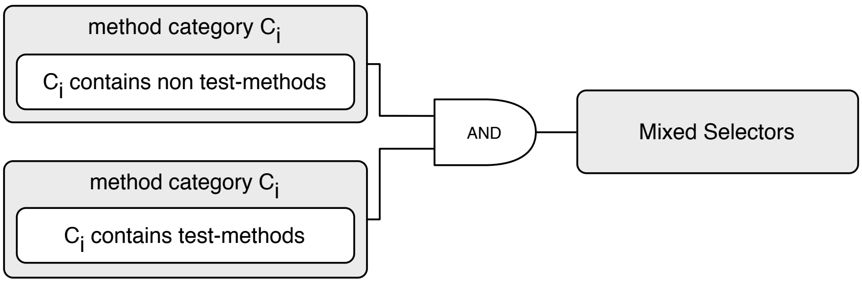


Figure 4.2: Schematics of *Mixed Selectors*

We regard *Mixed Selectors* not as a serious smell as it does not affect the functionality or the runtime of tests, nor does it badly influence refactoring actions. Due to the simplicity of our formalization we can completely exclude false positives.

### 4.1.3 Anonymous Test

***Node:*** *Test Method*
***Concerns:*** *Test Conventions, Documentation*

An anonymous test is a test whose name is meaningless as it doesn't express the purpose of the test in the current context. However tests can be regarded as documentation, and the name is an important part of that as it should abstract what the test is all about.

We detect anonymous tests by analyzing the signature of a test-method. For that, we split the test-method name by numbers and following camel-case notation, and then we check whether *all* the obtained tokens are found in the names of the application classes or methods. We do this by applying a simple pattern matching.

As an example, the rule to detect anonymous tests rejects method names like test1 to test31, but might allow testSHA256 if its context defines cryptographic objects or methods implementing 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 out implementation we use a simple and fast pattern matching applied on each part of the obtained tokens of the test-method name. We notice that other algorithms or heuristics might produce better results.

### 4.1.4 Literal Pollution

***Node:*** *Test Method*
***Concerns:*** *Obscure, Obfuscation, Multiplication, Costs*

When writing tests for the application code it is mostly required also to provide some data to be able to test the functionality. This is mostly done by defining literals in the test code. However an excessive use of literals can cause severe problems:

- Too many literals are distracting and obfuscate the functionality and purpose of a test. This makes a test hard to read and understand.
- The same or similar test data is often repeated within a test or test-suite. This is often a consequence of simply extending or adding tests without actually designing them. The result is a test-suite that is extremely hard to maintain and refactor. We detected such *Duplication* in harvesting our case study.

```
VariableEnvironmentTest >> #testAddRemove
  | refs |
  refs := VariableEnvironment new.
  refs
      addClass: RefactoringManager
      instanceVariable: 'refactorings'. ...
  refs
      removeClass: RefactoringManager
      instanceVariable: 'refactorings'. ...
  refs
      addClass: RefactoringManager
      instanceVariableReader: 'refactorings'. ...
```

We further argue that test data, especially data that is reused, should in general not be hardcoded into the test code, but rather stored somewhere else or accessed using accessor, example or factory methods. This further increases the maintainability as only the data-source has to be modified, but not the test or even multiple tests.

```
UrlTest >> #testUsernamePasswordPrinting
  #(  'http://user:pword@someserver.blah:8000/root/index.html'
      'http://user@someserver.blah:8000/root/index.html'
      'http://user:pword@someserver.blah/root/index.html'
  ) do: [ :urlText | self should: [ urlText = urlText asUrl asString ] ].
```

We formalize the detection of *Literal Pollution* by simply parsing the test-method for literals and rejecting any literal that represents an existing class or selector. We don't detect data duplication and don't define any kind of threshold as the use and purpose of literals is sensitive to the context.

The negative effects of *Literal Pollution* can be decreased, but mostly not completely eliminated by defining accessor methods or a database returning the literals. This would heavily reduce the number of duplicated literals within a test-suite and enhance the readability and comprehension of a test.

### 4.1.5 Assertionless Test

***Node:*** *Test Method*
***Concerns:*** *Pseudo-Test, Anti-Test*

A test that does not contain at least one *valid assertion* is not a *real test* as it does only execute plain source-code, but never assert any data, state or functionality. Besides, it can either succeed or throw an error but can never throw an assertion failure, unless thrown explicitly, which should not be done.

We define a *valid assertion* as one that is either provided by the underlying Unit Testing Framework or 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
```

An *Assertionless Test* is a weak test 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 such a test:

```
ICCreateCalendarTest >> #testCreatingSeveralCalendars
  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 (shown in Figure 4.3). 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.

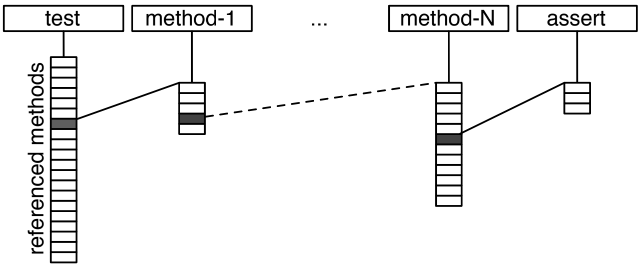


Figure 4.3: A simple but weak way of checking for assertions by checking each referenced method in the parse-tree

Although *Assertionless Tests* do not directly cause any serious problems they should be avoided or at least handled with great care as the intention of a test is to assert data, functionality or state, and not only whether the code runs without errors. In fact, code is often running without raising any error, but still working improperly. This might cause unexpected misbehavior and is hard to detect and debug. Moreover it is more difficult to understand a test without assertions as they normally document the relevant functionalities and objects under test.

### 4.1.6 Overreferencing

***Node:*** *Test Method*
***Concerns:*** *Coupling, Multiplication, Obscure*

Another problem very similar to *Literal Pollution* is *Overreferencing.* It is about test-methods referencing many times classes from the application code.

The main problem with an *Overreferencing Test* is that it causes a lot of unnecessary dependencies towards the model code. That distracts from the goal of the test. In our experimentations, such tests were also rather long and obscure. Furthermore we have detected overreferencing as a source for subtle code duplication and missing generics in the test code which makes it hard or impossible to maintain: different and slightly different fixtures are present in different test-methods.

Figure 4.4 shows how overreferencing can be detected. The first condition checks for the different referenced types. The second one counts how often the same type is referenced in the code. From our experimentations, we have found 3 to be a good value for the thresholds. It captures all the projected and really smelling tests while producing only a very small amount of false positives.

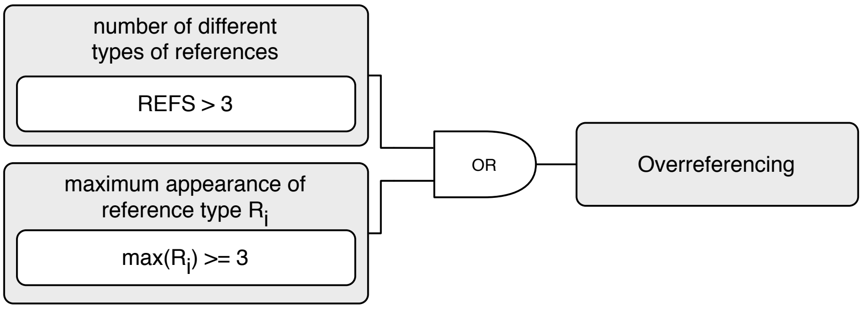

number of different types of references
REFS > 3
maximum appearance of reference type $R_i$
$\max(R_i) >= 3$
OR
Overreferencing

Figure 4.4: The Overreferencing Test rule.

The example below emphasizes and shows the problem of the smell *Overreferencing* and the negative side-effects (*e.g.,* code duplication, coupling) it can have. Applying our rule on the example test, we would get the value 5 for the first condition and 14 for the second one:

```
BooleanTypesTest >> #testTrueFalseSubtype
  | system boolType boolMetaType |
  system := TPStructuralTypeSystem new.
```

```
boolType := TPClassType on: Boolean.
self assert: (system is: (TPClassType on: True) subtypeOf: boolType).
self assert: (system is: (TPClassType on: False) subtypeOf: boolType).
self assert: (system is: (TPClassType on: False)
                     subtypeOf: (TPClassType on: True)).
self assert: (system is: (TPClassType on: True)
                     subtypeOf: (TPClassType on: False)).
boolMetaType := TPClassType on: Boolean class.
self assert: (system is: (TPClassType on: True class)
                     subtypeOf: boolMetaType).
self assert: (system is: (TPClassType on: False class)
                     subtypeOf: boolMetaType).
self assert: (system is: (TPClassType on: False class)
                     subtypeOf: (TPClassType on: True class)).
self assert: (system is: (TPClassType on: True class)
                     subtypeOf: (TPClassType on: False class)).
```

*Overreferencing* can be fixed using different techniques, depending on how the smell manifests in the code. For example it would make sense to define a shared fixture setting up all the Objects if that smell appears more than once in the test-suite, and all the class-references are used to instantiate basic objects. In cases like in the example above it would be reasonable to define accessor or generic (factory) methods that return the reoccurring code elements such as TPClassType on: True class. User-defined assertions might also be an valuable option.

### 4.1.7 Overcommented Test

***Node:*** *Test Method*
***Concerns:*** *Documentation, Model Complexity, Descriptive Encapsulation Break, Obscure*

*Overcommented Tests* define too many comments, obfuscating the code and distracting from the purpose of the test.

As tests can be regarded as a form of requirement-specification and documentation the model code, it is conceptually unnecessary to write comments within a test. Especially comments about the model code and how to treat it is a Test Smell. They are redundant and likely to become obsolete in the future. Furthermore they're defined at the wrong place (Descriptive Encapsulation Break).

Besides, too many comments within the code break the “reading flow”, making the test even harder to understand, therefore achieving the opposite of the purpose of comments. A lot of comments also indicate that the model code is complex, however this should not be commented but rather refactored – “good code only need tests and a very few comments”.

We detect overcommented tests by parsing the test code, extracting and analyzing the comments within the test context. We correlate the number of comments to the size of the shared fixture and the number of assertions:

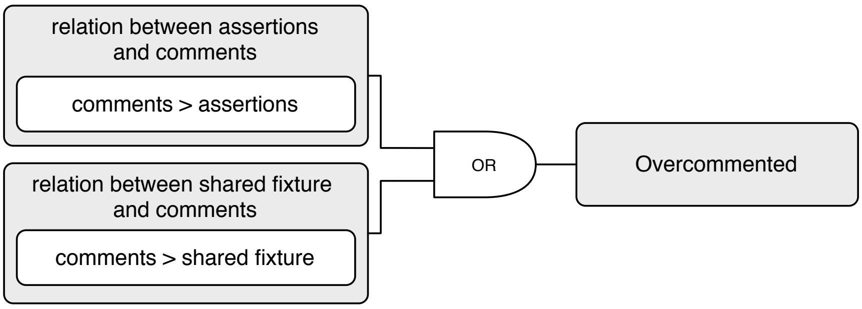

relation between assertions and comments
comments > assertions
relation between shared fixture and comments
comments > shared fixture
OR
Overcommented

Figure 4.5: The Overcommented Test rule.

This means that for every assertion or instance variable used in the test is actually allowed to have one comment of any size at most. For example the following test is overcommented as the relation evaluates to $7 > 3$:

```
DebuggerTest >> #testUnwindDebuggerWithStep
  "test if unwind blocks work properly when a debugger is closed"
  | sema process debugger top |
  sema := Semaphore forMutualExclusion.
  self assert: sema isSignaled.
  process := [sema critical:[sema wait]]
               forkAt: Processor userInterruptPriority.
  self deny: sema isSignaled.
  "everything set up here - open a debug notifier"
  debugger := Debugger openInterrupt: ’test’ onProcess: process.
  "get into the debugger"
  debugger debug.
  top := debugger topView.
  "set top context"
  debugger toggleContextStackIndex: 1.
  "do single step"
  debugger doStep.
  "close debugger"
  top delete.
  "and see if unwind protection worked"
  self assert: sema isSignaled.
```

Although our formalization seems to be weak by allowing quite a lot of comments, it actually proved to do the opposite in our case study by being very restrictive and focusing only on the very bad and obviously overcommented tests. During our analysis we detected numerous tests with many comments inside. Some of them could be identified as *Overcommented Tests*. In most of those cases either the test, the model under test or both showed design flaws.

We further notice that analyzing the concrete purpose of a comment is generally not possible or can only be approximated using heuristics as this would require one to fully understand the semantic meaning and the context of the comment. Therefore false negatives and false positives are inevitable.

### 4.1.8 Long Test

***Node:*** *Test Method*
***Concerns:*** *Long Tests, Complex Test, Documentation*

A *Long Test* is a test that consists of lot of code and statements. Such tests are mostly (but not necessarily) complex and badly document the purpose of the test and the application code. Furthermore they tend to test too much functionality, maybe even getting *eager*.

We detect long tests by simply counting the number of statements, ignoring the context, the purpose and functionality being tested. We then evaluate this number against a fixed threshold.

Our experimentation showed that a threshold above 10 to maximum 20 statements is a very good threshold to detect long tests. However we notice that due to the simplicity of this formalization it is quite likely to get false positives. On the other hand we detect really long tests and also slightly longer tests extremely quickly. We detected numerous *Long Tests* in our analysis and case study.

If a *Long Test* shows evidence an *Eager Test*, for example by testing various features of the model at once, it can be easily refactored into multiple smaller and distinct units. The same applies for tests which test a lot of different data values. Tests which verify a large or complex model are probably *Long Tests* and often difficult or even impossible to eliminate.

### 4.1.9 Guarded Test

***Node:*** *Test Method*
***Concerns:*** *Control Flow, Hidden-Test, Obscure*

*Guarded Tests* include boolean branching logics like ifTrue: or ifFalse:.

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

The problem of using such conditionals is that they break the linearity of a test by controlling the execution flow, making the test less predictable and harder to understand. The documenting nature of a test might vanish. Furthermore it could be that certain assertions are not executed. In the worst case an actual failing test returns a success letting the developer believe the test is green. Moreover, *guarding clauses* might reveal an encapsulation break of the application code by exposing the internal model logic to the tests. This also increases the coupling between the tests and the sources and leads to fragile and trust-unworthy tests.

We identify *Guarded Tests* by scanning the abstract syntax tree of a test-method for all occurrences of any conditional logic. The code-example from above would be identified as an instance of a *Guarded Test*

In particular we do not differentiate whether the conditional includes asserting or non asserting code, and we do not try to analyze the intention of the conditional as we regard the use of them as a general smell for tests.

We notice at this point that many projects we harvested use conditionals with a specific purpose to drop certain tests. In these situations we could regard *Guarded Tests* as false positives. We give some examples of such situations in the following:

- (Extremely) slow tests which badly influence the runtime of a test-suite, endangering the automation and manual execution of tests
- Platform-dependent tests
- Tests accessing an external resource like a database which might not always be accessible

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 Tests*, when used for such purposes, can be regarded as false positives. Unfortunately it is extremely difficult or mostly impossible to automatically identify the meaning and intention of a piece of code. Therefore false positives cannot be eliminated. However because of the following reasons we regard the way those tests are realized as a Test Smell:

- Conditional code is repeated like a pattern all over the tests (code duplication) 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 even if they didn't run any code or assertions. However this is critical as it is not possible to decide whether a test actually ran or not.

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.

A simple way of extending an unit testing framework with such a feature could be by creating a new subclass ConditionalTestCase of the class TestCase which must implement a condition that evaluates before a test can run. Like that the testing framework would know why it cannot run certain tests and notify the tester about that circumstance. Extending the exception mechanism could be another solution.

## 4.2 Dynamic Smells

Dynamic Test Smells require the test to be run to assess quality information. Therefore dynamic analysis applies to test-methods only. However almost all dynamic smells also include static analysis.

For most dynamic Test Smells it suffices to execute the test once, but there are also some that require multiple runs. Some Test Smells require an instrumentation of the application code, the test fixture or the test itself. As not all code can be successfully instrumented, these smells include an uncertainty factor and might therefore not return a result.

Dynamic test analysis is more complex and runs much slower than the static ones. Furthermore results are harder to interpret due to the amount and complexity of information collected at runtime.

### 4.2.1 Under-the-carpet failing Assertion

***Node:*** *Test Method*
***Concerns:*** *Pretending Test, Hidden Failure, Obscure*

A test having the smell *Under-the-carpet failing Assertion* is a test that returns a successful test-result, but contains *hidden assertions.* A hidden assertion is an assertion that is put into comments, is not executed when the test runs, and which would actually throw an Error or Failure if the comment were removed.

The problem of such a test is that it drops one or maybe even all assertions, pretending a good test-result. Thats because a Unit Testing Framework is normally not aware of the assertions it executes, but only about Errors and Assertion Failures raised. This lets the developer believe the application code works properly. The following code gives an example for an *Under-the-carpet failing Assertion* as it can be found in many test-cases. We notice that the *hidden assertions* are not visible without code highlighting at first glance:

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

A closer analysis of the context of the commented code in the example code above reveals that the method categoryWithSummary: can throw an Error if aString is not detected in categories – a situation the test doesn't catch due to the comment.

```
ICCalendar >> #categoryWithSummary: aString
  ^ self categories detect: [ :each | each summary = aString]
```

We detect *Under-the-carpet failing Assertion* using a 2-pass procedure, formalized in Figure 4.6. In the first pass we simply run the test and save the result. In the second pass we parse the comments of the tests, identify comments including assertions, safely remove any comment-tokens around valid code and asserting statements and run the test again. Finally we compare both test results.

This Test Smell is important to detect as developers obviously often comment out failing assertions due to several reasons. We have encountered

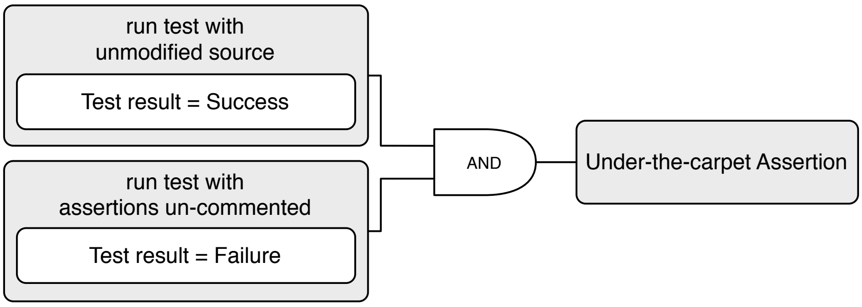


Figure 4.6: Schematics of Under-the-carpet failing Assertion

assertions that were probably commented in the course of debugging activities, and later forgotten to be removed. We have also detected commented assertions that were just obsolete and tests that were completely commented just for the sake of making the test green because they run slow, or require a special environment to run.

### 4.2.2 Badly Used Fixture

***Node:*** *Test Method*
***Concerns:*** *Slow Test, Complexity, Obscure*

A *Badly Used Fixture* is a fixture that is not fully used by the tests in the test-suite. Figure 4.7 shows an example of a badly used shared fixture.

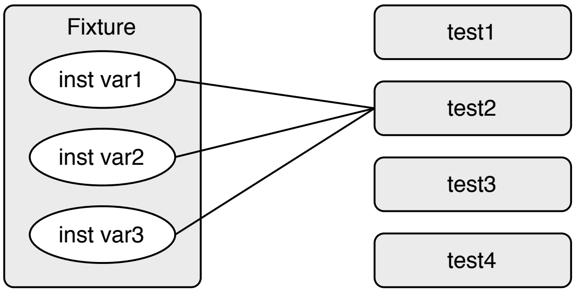


Figure 4.7: A (badly) shared fixture only used by one of four tests

This smell mainly appears in tests that use implicit fixtures. Such a fixture is shared by all the test-methods and mostly includes optional setup and teardown instructions which are executed before and after each test-method. However not all of the tests require this fixture or at least not everything that the fixture provides.

As a consequence tests start to run slowly. That's because the implicit fixture is always executed, even if the tests don't require the data being set up – or only few parts of it. Furthermore, the size of the fixture normally grows with the evolution of tests as extending the fixture is a simple way to setup and share test-data and objects. This further aggravates the problem. Finally as such tests start to run slowly they're also executed more seldom, potentially hiding newly introduced bugs which get overseen or discovered too late.

We detect a *Badly Used Fixture* by first instrumenting all instance variables including all fixture and test-methods of a test-class. In a second step we run the tests gathering all read and write accesses to the shared fixture for each method. Finally we evaluate the information about the shared fixture and apply the following metric (Figure 4.8) to gain knowledge about its usage.

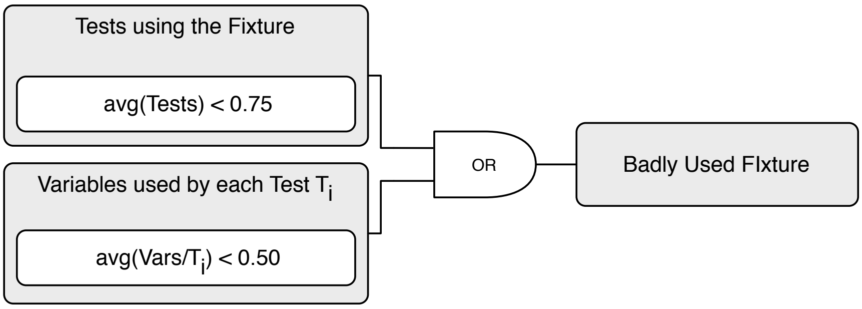


Figure 4.8: Schematics of Badly Used Fixture

The first simple condition in Figure 4.8 says that at least 75% of all tests should use the fixture. The second condition requires that in average all tests should use at least 50% of the variables defined within the fixture. When we apply this metric to the example of Figure 4.7 then the first condition would evaluate to 0.25 (=1/4), and also the second condition (=(0+0+1+0)/4).

False positives cannot be totally excluded as the purpose of an instance variable is difficult to determine, therefore the rule to detect *Badly Used Fixture* is quite sensitive to the context of the test. Furthermore the thresholds we've chosen for both conditions are 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. However, in our experimentations, those thresholds proved to return reliable results in most situations.

### 4.2.3 Transcripting Test

***Node:*** *Test Method*
***Concerns:*** *Test Conventions, Slow Test, Obscure*

A *Transcripting Test* is writing information to the console or a global stream, for example the Transcript in Smalltalk, while it is running. The following code shows an example of method called in one of the test-methods of the test-class HeapTest.

```
HeapTest >> #testExamples
  self shouldnt: [ self heapExample ] raise: Error.
  self shouldnt: [ self heapSortExample ] raise: Error.
```

```
HeapTest >> #heapSortExample
  "HeapTest new heapSortExample"
  "Sort a random collection of Floats and compare the results with ... ''
  | n rnd array  time sorted |
  n := 10000. "# of elements to sort"
  rnd := Random new.
  ...
  Transcript cr; show:'Time for heap-sort: ', time printString,' msecs'.
  "The quicksort version"
  ...
  Transcript cr; show:'Time for quick-sort: ', time printString,' msecs'.
  "The merge-sort version"
  ...
```

Aside from the effect of slowing down the test execution, writing data to the console contradicts the idea of unit testing: A test should only return the result of the test, for example "green" or "failed". It should never output any additional data that has to be processed or harvested manually as this would be distracting and encumber the automation of tests. Furthermore, *transcripting code* obfuscates the test-method and distracts from the goal of the test and its documenting attribute.

We detect *Transcripting Tests* by instrumenting code responsible for writing data to the console, and running the tests, collecting all methods and data written to the console. Any attempt to access the console is recognized as a Test Smell.

### 4.2.4 Interactive Test

***Node:*** *Test Method*
***Concerns:*** *Test Conventions, Slow Test, Obscure*

An *Interactive Test* is a test that interrupts the automatic execution of a test, requiring manual actions from the user, for example pressing a button, closing a window or entering some data.

The problem of such tests is that they contradict the idea of automating tests as they require the full attention of the developer while they're running. Furthermore they're not predictable and repeatable as the input might change for each execution. Besides the result of the test might change depending on the input. For example, the tester once chooses to enter some data and press the "ok" button, another time he selects the "cancel" button, not entering any data which causes a test failure.

To detect an *Interactive Test* in Squeak we instrument the core methods of the package "Morphics"[1]. We use the following code for the instrumentation using ByteSurgeon [Denk06a]:

```
(<:#methods>
    at: RunningSUnitTest current
    ifAbsentPut: [ Set new ])
        add: (<meta: #receiver> class) -> (<meta: #receiver>).

(<:#pattern> match: (<meta: #receiver> class name))
    ifFalse: [ TestFailure signal ].
```

#methods represents a Set containing associations of Morphs that were attempted to be displayed as well as their classes for a particular test (RunningSUnitTest current ). We throw a TestFailure as soon as the test tries to open a Morph. We reject non critical Morphs like progess- or status-bars using a name-based #pattern matching.

[1] Morphics is a graphical library of Squeak to build graphical objects or windows and dialogs

As the detection of interactive Morphs is context sensitive – we cannot analyze the meaning and intention of code – the rule to detect *Interactive Tests* also collects tests that simply open and close Morphs without requiring any interaction. Although those tests don't have the Test Smell of interactivity, we still regard them as badly designed as no unit-test is supposed to open Morphs except the tests of the package Morphics or those specifically testing the GUI.

Chapter 5

# Case Study

The following sections are dedicated to the details of applying TestLint to a large set of tests. Section 5.1 will give a brief overview about the configuration of our case study. Section 5.2 will give an introduction about the distribution of Test Smells. Section 5.3 handles accuracy and significance. Section 5.4 finalizes our global analysis by locating the Test Smells. At the end of our case study we examine a selection of well known packages in more detail and also evaluate the usability of TestLint in action.

## 5.1 Overview

Our case study is based on a Squeak 3.9-7067 image including the latest version of Christo Code Coverage, TestLint and several scripts to easily and quickly recreate the case study data and diagrams. Both tools as well as the case study scripts are Open-Source and can be downloaded from SqueakSource. For mapping and visualizing data we use Microsoft Excel.

In the very first step we harvested from SqueakSource any freely-available and installable packages including tests. We notice that this was quite a time consuming process as many packages are

- obsolete, unmaintained or badly documented,
- difficult to set up due missing documentation, and complex and unknown dependecies,
- cause unforeseen conflicts with other already installed packages
- don't define any tests at all.

We finally set up our case study image with approximately 70 packages including 5’500 test-nodes whereas a test-node is either a test-method or a test-class. Table 5.1 details the configuration of the case study.

| | |
|---|---|
| Packages | 67 |
| Test-methods | 4834 |
| Test-classes | 742 |
| Classes in total | 2355 |
| Methods in total | 27054 |
| Detected Authors | 93 |

Table 5.1: Overview about the composition of the case study

Based on harvesting SqueakSource, we determine that writing tests is not a common or often used technique in the Squeak community. We conclude this due to the fact that SqueakSource contained about 630 projects at that time, however we found only 67 usable packages. Therefore approximately 80% of all packages on SqueakSource don’t define any tests at all.

Besides, we notice that the number of detected authors, declared in Table 5.1, is fuzzy. Many methods are missing “author initials” [1] because they were not declared or cannot be retrieved from the sources. Furthermore some authors are using multiple different, but similar ones to declare their code, for example SR and SReichhart. The case study showed that these situations are exceptions and don’t have any severe side-effect on the results. We therefore didn’t correct this discrepancy for the case study.

To get a first rough impression about the case study we present some further information about the distribution of tests on packages and authors. Table 5.2 shows how tests are spread over the packages in the case study. The differences in average, median and max value let us assume that the distribution of tests is very unbalanced. Therefore we expect that only a few packages define a lot of tests and the rest is provided by the majority of packages.

| | |
|---|---|
| Average Tests per Package | 6 |
| Median Tests per Package | 36 |
| Max Tests per Package (’KernelTests’) | 663 |

Table 5.2: Distribution of Tests on Packages

[1] Each method and class in Squeak is saving its author using an arbitrary string, normally the author’s initials, identifying it.

We can endorse this assumption as we get a similar result when applying the same metrics to authors, shown in Table 5.3. As the average and median value of the number of tests are very low and close together and in consideration of the results by Table 5.2 and the configuration of the case study by Table 5.1 we determine that most authors define only a very few tests. However we also notice that there are also some exceptional authors writing a lot of tests.

| | |
|---|---|
| Average Tests per Author | 7 |
| Median Tests per Author | 2 |
| Max Tests per Author ('lr') | 789 |

Table 5.3: Distribution of Tests on Authors

## 5.2 Distribution of Test Smells

The first questions we wanted to answer is to find out *how many Test Smells* do the tests within the case study have, are there any nodes having *more than one smell*, and *how distributed* are the smells among the tests.

We therefore applied all available rules to each package separately and on the complete image at once. This process took a few hours. As the results of distribution between packages and the complete image is quite stable we can conclude the results shown in Table 5.4.

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

Table 5.4: Distribution of Smells per Test-node

We discover that about 60% of all tests are supposed to be free of Test Smells whereas about 40% of all tests have one or more possible smells. We also observe that nodes having more smells appear less often, in decreasing order. Tests with more than 4 to 5 smells are extremely rare and can be neglected or regarded as exceptions.

We must further notice that we did not filter *false positives* or *true negatives* for this first analysis. This would have been an huge overhead due the size of the case study.

## 5.3 Accuracy and Significance

The next step in doing the case study was to find out *how accurately* are the rules working, which ones need refactorings and further improvements and what kind of changes would be necessary to make them more reliable. Furthermore we wanted to answer *how many false positives* do our rules produce, *how often* does a certain smell really appear and *how important* is it to detect that smell.

We therefore manually analyzed each of the observed Test Smells to exploit the measure of accuracy of our detection rules. Furthermore we also inspected the values and reasons that caused the detection. Like that, we could easily filter out false positives and further gather new valuable criteria to improve the rules. Due the size of the case study and the fact that we designed our detection rules rather restrictively (to capture only the serious problems), we did not do the opposite operation to find *true negatives*.

The graph in Figure 5.1 displays the detected Test Smells and the corresponding rules in descending order based upon the total number and global percentage of their occurrence frequency.

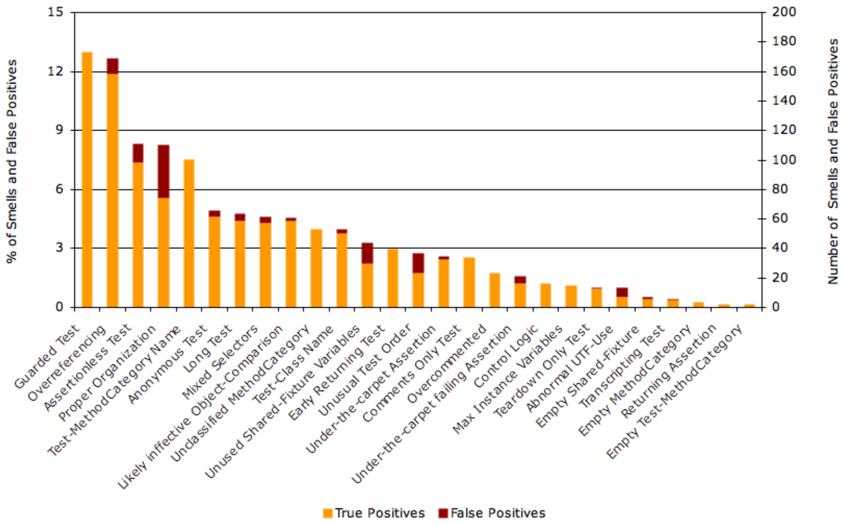


Figure 5.1: Each bar represents the total number and percentage of detected smells by the corresponding rule. True positives are light-orange, false positives are dark-red

Figure 5.1 shows that most of the rules have a high accuracy, not producing too many false positives in general. We notice that the early prototype of TestLint had an overall accuracy of approximately 80% at the time of this case study.

Furthermore we discover that certain smells, especially the ones about *Logics*, *Naming* and *Organization* appear quite frequently. We therefore conclude that these are the most common problem in unit tests (based on the selection of rules we used for this case study). The other Test Smells appear less often and are rather equally distributed.

Figure 5.1 also shows that certain rules like *Proper Organization*, *Abnormal UTF Use* or rules trying to analyze the *Shared Fixture* tend to produce more false positives than other rules. We can explain this with the following reasons:

- Test Smells are fuzzy and context sensitive. For instance, the meaning and purpose of a shared variable in the *Shared Fixture* cannot be estimated. Hence, there is no way to formalize such a rule.
- There is no common convention to test against. Therefore a precise rule cannot be formalized. Example: *Proper Organization.*

Due to these facts it will not be possible to detect certain Test Smells without false positives. However it might be possible to reduce them using fuzzy heuristics.

Including and combining the results after sorting smells by their appearance frequency and accuracy of their corresponding rules (Figure 5.2), we also discover that *conditional* smells including branching logics (*Guarded Test*), missing assertions, smells causing a higher coupling (*Overreferencing*), or breaches of testing conventions (*Assertionless Test*) appear quite often. We can assume they have a higher relevance. We refer to this measure as *Significance.*

Exotic Test Smells like *Transcripting* or *Early Returning Test* appear rather rarely, however they exist. We further notice that on our test data statically analyzed Test Smells appear more often and mostly have a higher accuracy and significance than dynamic ones.

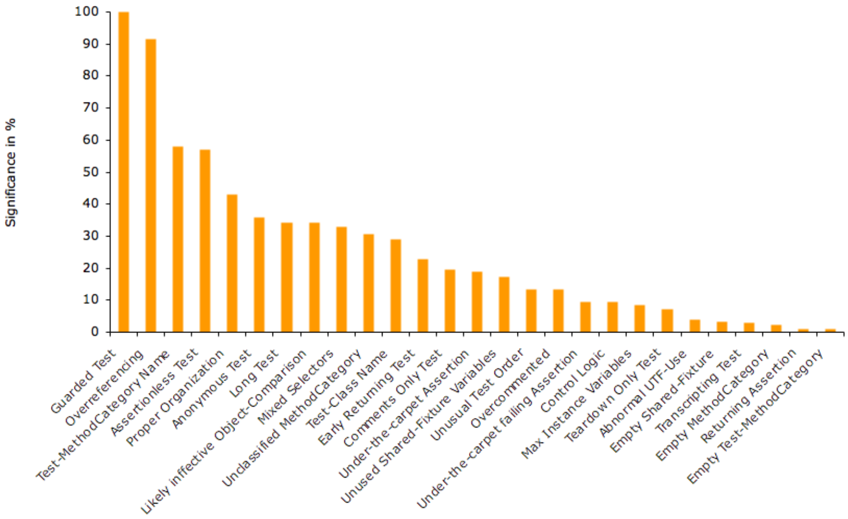


Figure 5.2: Test Smells and their corresponding rules sorted by their appearance frequency and accuracy

## 5.4 Localization

To finalize the case study we also wanted to know *where* we can actually find smells, in particular in which packages and what kind of smells are distributed in which package. Moreover we would like to know *who* creates Test Smells and whether we could find any relation between authors writing a lot of tests and such ones only writing a few tests.

We get a rough impression about the distribution of smells on packages when calculating again some synthetic values based only on the total number of packages and detected Test Smells, neglecting false positives and relations (Table 5.5).

| | |
|---|---|
| Average smells per Package | 32.4 |
| Median smells per Package | 12 |
| Max smells per Package ('KernelTests') | 433 |

Table 5.5: Synthetically calculated smells per Packages

We discover again the larger differences between average, median and maximum values, indicating large differences between the number of smells per package. We therefore expect some packages with a high number of smells and several packages with only a few smells.

We get quite a similar result when doing the same for authors. (Table 5.6). Due to reasons of anonymity and to avoid offending people we do not publish data that directly maps authors, smells and tests.

| | |
|---|---|
| Average smells per Author | 47.6 |
| Median smells per Author | 8 |
| Max smells per Author | 449 |

Table 5.6: Synthetically calculated smells per Author

To further refine localisation results we made a simple mapping between packages and smells to get the relation between packages, smells and tests. We visualize this correlation in Figure 5.3.

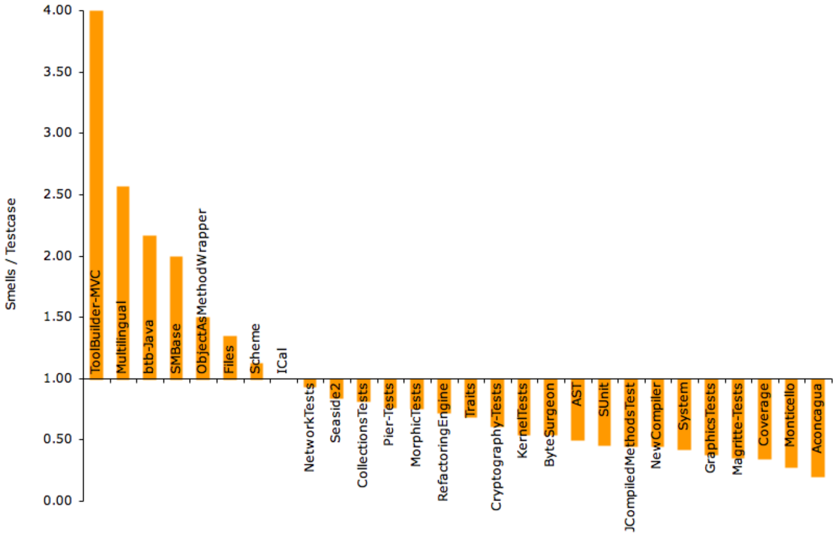


Figure 5.3: Relations between packages and smells per tests

After doing a full manual test analysis on a few packages as shown in Figure 5.3, we can determine that packages, known to have good code and tests, maybe even done by Test-Driven Development like Aconcagua or Magritte tend to have much better tests. This means fewer Test Smells per node than other packages like Tool-Builder or SMBase, containing “Spaghetti”-code and just a very few, bad or almost no tests.

To better visualize the relation of smells, tests and packages we mapped all available data together into Figure 5.4. This graph shows all packages on the x-axis, sorted by number of tests in ascending order. The y-axis show the absolute values of tests (dark-blue) and smells (light-orange). We further clarify the data results by calculating an exponential approximation of tests (dark-blue) and smells (light-orange).

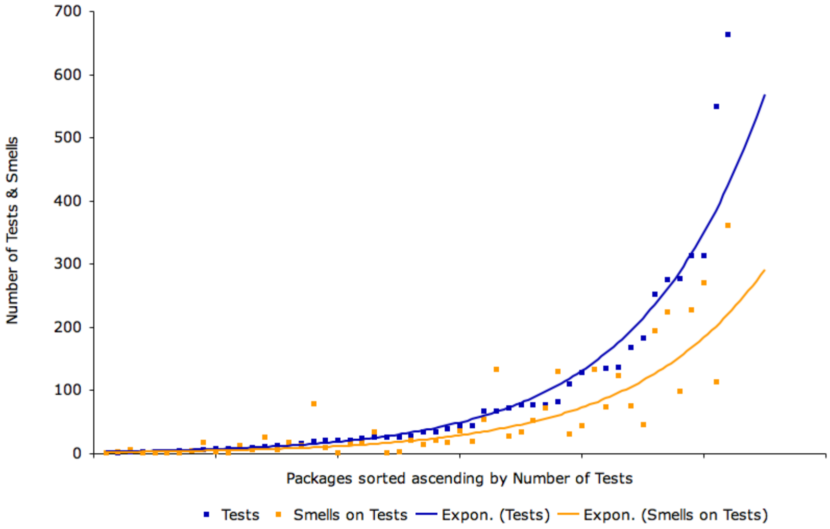


Figure 5.4: Tests and Smells on Packages, sorted by the number of tests in ascending order

Based on Figure 5.4 we determine that Test Smells grow steadily with the number of tests written. Moreover we discover that the number of smells is growing slightly slower than the number of tests and that we mostly have fewer smells than tests within a package, however exceptions exist.

Moreover, Figure 5.4 indicates and lets us assume that developers writing more tests also tend to write better tests with the quality of the tests raising with the amount of tests written.

We could actually prove this assumption by re-mapping the data of Figure 5.4 to authors instead packages. Figure 5.5 visualizes the total number of tests (primary y-axis), total number of Test Smells per test (secondary y-axis) and the authors writing tests (x-axis), sorted ascending by the total number of tests.

We clearly discover that the number of smells per test is decreasing steadily while the number of tests is increasing. So our assumption of Figure 5.4 is proven and we can conclude that developers writing more tests produce

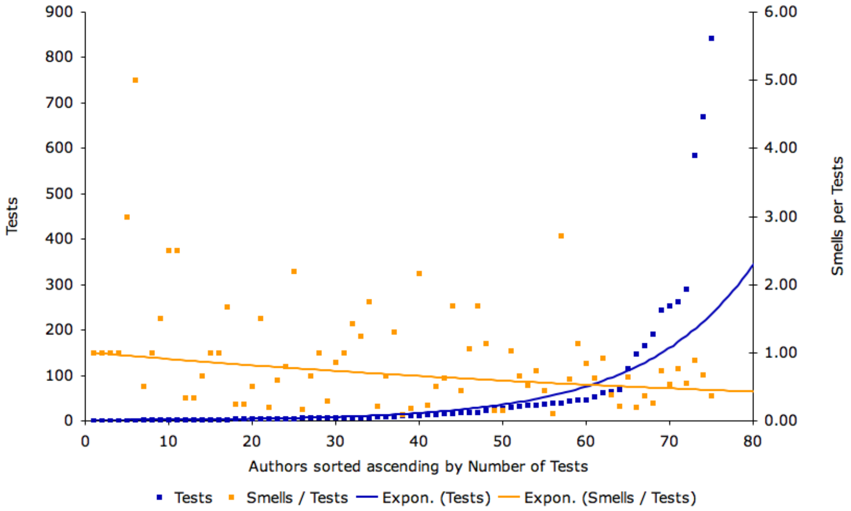


Figure 5.5: Tests and Smells on Authors, sorted by the number of tests in ascending order

proportionally fewer Test Smells than those writing only a few tests, or the other way round. Furthermore we also notice that this conclusion might only apply to the whole case study at once, but might not be true for each individual developer.

Besides, Figure 5.5 is a good and strong argument for doing testing and Test-Driven Development:

The more tests a developer writes the better the tests

So we conclude that testing and especially Test-Driven Development does, at least in long terms, scale extremely well.

An interesting question that arises from this conclusion is whether the quality of the tests positively affects the quality of the application code and design – or the other way around. Although we discovered in our studies that packages having only a few Test Smells mostly also have well-designed sources, and that packages having many Test Smells show lack in the design of the model, we did not further investigate that within this work. We leave this question to the future work on Test Smells.

Finally, the results taken by Figure 5.5 let us expect a package, especially the tests of one author to change, even tend to get better in time. This means earlier packages have more Test Smells than later ones. Although this sounds reasonable, we did not further investigate this assumption due to its complexity of including additional context-sensitive information like time into the statistics. This might be another interesting question for the future work.

## 5.5 Examples

### 5.5.1 Aconcagua

Aconcagua defines about 549 tests and is known to have very good tests, most of them done by Test-Driven Development, in particular Test-First Development. Our manual analysis enforces this by discovering that most tests are rather short and easy to understand, however exceptions exist.

Using TestLint does only show a few Test Smells compared to the size of Aconcagua:

- 14 tests doing *Overreferencing*, however just reaching the threshold of being recognized as such
- 1 class (NumberMeasureProtocolTest) mixing test methods with other methods *e.g.,* fixtures, accessors or utilities
- 3 tests are too long and exceed the maximum number of statements (using 15 as a threshold)

We found several false positives in Aconcagua. In particular we found 6 *Assertionless Tests* in the class EvaluationTests whose assertions cannot be retrieved using static analysis as the tests get re-composed at runtime. We also encountered a lot of *Magic Literals*, however this is to be expected as Aconcagua is about numbers and units. Still the amount is critical.

Using TestLint we actually found all issues of Aconcagua that might be the source of current or future problems. However the number of false positives is quite large, giving the impression that TestLint is not really useful or not well enough formalized – which might be true for packages defining very well designed tests. However we notice that we designed TestLint to catch all possible issues, not to detect them perfectly.

### 5.5.2 Magritte

Magritte [Reng06a] is a meta-description framework to build user-interfaces, reports, queries and persistency. It defines a large number 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.

### 5.5.3 Refactoring Engine

By manual analysis we discover that Refactoring Engine defines a lot of tests, mostly long and complex test methods, equally distributed over multiple classes using method categories to further organize tests. We notice that at the time of this examination Refactoring Engine had a lot of errors and failing tests. Besides, tests run extremely slowly or even let the Squeak image freeze. The latter issue complicated the automatic analysis as certain rules need to run the tests. Therefore we had to repeat this analysis multiple times.

Analysing Refactoring Engine using Testlint identifies it as a package of average test quality, having an average distribution of Test Smells among all tested packages. Using Figure 5.1 we discover that most tests seem to have Test Smells. TestLint reveals a bunch of serious and less serious Test Smells, in particular, Refactoring Engine contains:

- Two *Transcripting* Tests, this is tests writing some data on the console
- 20 tests about *Overreferncing*, therefore strongly exceeding the maximum number of references. All of those tests are also very long and complex.
- 10 tests using *Logic Operators*, heavily obfuscating the test code
- About 25 tests are too long, exceeding the maximum number of statements

- 11 tests have valid code put into comments, whereas 2 tests include *Under-the-carpet Assertion*, one even including a *Under-the-carpet failing Assertion* in ExtraParsingAndFormattingTests
- 7 tests with too many or too long comments

Besides, all tests are infected by *Magic Literals*. However we expect most of them as Refactoring Engine needs to reference and define a lot of data and includes a lot of parsing-related code. Still we do not count all of them as false positives as some literals could have been avoided by a better test design. Finally, shared fixtures in Refactoring Engine are not always fully used and TestLint even recommends to use a fresh fixture in RefactoringTest instead of the large shared one.

We also found several other false positives besides *Magic Literals*. There are approximately 146 tests wrongly detected with the smell *Assertionless Test*. This failure is caused by a special user defined assertion, used in almost every test, which is not correctly recognized. Furthermore TestLint wrongly identified one *Under-the-carpet failing Assertion*, caused by a bug when parsing comments.

The results from TestLint are mostly the same as those obtained by the manual analysis, except for the problems TestLint is not able to detect due to missing rules. However TestLint could also detect flaws that were not detected by manual analysis *e.g., Under-the-carpet failing Assertion*. Although Refactoring Engine has many tests, we detected many smaller but also some more severe flaws. Moreover, many tests fail, throw errors or have to be executed manually.

### 5.5.4 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 tests 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 meaningless names. Furthermore, several test methods are mixed up with non test methods.

Our manual analysis and the results obtained by TestLint are very similar. Especially the organization of tests makes 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.

### 5.5.5 Network

Network defines only about 57 tests, not much for such a big and important package. Our manual analysis mainly concludes many small and low level tests, using a lot of meaningless literals and strings. There are also some code duplications.

TestLint detects the following major issues in Network:

- 2 tests in UUIDPrimitivesTest might return too early due to a *Logic-Operator* and/or dropping all the assertions.
- 3 tests in UrlTest make too many references to the same class. Besides they define too many comments and exceed the maximum length of a test. The manual check reveals heavy *Code Mutliplication* and *Building blocks.*
- Almost all tests in TestURI (45/47) and 4 in UrlTest have meaningless selector names *e.g.,* test1, test2, and so on.
- The class category NetworkTests-Kernel belongs to the testing categories of the package Network but does not define any tests, only a mock object.
- The fixture of UrlTest includes a variable that is never read or written by any of the tests defined. Furthermore the content of the fixture is shared but not correctly setup.

Besides, TestLint found a lot of *Magic Literals.* Again, we could regard them as false positives as literals can be expected for such a package. However as the use of them is extremely excessive in Network we would not regard the detection as such but rather as a global design flaw. Furthermore there are 6 possible *Mystery Guests*, which however can be identified as false positives.

Although TestLint cannot identify all the issues of Network due missing rules *e.g., Code Duplication*, its result fully agrees to the previous manual checking of Network.

# Chapter 6

# Conclusion

Unit testing is an important activity in a software project to assure the requirements and features of an aplication are properly reflected in the source-code. In the same way assessing the quality of tests is important to assure the quality and reliability of tests.

As we ascertained in our analysis, many problems emerge from Test Smells. Tests can become complex, hard-to-understand and hard-to-maintain. In the worst case the developer cannot rely on them any more due to their instability and obscure-ness, and don't use them any more. Therefore it is crucial to be able to automatically detect such design flaws early and refactor the tests.

In our approach towards solving those problems, we analyzed Test Smells at the code level, instead of describing them informally or formalizing them at a high abstract level. We formalized concrete rules and discovered and verified the need and significance of being able to detect Test Smells. Furthermore while conducting a large case study, we could actually prove that writing tests contributes to the writing of better tests.

## 6.1 Lessons Learned

- The automatic detection of Test Smells is important as design flaws in tests occur quite often and could have severe drawbacks on the evolution and understanding of an application.
- Writing a lot of tests enhances the quality of tests, leading to more robust and trustworthy tests which are simple to understand and better document the requirements and functionalities of the software.

- Dynamic analysis is a powerful technique which can considerably enhance the understanding of a software system and its tests. Understanding the runtime of an application, especially how the entities and features are interconnected can help finding flaws in the model design but also bugs in the code.
- Visualization of data is an important technique to compact and simplify complex problems as well as providing a specific focus on interesting or problematic locations on the application.It is a considerable help for understanding software. However finding the right visualization and focus is not trivial.

## 6.2 Future Work

In our research and implementation to assess the quality of tests we have primarily focused on Test Smells. A part of the future work will be an evaluation of Mutation Analysis. It will be interesting to know how useful and applicable this methodology is in dynamically typed languages like Smalltalk, and whether it adds new valuable information about a software system, especially about the quality of tests.

Furthermore we want to investigate whether there is connection between the quality of tests and the quality of the model. In particular we're interested whether writing tests and the quality of tests does affect the quality of the application code and design. Correlating the results of Lint and TestLint might give further insights into that question.

Using TestLint on well-designed tests produces a considerable number of false positives. We plan to formalize new rules trying to increase the precision of the rules to get more accurate results. Furthermore we will add several missing rules, for example to detect *Eager tests*, *Code Duplication* or *Building Blocks*. We want to enhance TestLint to become a valuable and indispensable part of the testing process.

We also want to focus on further enhancing Code Coverage to gather insight about the quality of tests as well as evaluating the usability and applicability of Partial Order and Delta Debugging (see the additional chapters in the Appendix).

The future goal is to propagate Quality-Driven Development by tightly integrating methodologies measuring the quality of tests into the development process. We want to achieve this by providing tools giving access to those methodologies while hiding the complexity, and visualizing the results in a simple and comprehensive way.

# Appendix A

# TestLint: Measuring Test Quality

## A.1 List of all detectable Test Smells

The following sections and tables present a list of all Test Smells extracted during our analysis. We notice that this list in not complete and that not all smells are implemented in TestLint. The Test Smells mentioned in our case-study in Figure 5.1 and Figure 5.2 are regarded to work reliable.

| **Test Smell** | **Description** |
|---|---|
| Mystery Guest | a test that might access an external resource |
| Code Duplication / Multiplication | copy-pasted code and data |
| Magic Literals | using many strings, symbols and numbers |
| Literal Pollution | high level literals used in multiple test-methods within a test-suite or between test-suites |
| Overreferencing | test creating unnecessary dependencies and causing duplication |
| Assertionless Test | pretending to assert data and functionality, but doesn't |
| Cascaded Assertions | assertions in cascades |
| Manual Test | a test that is not recognized as test by the Unit Testing Framework, but either contains assertions or executable code in the header comment |
| Long Test | tests including too many statements |
| Under-the-carpet Assertion | some assertions put into comments |
| Comments Only Test | all test-code put into comments |
| Over-commented Test | test having too many comments |
| Large Comments | comments that consist of text using multiple lines |
| Pseudo Test | a test that is an Assertionless and Manual Test |
| Max Instance Variables | large or oversized fixture |
| Likely ineffective Object-Comparison | objects comparisons which can never fail |
| Building Blocks | blocks of similar code and assertions within the test, separated by empty lines or comments |

Table A.1: Code Test Smells

| **Test Smell** | **Description** |
|---|---|
| Proper Organization | violating testing conventions by using bad organization of methods |
| Test-MethodCategory Name | method categories having a meaningless name |
| Test-Package Name | meaningless name for the package containing tests |
| Mixed Selectors | violating common organizational testing conventions by mixing up testing and non-testing methods |
| Unclassified MethodCategory | methods not being organized by any method-category |
| Empty MethodCategory | empty method categories |
| Empty Test-MethodCategory | empty testing method categories |

Table A.2: Organizational Test Smells

| **Test Smell** | **Description** |
|---|---|
| Under-the-carpet failing Assertion | failing assertions put into comments |
| Slow Test | a test that runs slowly |
| Erratic Test | a test with a (permanently) alternating test result |
| Transcripting Test | test writing and logging to the console |
| Morphic Test | test opening graphical windows or dialogs, maybe requesting input |
| Early Returning Test | test returning a value and too early, maybe dropping assertions |
| Returning Assertions | an assertion that returns |
| Control Logic | test controlling the execution flow by using methods like *debug* or *halt* |
| Guarded Test | conditional test including branches like ifTrue:aBlock or ifFalse:aBlock |
| Empty Test Suite | test class defining tests, but the suite is empty |
| Unused Shared-Fixture Variables | parts of the fixture that are never used |
| Empty Shared Fixture | a fixture that is explicitly declared but empty |
| Teardown Only Test | test-suite only defining teardown (unusual for unit tests) |
| Abnormal UTF-Use | test-suite overriding the default code-behavior of the unit testing framework |
| Empty Shared-Fixture | fixture defined, but empty |
| Hidden Shared Fixture | fixture variables are used/shared, but not explicitly defined in the shared fixture |

Table A.3: Behavioral Test Smells

| **Test Smell** | **Description** |
|---|---|
| Anonymous Test | test-methods having a meaningless name |
| Test-Class Name | test-class having meaningless name |
| Unusual Test Order | tests calling each other explicitly (unusual for unit tests) |

Table A.4: Inofficial Test Convention Test Smells

| Test Smell | Description |
|---|---|
| Eager Test | testing too much different (non-related) functionality of the model |
| Unused Test Behaviors | methods defined within the test-suite, but never sent during a test run |
| Empty Test Class | class defined but not providing any test (or methods) |
| Missing Tests | model behavior not covered by any tests |
| Insufficient Test Coverage | test not covering any possible execution of a method |
| Bad Test Data | using dummy input values that don't have any meaning in the context |

Table A.5: Other Test Smells of various categories

## A.2 Implementation

As mentioned in Section 3.2.1 the model of TestLint is based on the basic model entities *rule* and *node*. This also applies to the implementation of the model. Rules and nodes are each organized and implemented in a single inheritance class hierarchy and provide each a common superclass called Rule and Node. Besides, rules and nodes make heavy use of Traits [Duca06b, Scha02b] to share common behavior that could not be easily shared avoiding code duplication or using single inheritance.

Nodes are common Smalltalk entities, polymorphically encapsulated in an instance of a Node-class and provide and hide certain functionality. Nodes are interconnected using an abstract ancestry-relation. For example a Class-Node has the sub-nodes Method Categories and one super-node Class Category. The sub-nodes of a Method-node is an empty collection. Figure A.1 presents an selection of the different types of nodes available in TestLint. We notice that other nodes can by easily introduced at any time without the need of modifying the base-model.

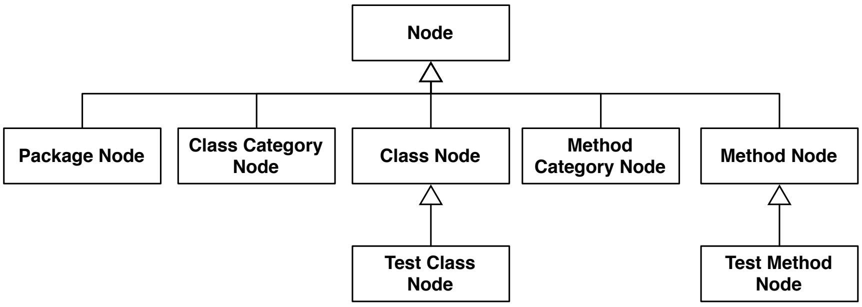


Figure A.1: Node-hierarchy used in TestLint

Rules are basically organized by the type of node they apply to respectively they accept as input. There exist multiple abstract classes to share common behavior to reduce code duplication. Figure A.2 shows the class hierarchy implemented for rules.

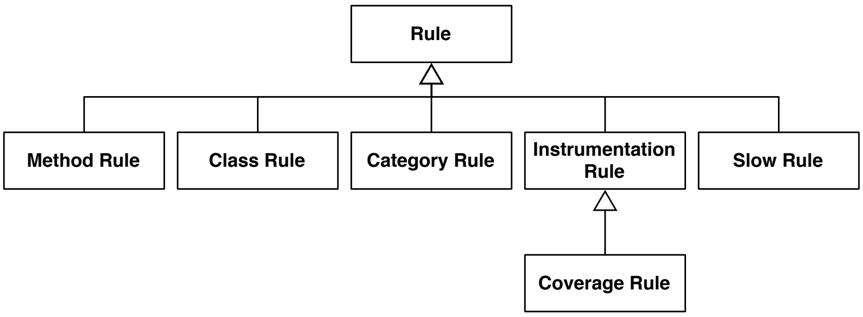


Figure A.2: A selection of abstract and predefined rule classes implemented in TestLint

We notice that organizing rules by the categories of Test Smells (Section 3.1.2) would conceptually be more natural and comprehensive. However that would considerably complicate the model and also be more difficult to realize as most smells can be associated to multiple categories.

Most rules provided with TestLint are dedicated to detect one fine-grained Test Smell. However rules share common detection-code with other nodes. Furthermore, the granularity and abstractness of the rules is not restricted. Therefore TestLint can be used to detect abstract and fine-grained Test Smells.

Each rule is dedicated to analyze one particular node. As explicitly modeling this connection would be complex, we use Traits to share the node-related code and make sure that each rule applies to the right kind of node. This keeps the model extremely lightweight and flexible. Figure A.3 visualizes the connection between rules and traits.

### A.2.1 Model: Synthesis of Rules and Nodes

The model of TestLint is based on *Double Dispatches* between the model entities *rule* and *node*. These double dispatches make the analysis of Test Smells completely transparent by hiding the fact that rules are dedicated to certain node-types. Due the dispatches and the use of node-specific traits we ensure that the right rule is applied on the right type of node without the need of explicitly specifying the relationship between nodes and rules.

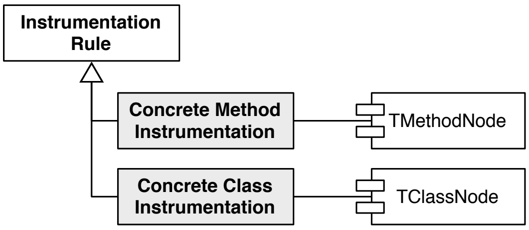

Instrumentation Rule
Concrete Method Instrumentation
Concrete Class Instrumentation
TMethodNode
TClassNode

Figure A.3: Depending on the target node, a rule requires a different base code which is provided by traits

We visualize the connection between rules and nodes as well as the basic functionality of the TestLint-model in the sequence diagram of Figure A.4. The diagram shows the dispatches when applying a Method-rule on a ClassNode.

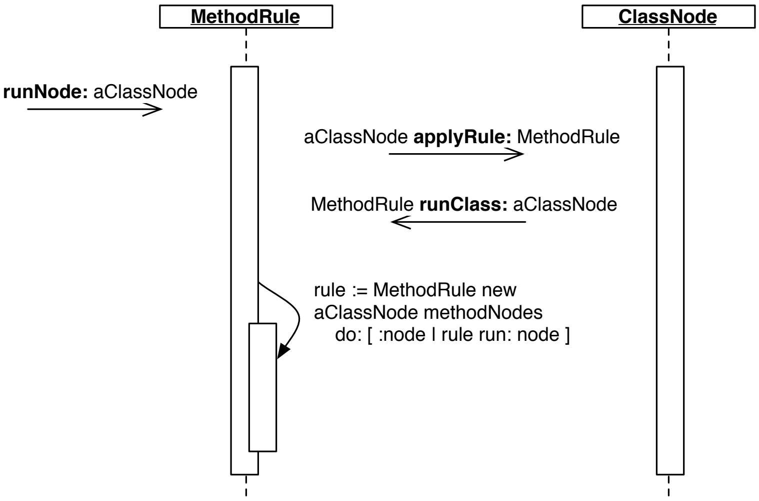

MethodRule
ClassNode
runNode: aClassNode
aClassNode applyRule: MethodRule
MethodRule runClass: aClassNode
rule := MethodRule new
aClassNode methodNodes
do: [ :node l rule run: node ]

Figure A.4: Double Dispatch between nodes and rules building the basic code model of TestLint

The resulting model is extremely loose coupled with high cohesion within each entity. This simplifies the extendibility and usability of TestLint. We can therefore reduce the public interface of TestLint to the following basic methods:

```
Node >> # applyRule: aRule
Rule >> # runNode: aNode
```

Moreover, the double dispatches enable a convenient handling of TestLint as the analysis can either be started from a node or from a rule. This simplifies the design of a user interface as the analysis is not dedicated to a specific entry point.

## A.3 User Interface

A good and easy-to-use interface [Shne98a, Smit82a, Coll95a, Coop95a] is crucial for doing efficient software analysis and testing. However, developing a good user interface that provides full functionality, hides all the details of the model beneath and is still easy and intuitive to use is difficult and time consuming. Furthermore, any kind of interface is subjective due different human expectations and perceptions.

As the primary goal in our research was to analyze and detect Test Smells we decided to make a little tradeoff in the development of the user interface by providing only very basic and standardized browsers and features without further regards to their usability and handling. Therefore TestLint consists of 3 slightly different browsers, each based on the Omnibrowser Framework [Putn, Berg07b].

- *TestLint SystemBrowser* integrates unit testing, debugging, profiling and Test Smell analysis
- *TestLint Package Browser* offers a way to browse detected Test Smells based on packages
- *TestLint Rule Browser* allows to browse tests by detected Test Smells

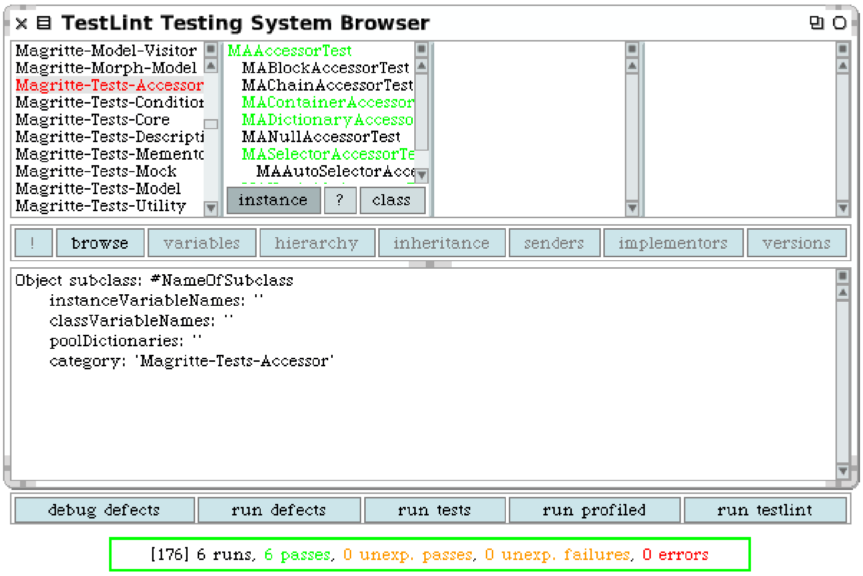
Each browser opens another perspective on tests and Test Smells and provides a reasonable amount of functionality, depending on the current scope. Furthermore they completely hide the details of the TestLint detection engine, making it easy to use TestLint without further knowledge of knowing how the model beneath works.

### A.3.1 TestLint System Browser

This browser is a modification of the Testing SystemBrowser of the *OmniTesting*[1] package. This package includes a default system browsers that

[1] OmniTesting is an open-source package on SqueakSource

provides rich testing actions based on SUnit and enhanced browsing capabilities between sources and tests. It tightly integrates testing and source-code development.



```
Object subclass: #NameOfSubclass
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Magritte-Tests-Accessor'
```

```
[176] 6 runs, 6 passes, 0 unexp. passes, 0 unexp. failures, 0 errors
```

Figure A.5: TestLint SystemBrowser providing unit-testing and quality-testing features at once

We decided to extend this browser with TestLint-specific actions because many developers already use Omnibrowser-based browsers, increasing the acceptance of our browsers and the ease of using it. Furthermore this browser already provides testing actions and fulfills our goals of integrating testing into the development process. That makes it ideal for TestLint actions as developers don't need to switch to a dedicated "tool" for testing or detecting Test Smells.

Detecting Test Smells is very simple and can be achieved by selecting a testing node[2] clicking on the "run testlint" button or selecting the action from the context menu. The action will display a small menu (Figure A.6) with a list of available rules.

The main-menu shows the different categories of rules, the sub-menu all the concrete rules. by clicking on any list entry will either run one rule, all the

[2] A testing node is a node containing or defining test-methods according to the underlaying testing framwork, for example SUnit

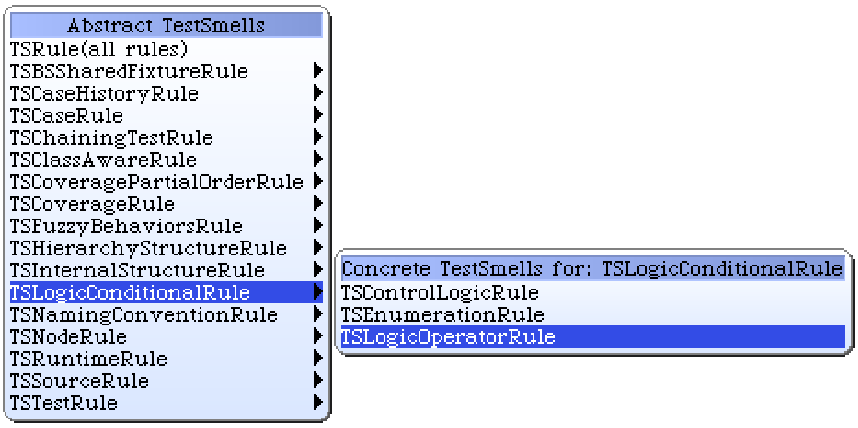


Figure A.6: Menu displaying available rules for detecting Test Smells

rules of the category or all available rules. A *TestLint Package Browser* opens the current context after the analysis.

We notice that the names of the rules (=names of the rule classes) used in the TestLint implementation slightly differ from the Test Smell-names used in our case study. This is a subject to change in future versions to simplify TestLint and synchronize it with the theory behind.

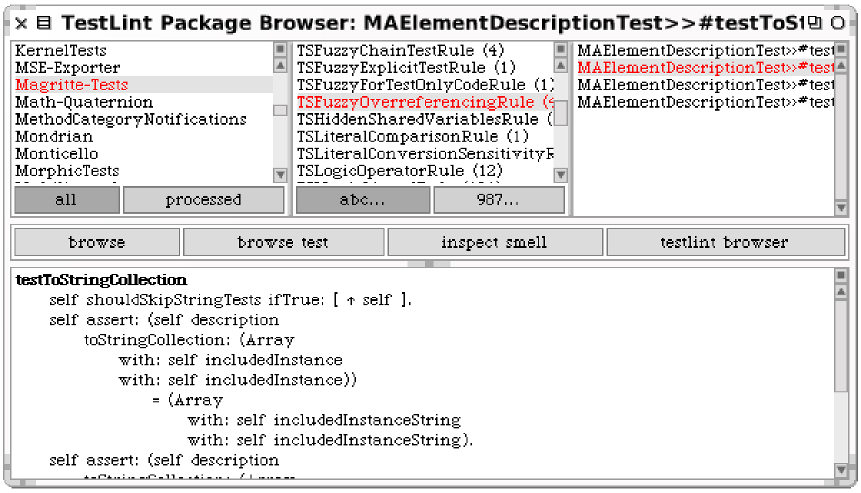
### A.3.2 TestLint Package Browser

This browser is partially based on the *OB Package Browser*[3] and dedicated to browse (only) packages defining tests and packages for which TestLint data is already available. It provides a selection of elementary actions for packages and actions to run TestLint rules on an entire package.

The result of each Test Smell or rule in a selected package is displayed in the second column and can be sorted alphabetically abc... or by the number of smells 987... found. Smelling nodes are shown in the third column including their default representation, for example source-code for a test-method, in the description panel below.

As some developers might be interested why Test Smells have been detected on a particular node, we provide a low-level inspector of the result or the data a rule collected. Furthermore the developer can annotate nodes as false positives to hide them to avoid further distractions from invalid results.

[3] OB Package Browser is an ope-source package on SqueakSource



TestLint Package Browser: MAElementDescriptionTest>>#testToS

KernelTests
MSE-Exporter
Magritte-Tests
Math-Quaternion
MethodCategoryNotifications
Mondrian
Monticello
MorphicTests

TSFuzzyChainTestRule (4)
TSFuzzyExplicitTestRule (1)
TSFuzzyForTestOnlyCodeRule (1)
TSFuzzyOverreferencingRule (
TSHiddenSharedVariablesRule (
TSLiteralComparisonRule (1)
TSLiteralConversionSensitivityR
TSLogicOperatorRule (12)

MAElementDescriptionTest>>#test
MAElementDescriptionTest>>#test
MAElementDescriptionTest>>#test
MAElementDescriptionTest>>#test

```
testToStringCollection
    self shouldSkipStringTests ifTrue: [ ↑ self ].
    self assert: (self description
        toStringCollection: (Array
            with: self includedInstance
            with: self includedInstance))
                = (Array
                    with: self includedInstanceString
                    with: self includedInstanceString).
    self assert: (self description
```

Figure A.7: TestLint Package Browser

### A.3.3 TestLint Rule Browser

This browser is basically identical to the *TestLint Package Browser* with the small difference that it scopes the entire Squeak image and not packages. Therefore it is reduced to two columns, displaying only rules and smelling nodes. Besides, it provides the same functionality and behavior.

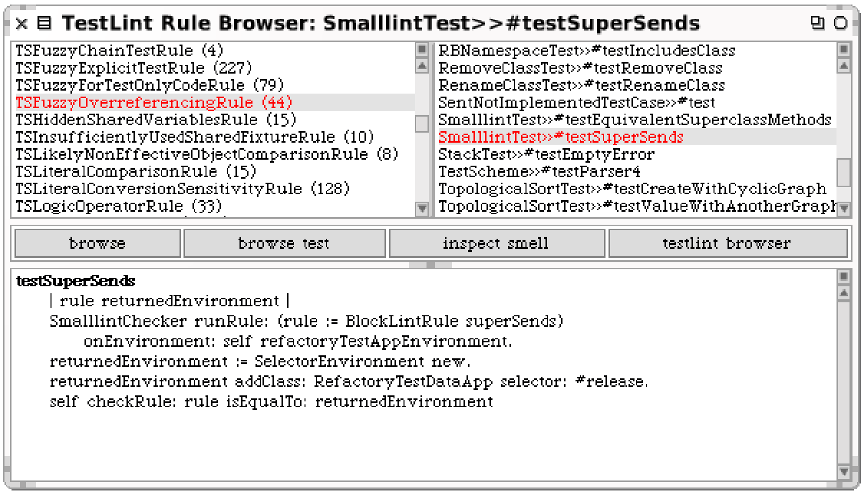
## A.4 How to install TestLint

To use the TestLint-browsers make sure you have exactly the following *OmniBrowser*-packages installed:

- *OB-Standard.39*, cwp.161
- *OmniBrowser.39*, cwp.318

It is recommended to have Christo, ByteSurgeon and Mondrian installed as certain rules and features of TestLint depend on those packages. However it is not required in general.

The simplest way to install TestLint, including all detection rules and browsers, is to use the *PackageLoader*. That will automatically load all working dependencies in the right order and setup the environment. The PackageLoader can be retrieved from the following repository:



TSFuzzyChainTestRule (4)
TSFuzzyExplicitTestRule (227)
TSFuzzyForTestOnlyCodeRule (79)
TSFuzzyOverreferencingRule (44)
TSHiddenSharedVariablesRule (15)
TSInsufficientlyUsedSharedFixtureRule (10)
TSLikelyNonEffectiveObjectComparisonRule (8)
TSLiteralComparisonRule (15)
TSLiteralConversionSensitivityRule (128)
TSLogicOperatorRule (33)

RBNamespaceTest>>#testIncludesClass
RemoveClassTest>>#testRemoveClass
RenameClassTest>>#testRenameClass
SentNotImplementedTestCase>>#test
SmalllintTest>>#testEquivalentSuperclassMethods
SmalllintTest>>#testSuperSends
StackTest>>#testEmptyError
TestScheme>>#testParser4
TopologicalSortTest>>#testCreateWithCyclicGraph
TopologicalSortTest>>#testValueWithAnotherGraph

```
testSuperSends
    | rule returnedEnvironment |
    SmalllintChecker runRule: (rule := BlockLintRule superSends)
        onEnvironment: self refactoryTestAppEnvironment.
    returnedEnvironment := SelectorEnvironment new.
    returnedEnvironment addClass: RefactoryTestDataApp selector: #release.
    self checkRule: rule isEqualTo: returnedEnvironment
```

Figure A.8: TestLint Rule Browser

```
MCHttpRepository
  location: 'http://www.squeaksource.com/PackageLoader'
  user: ''
  password: ''
```

Open a workspace and do: TestSmellsLoader new loadAll. During the installation you will be asked to specify whether or not to install optional packages. After the installation TestLint will run some setup-procedures. This might take some time and should not be interrupted.

# Appendix B

# Christo: Beyond Test Quantity

This chapter is structured as followed. Section B.1 gives a brief overview about the problems of traditional Code Coverage. In Section B.2 we enhance Code Coverage to provide a developer with deeper and more comprehensive insights into the system. We give examples of such enhancements in Section B.2.1, Section B.2.2 and Section B.2.3.

## B.1 Introduction

In Section 2.1 we described the traditional principles of Code Coverage and gave two representative examples of well-known and very sophisticated tools to gather coverage data. We determine that the traditional analysis lacks several problems:

First, the *coverage visualization* is poor and lacks of focus towards testing and maintenance. For example methods having no coverage and methods having full coverage, are mixed up and displayed the same way. Furthermore, the associations of method names to numbers and percentages are very abstract and difficult to compare, especially if they appear in large quantities like in Figure 2.2. Such a result is hard and time consuming to embrace and therefore not really suitable.

Second, the *results of the analysis* are incomplete or reduced too much. They're missing the links between sources and their tests. For example it is not possible to find out which method or which statements of the method gets executed by a particular test. However this information is crucial for the development and testing cycle to complete tests to ensure high coverage of the methods under test.

Finally, the dynamically collected data is not *re-used* or further manipulated to help the developer in the testing and development process.

The consequence of these problems are that the coverage analysis loses its value as relevant and crucial results and conclusions are hard or impossible to extract. Furthermore as dynamic analysis requires a higher computational effort, but is not sufficiently used, it becomes a very time and cost intensive analysis. Therefore we even claim that the traditional analysis is a waste of computational power and development and testing time.

We fix these issues by enhancing the traditional approach of Code Coverage with several simple methodologies and ideas which we explain in the following sections:

- Increasing the amount of data collected (Section B.2) during the dynamic analysis, establishing dynamic links between sources and their tests.
- Using the results to provide more insights into the tests and sources. We demonstrate this with Partial Ordering of tests (Section B.2.1) and a simplification of Delta Debugging (Section B.2.2).
- Displaying and visualizing coverage results in a more perceivable and natural way to support the testing process (Section B.2.3).

We are convinced that by enhancing Code Coverage and adapting it to the real needs of Quality-Driven Development we can express more and contribute high level quality feedback to further support developers with crucial information about their application code and tests.

## B.2 Enhancing Code Coverage

The problem of traditional Code Coverage is that it mostly annotates methods or statements as *executed* or *not executed*. We demonstrate this in the following instrumentation example.

```
CodeInstrumenter >> #instrumentMethod: aMethod
  aMethod insertBefore: [ self markExecuted ]

CodeInstrumenter >> #instrumentStatement: aStatement
  aStatement insertAfter: [ self markExecuted ]
```

Although this might be enough to get a rough impression about the covered code it is not enough for a thorough test-driven development. Using the traditional coverage information we are not able to draw any conclusions about how much, which methods and what parts of a method a single test is covering. However this information is important to know to adapt tests to raise coverage where it is mostly required or to add new tests covering new or other yet-untested features of a method. We therefore instrument all source methods in the scope of the dynamic analysis with more sophisticated code to gather more information about the runtime.

The most basic but important extension is to gather the *currently executing test* for each method, statement or node of the abstract syntax tree. This information is responsible to establish the connections between the methods and their tests – or the other way round. The following pseudo-code illustrates this idea, again using method-coverage:

```
CodeInstrumenter >> #instrumentMethod: aMethod
  aMethod insertBefore: [ self coverage append: self system currentTest ]
```

The consequence of this small change is a mapping of source-methods to test-methods. Therefore we can query the system to find out which methods are covered by which test, and the other way round. The same also works for sub-method coverage.

If required, we can further enhance this by also collecting information about the *senders* of a method, its *receivers* or even collect the method's *context* or *argument values* passed to it. The more we collect the more information we get from the runtime, the more thorough but also complex are the results and the conclusions we can draw from them.

We notice that when collecting senders, receivers, but especially objects and values, it might be more appropriate to use a *Tracing*-technique [Lien06a, Kuhn06d, Gree05b] as this one is more suitable for gathering this kind and amount of information.

As the basis of our Code Coverage analysis we use various commonly known code instrumentation methodologies which enable dynamic analysis:

- *MethodWrappers* [Bran98a, Denk06a] are a well known and common technique to instrument methods and tests to gather dynamic runtime data, for example Code Coverage. A MethodWrapper basically installs a piece of source-code in front or after the method-source and recompiles it to activate the additional code. This code can then be used to execute additional functionality or gather information about the runtime.

- *ByteSurgeon* [Denk06a] is a flexible framework to enable on-the-fly bytecode transformation at runtime. It is doing something similar to MethodWrappers. However the instrumentation works on a lower level, transparently transforming normal source-code to byte-code and adding it to the existing byte-codes of a method. The instrumentation normally runs faster and does not need to recompile the whole method after adding and removing code.

- *JMethods/Persephone* [Mars06a] introduces methods as a first-class and high level abstraction. It provides rich structural reflection at sub-method level, for example to ease meta-programming. We can use JMethods/Persephone for sub-method coverage by annotating the nodes of the abstract syntax tree of a method.

### B.2.1 Partial Ordering of Tests

Using the extended coverage information gathered at runtime we are able to partially sort and order tests [Gael03b, Gael04b] based on the comparison of their coverage-sets[1]. A unit tests *covers* another unit test, if the coverage set of methods invoked by the first test is a superset of the coverage set of the second. We illustrate this in Figure B.1.

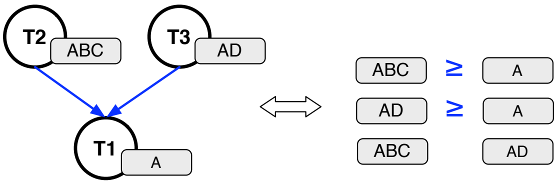


Figure B.1: Schematics of Partial Order relations (dark-blue) between the tests T1, T2 and T3. The coverage-sets of each test is colored in gray and contains letters which represent methods

The Partial Order of tests give various insights into and hints about an application and its tests:

- It can help in comprehending the features and requirements of an (completely unknown) application by systematically reading the tests, for example starting with the most abstract ones, then steadily diving deeper into the lower tests and more details of the system.
- The ordering of tests show the relations between the tests and the methods they're covering. The resulting sub-graphs of the ordering might even be used to identify groups of *features* whereas a feature would be set of methods which belongs together. This can help in understanding the functionality of an application.

[1] A coverage-set is either a collection of methods which are executed by a test or a collection of test-methods executing a particular source-method

- Partial Order might also be used to minimize the effort spent in testing and debugging (Section B.2.2), by prioritizing a set of tests most likely covering the sources containing the defects [Gael04b, Gael06b, Roth01a, Elba00a].

We visualize Partial Order relations between tests in Figure B.2 using a graph-layout and interconnecting the tests with appropriate relations.

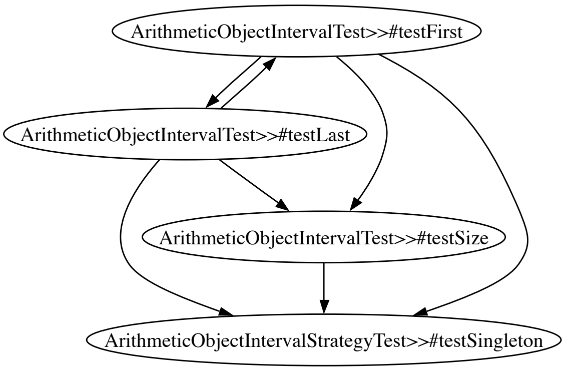

ArithmeticObjectIntervalTest>>#testFirst
ArithmeticObjectIntervalTest>>#testLast
ArithmeticObjectIntervalTest>>#testSize
ArithmeticObjectIntervalStrategyTest>>#testSingleton

Figure B.2: Partial Order on a small set of tests taken from the package Aconcagua

We notice that the tests testFirst and testLast in Figure B.2 have a special relation - there are *equivalent*. This means that they have the identical coverage set and therefore covering the same functionality. However the tests themselves are likely not identically as the data being tested or the execution flow during the runtime might vary.

### B.2.2 Delta Debugging

With the evolution of a software project the amount of source-code written and the complexity of a system is raising. Writing and maintaining tests [Beck03, Beck98a] can help to maintain the source-code, minimize refactoring efforts and minimize the probability of severe failures and errors unexpectedly [Zell99a, Zell05b] appearing in the development and deployment process.

However bugs have many possible causes [Zell05a] and can never be completely removed, nor their absence be proven. Debuggers are often the only change to effectively identify bugs and very often used during the development cycle. However as bugs seldom cause an exception on their own the debugger jumps over the buggy code directly to the location where the exception occurs, sometimes hiding the methods of interest.

*Delta Debugging* [Zell02a, Zell02b] is an automatic and systematic technique to automatically debug an application to reduce the effort of manually debugging an application and to reduce the set of failure inducing sources to a minimum. Delta Debugging can be used for various debugging purposes like checking data boundaries using various good and malicious data-files and comparing the results from the test runs or testing source-code behavior by calculating the difference between multiple revisions. Other applications are possible.

Based on our dynamic analysis with Christo we implemented a highly simplified and distant kind of Delta Debugging to demonstrate the capabilities of using dynamic information to reduce the possibilities of failure inducing sources. We implemented various simple and more complex strategies for our experimentation and to precise the results. We give two examples in the following.

We notice that there are many other or ways to realize techniques to automatically find the failure inducing sources. For instance we believe that tracing technologies gathering more dynamic data at runtime as well as and graph-matching techniques applied to the traces or partial-order graphs could be used to enhance *Delta Debugging* .

### Strategy 1: Diffing Coverage-Sets

This strategy is based on the previous calculation of the Partial Order of tests and is optimistically reducing the set of failure inducing methods.

Using the ordering of tests we iteratively calculate the differences between the failing test under analysis and the tests it is covering. The result of those differences are sets of methods not being covered by another test. We assume that those methods are most likely to contain the failure inducing sources because no test is checking them. We collect each source-method within those sets by its number of appearance and sort them in descending order. We show this in Figure B.3.

In the example of Figure B.3 we calculate the differences between the failing tests (`T3`) and the tests it is covering (`T2`, `T1`). We sort the results by the source-methods that is most often appearing in the differences (`C`), weakly assuming it is more likely to fail.

We determine that we cannot guarantee that this strategy is able to find the failure inducing source. That's because the calculation of the differences assumes the functional correctness of the sources being covered by the succeeding tests. However tests can never prove the absence of a bug nor the

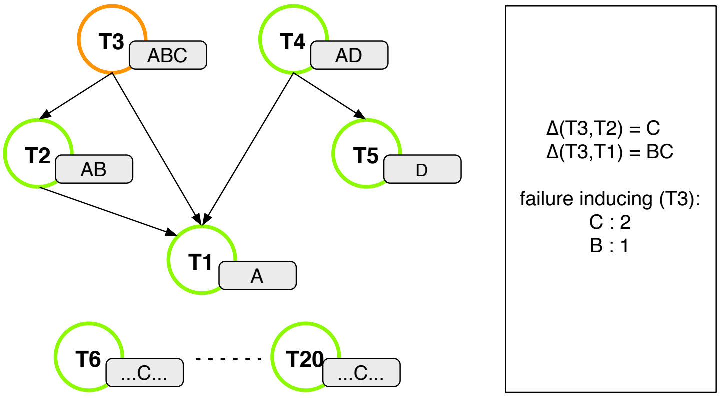


Figure B.3: Diffing Coverage-Sets Strategy

correctness of the covered source functionality. Therefore this strategy is actually based on a wrong assumption. As a consequence it is more aggressively reducing the number of failure-inducing possibilities, maybe also removing the actual bug-method. On the other hand, assuming the covered functionality really behaves correctly it likely delivers a very precise result.

We demonstrate this with the example from above. We assume the same test configuration and result. We introduce a nasty a bug in A which does not cause test T1 and T2 to fail, but T3. We further assume that C is correctly implemented. As the strategy assumes complete coverage and correctness of A and B it concludes the failure in C. Therefore the strategy would not be able to find the actual bug.

### Strategy 2: Example-Methods

The strategy of *Example Methods* is a pessimistic approach of detecting the failure-inducing sources. It eliminates the wrong assumption of the *Diffing Coverage-Sets* strategy and assumes that every method is likely to have a failure, independent of its coverage value or test results.

For this strategy we calculate the probability of a source-method inducing a failure or error. This probability results by enumerating the *good* and *bad examples* of covered source-methods within the current context whereas the *good examples* are the succeeding tests and the *bad examples* are failing tests or tests raising an error. We illustrate that in Figure B.4.

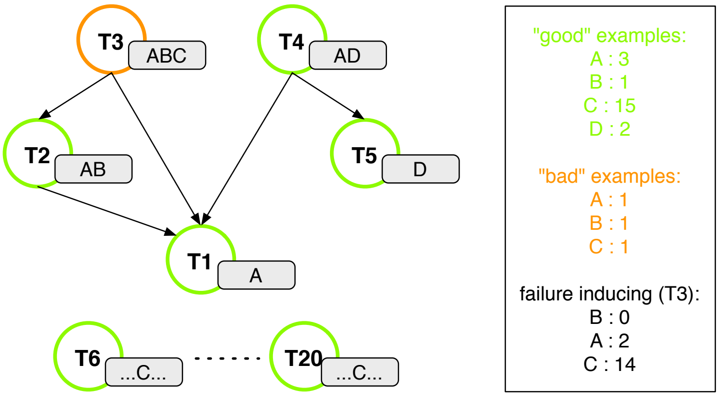


Figure B.4: Example-Methods Strategy

In the example of Figure B.4 we calculate the examples for each source-methods in the context. For example A is covered by 3 successful tests and one failing one, leading to 3 good and 1 bad example.

The advantage of this strategy is that it runs extremely fast as neither Partial Order nor complex and time-consuming set operations have to be done. Furthermore it doesn't depend on the partial order and respects the context as more than just the tests within a subgraph of the Partial Order-graph are taken for the calculation.

### B.2.3 Visualization

The comprehension of rich information, complex problems and their coherence depends a lot of human perception. Therefore a good visualization is an important and inevitable technique of presenting and communicating rich and complex information by focusing on and revealing the interesting parts of the underlaying data without the need of introspecting a large amount of raw data. Visualization in computer science is often used for understanding software architectures [Eden01a, Lang05a], reverse engineering [Duca99s] or evolution of software [Girb06a, Girb04a] and data.

In Section 2.1 we notice that the presentation of Code Coverage data in todays implementations is poor. Most approaches use large tables filled with numbers, meaningless coloring, sorting the methods and their coverage results alphabethically, or miss the focus towards the relevant source and test parts. Others reduce the data in such an extent that no value of the coverage analysis remains.

In the following we present several example visualizations provided by Christo based on collecting dynamic runtime information, in particular coverage, in a very compact and comprehensive way providing a much better focus on tests and coverage than numbers would do, and without loosing or reducing the amount of data colected. We use Mondrian [Meye06a, Meye06a] and GraphViz for our visualizations.

### Coverage Map

The purpose of the *Coverage Map* is to get an overview of the coverage of a collection of source-nodes without being cluttered with a large amount of numbers. It is supposed to communicate the most relevant information about coverage and tests at thee first glance.

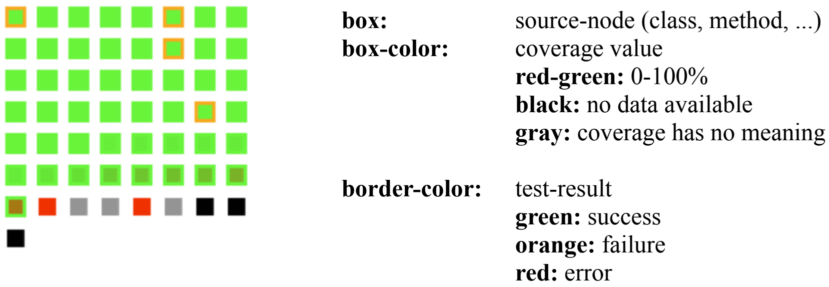


Figure B.5: The Coverage-Map of the package Aconcagua based on a selection of source-classes

This visualization is directly based on the raw coverage data. We display a number of nodes, for examples source-classes or source-methods, represented as small squared boxes sorted in descending order by the amount of coverage.

The coverage value, normally given as a number in percent, is a color of a heat-map whereas green represents 100% and red 0% coverage. All nodes within the scope of the visualization which don't have any coverage information available yet are black. Nodes for which coverage would have no meaning, for example methods only declaring self subclassResponsibility are displayed in gray.

Furthermore we draw a colored border around each box, displaying the accumulated result of the tests covering that particular node. A green border means that all tests covering that node return a success. An orange or red border represents at least one failure or one error within the covering tests. As we know the link between sources and tests, we can quickly browse the tests of a node by clicking on the node and opening it in a code browser.

The advantage of this visualization is that it contains the most important information at the first glance while using only little space. The interesting parts, for example methods having no coverage, not enough or failing or error tests can be identified immediately without harvesting large reports containing numbers (Figure 2.2, Figure 2.1). Furthermore due the interactivity of Mondrian we are able introspect each node in an agile way and open further visualizations, giving more details, or open the selected node in a code or class browser.

### Test Complexity

To gather insights about the tests and their quality it is not enough to visualize the coverage of the sources. It is sometimes interesting and important to know what kind of tests we look at, how complex, abstract or eager[2] they are, or what and how many sources they cover. The correlation of this information can help in comprehending the tests, identifying special or abnormal tests and finally improving the quality of the test-suite.

We visualize this using a scatter-plot technique. Each test-method is represented by a squared box in a coordinate system with the origin in the upper left corner. The color of the box stands for the size of the coverage set, and therefore for the abstractness of a test. A dark color means the test is covering a large amount of functionality whereas white can be interpreted as a primitive unit test. The x-axis measures the amount of statements and the y-axis the number of assertions.

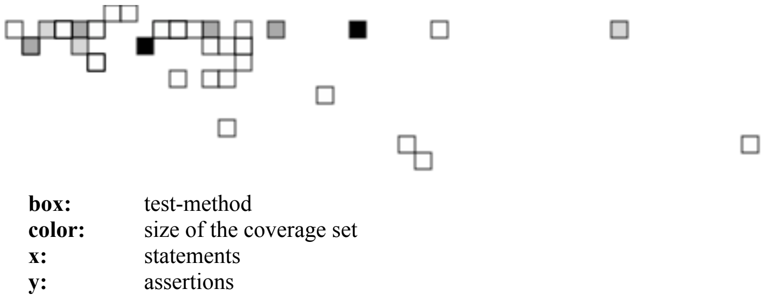


Figure B.6: Test Complexity taken from a test-class of the package Aconcagua

[2] A test is said to be eager if it tests a lot of code or different features. Following the methodology of Test Smells, eager tests are "bad" tests and should be avoided

We identify 2 tests (dark boxes), covering a lot more functionality than any the other test in the test-suite. The one in the upper middle probably has the double amount of statements than the one on upper left side. Both have only a very few assertions. We can assume that those tests are checking high level functionality. We also detect a very interesting test (bottom right corner) which covers only a very little functionality but seems to be a test including a lot of statements and assertions. That tests probably verifies a lot of data and data boundaries or includes lot of code which does not belong to the context of its package.

### Coverage Relevance

As we calculate the coverage of a selection of sources, we might discover some that are not covered at all or not sufficiently enough. In such cases it would be interesting to get a pinpoint to the sources which would be most relevant or important to have tests for, respectively to have coverage information available. Especially for applications in which test-coverage is crucial this hint could prove helpful to resolutely develop further tests.

We try to achieve this by correlating the *size* of a source-method to its *senders* within the current context, for example the package under analysis. We assume that the larger a source is the more complex it is and therefore more susceptive to hidden bugs or missing requirements. Furthermore the more often a source-method is used the more features depend on it and the more important it is that this method correctly fulfills the requirements of a software system. Therefore, we conclude that it is most relevant to have tests for such a method or feature.

We visualize this in Figure B.7. Each source-method which is not covered by any test within the context is represented by a rectangular box. The color stands for the usage of that method, in particular the number of different senders. The width and height of a box represent the usage and size of that method. As a normalization for the color and the size of a method we use the entire context of the analysis. In the example of Figure B.7 the height can be interpreted as the number of different methods called and the width as the number of statements.

We detect a large number of extremely small and medium-sized methods. Furthermore we notice several rather big ones. Whereas most methods are not often sent we detect 5 exceptional methods (black boxes) which are heavily used within that package. Those methods likely fulfill an important task.

Using this knowledge we can very easily and quickly detect and verify for which source-methods it would be most appropriate to write the next tests

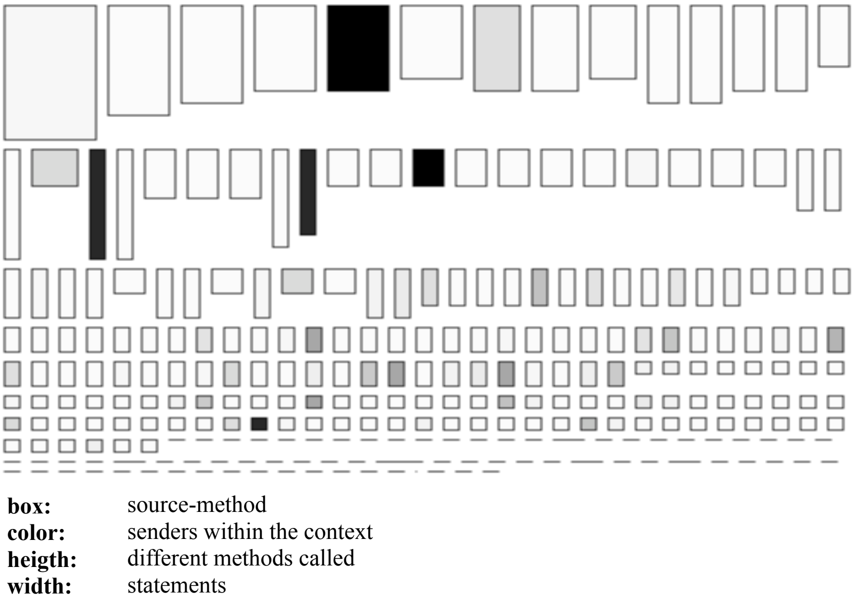


Figure B.7: Coverage Relevance on non covered or insufficiently covered methods in the package Aconcagua

for to cover the most important sources and functionality with tests. In our example we might first have a look at the sources represented by the black and darker boxes, as well as the very large ones.

## B.3 Implementation

### B.3.1 Model

The model of Christo Code Coverage is based on the following principles:

- *Simplicity.* The public user interface of the model should be as simple and small as possible and should not require any knowledge about dynamic analysis or Code Coverage.
- *Safety.* Code Coverage should be safe, for example using *MethodWrappers* on *unsafe* elements should not crash the image.
- *Transparency.* Christo should be able to work using various different technologies to gather dynamic data. Furthermore the developer

should not necessarily need to know how they work to enable Code Coverage.

We realize this using a 3-layer architecture (Figure B.8). The first layer consists of the *technology*, providing *Adaptors* to hide the technology specific features. The second layer is the *security*, making sure Christo does not execute any actions which could result into an unstable system. The last layer is the *public user interface* which gives a unified access to Code Coverage through *Nodes* and *Sets*.

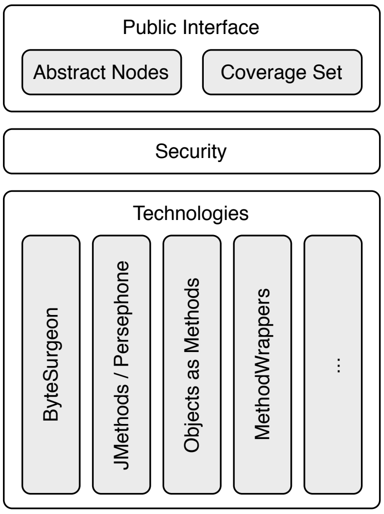


Figure B.8: Overview of the Christo Code Coverage model

The advantage of this model is that each layer can be replaced or extended with additional functionality without the need of adapting or rewriting all the other components. For example it is fairly easy to add new technologies as they don't affect the rest of the implementation. Furthermore the model has a loose coupling between and high cohesion within the components. This makes it very simple to replace certain features without affecting others.

On the other hand this model also has one big disadvantage. Due high level abstractions, many small objects are involved in the dynamic analysis, making it slower than a hard-coded one-method-coverage approach. However, the performance primarily depends on the number of methods being called during the runtime. The more method calls the slower it is compared to a less abstract implementation as the gathering of coverage on a high-level requires additional method calls and creation of objects.

We detail the model of Christo in the UML of Figure B.9, describe each entity and briefly explain how they interact.

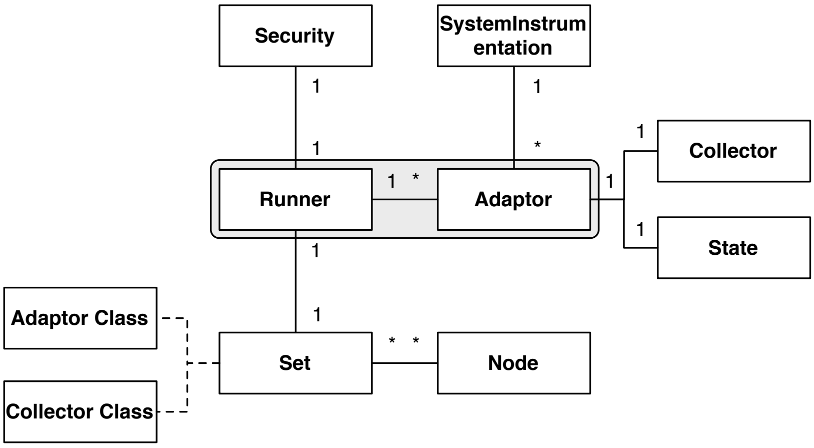


Figure B.9: Model of Christo Code Coverage

**Node.** A node is any kind of an abstract Smalltalk node, for example a package, class or method. A node is either a source- or a test-node[3], or contains such. It provides a minimal interface to query parent- and child-nodes as well as sources and tests.

**Set.** A set is an collection of nodes. It must contain source- and test-nodes to make the analysis meaningful. A set also knows which technology (adaptor) to use and what and how much data to collect (collector).

**Adaptor.** The adaptor is the interface to the underneath technology used to gather coverage and should only be used internally. It encapsulates all the technology-dedicated functionality and features and offers only a minimal interface which basically consists of the methods *install* and *uninstall.*

**Runner.** The runner is the runtime that accepts a set as input and knows how to prepare the nodes, adaptors or the system to enable it for gathering coverage. It executes the tests and ensures the system is put back into the state is was before the analysis started. A standard routine might look like: install, run tests, uninstall.

[3] The underlaying Unit Testing Framework decides what a test-node is, for example in SUnit a class inheriting from the class TestCase is a test-node

**Collector.** The collector is an instance that knows which and how much data to collect, and how and where to store it. A collector becomes active whenever a method, statement or sub-method node[4] is called – depending on the technology/adaptor being used. The default collector collects the test in execution and how often it calls the current method or node.

**State.** Each adaptor has a state declaring the condition of the node's coverage, for example *adaptor installed*, *adaptor ready*, *adaptor failed installation* and so on.

**SystemInstrumentation.** This is a cache for the adaptors that can swap in/out the original and instrumented method-code to speed up the instrumentation and un-instrumentation process.

**Security.** This is a component called by the runner to decide whether or not it is safe to instrument a particular node. For example the security component would reject the installation of a *MethodWrapper* on any method of the class Array as this is likely to cause problems.

### B.3.2 User Interface

The user interface of Christo is dedicated to efficiently configure and browse coverage and Partial Order as well as visualizing various interesting aspects of tests Section B.2.3, including coverage and Partial Order.

Christo consists of three basic browsers, each providing a specific focus on one or multiple features and functionalities. Besides, all browsers are interconnected among each other and are accessible through the main-menu of Squeak. To gain a higher acceptance for Christo and to keep it as simple as possible we used the Omnibrowser-Framework [Putn, Berg07b] to model the browsers.

We briefly describe each browser and its functionality in the following sections.

[4] A sub-method node is a node in the Abstract Syntax Tree of a method

### System Coverage Browser

This is the main-browser of Christo and provides the following features:

**Configuration.** A developer can set-up various *Coverage sets*. Each set contains a selection of sources and tests which have to be specified by the developer. This can be done by drag&drop-ing nodes from any browser displaying system-nodes like classes, methods and so on. Besides, the configuration has knowledge about the technology (*e.g.,* ByteSurgeon, Persephone) used to collect coverage with and the granularity or amount of information to be gathered. In general Christo is making the best choice automatically, but the developer has the possibility to make other specifications.

Actions on a *Coverage Set*: new, rename, remove, merge, choose adaptor, choose collector, inspect, coverage result

**Running Coverage.** The developer has the choice between running the full analysis, regenerating the obsolete[5] results or updating the missing[6] values. The later two options can considerably speed up the analysis process by optimizing the set and runtime based on the last result.

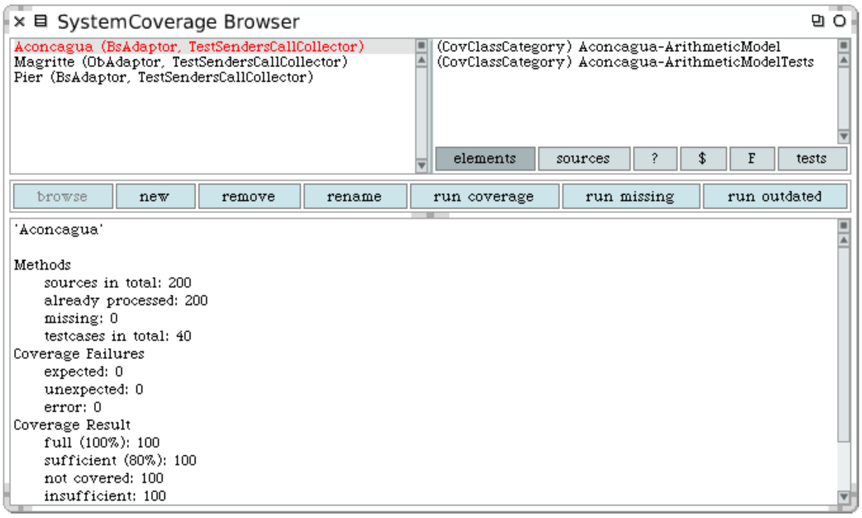
Runtime Actions : run coverage, run obsolete, run missing

**Browsing Configuration.** The browser provides basic insights into the coverage set. First, the lower panel displays a coverage overview of the currently selected element, for example showing the coverage value in percent and the number of source- and test-methods being processed. Second the upper right panel shows the contents and status of the set. You can browse the abstract nodes elements and the concrete source- and test-methods they contain. Furthermore, it provides access to specifically browse the obsolete nodes ?, nodes insufficiently covered $ and unexpected failures produced during the last analysis on that set.

As the System Coverage Browser focuses on configuring coverage it is ideal to setup a new coverage configuration, declare the technology used or to specify the kind and amount of data collected. It is also suitable to inspect failures, search for badly covered nodes or obsolete coverage. Moreover, it provides access to the caches used to store information, for example to

[5] A coverage result of a node, for example a method, becomes obsolete if its source-code changes.

[6] When adding new nodes to an already processed set, their coverage is normally missing at that time.



```
Aconcagua (BsAdaptor, TestSendersCallCollector)
Magritte (ObAdaptor, TestSendersCallCollector)
Pier (BsAdaptor, TestSendersCallCollector)
```

```
(CovClassCategory) Aconcagua-ArithmeticModel
(CovClassCategory) Aconcagua-ArithmeticModelTests
```

```
'Aconcagua'

Methods
     sources in total: 200
     already processed: 200
     missing: 0
     testcases in total: 40
Coverage Failures
     expected: 0
     unexpected: 0
     error: 0
Coverage Result
     full (100%): 100
     sufficient (80%): 100
     not covered: 100
     insufficient: 100
```

Figure B.10: System Coverage Browser having multiple configurations setup

remove the results of the current set or to wipe-out everything. This helps to keep the image size low and Squeak responsive.

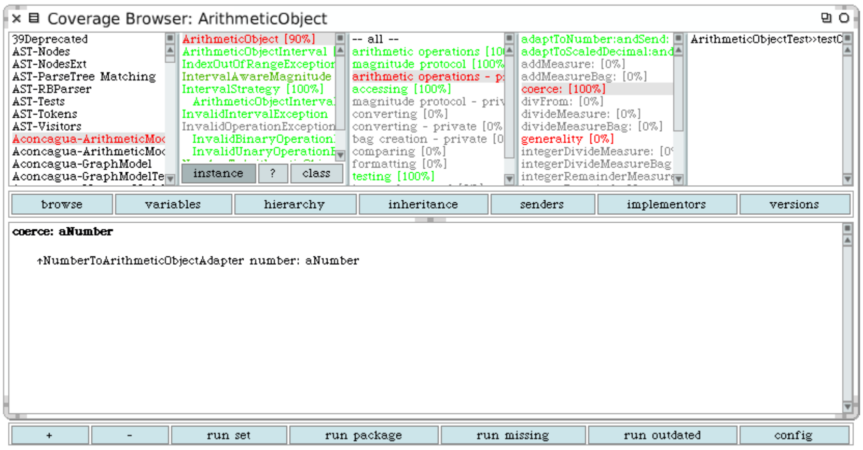
### Coverage Browser

The Coverage Browser is a normal *System Browser* with the focus on coverage by annotating each node with the traditional coverage value as a number in percent and a more intuitive color representing this value. The colors are taken from a heat-map between green and red whereas green represents 100% and red 0% coverage. Black is used for nodes having no coverage available and gray for nodes which coverage would have no meaning.

A fifth panel shows up when selecting on a source-method, displaying all the tests executing that method. Selecting a test-method will show all source-methods and their classes being executed during that test.

Besides, the browser provides the same features like a normal *System Browser*, but extending them with coverage-related actions. The most important actions are placed in a button-panel at the bottom of the browser. It provide the following set of features:

**Quick Configuration.** Using the buttons + and - you can very quickly add or remove the currently selected node from a coverage configuration



```
coerce: aNumber

    ↑NumberToArithmeticObjectAdapter number: aNumber
```

Figure B.11: Coverage Browser showing the coverage of a selection of the package Aconcagua

without opening the System Coverage Browser. A list of available sets pops-up on clicking on either of those buttons. The button **config** on the right-most of the panel opens the System Coverage Browser.

**Running Coverage.** The browser provides multiple buttons to run a specific set **run set**, the currently selected package **run package** or to update the coverage using **run missing** or **run outdated**.

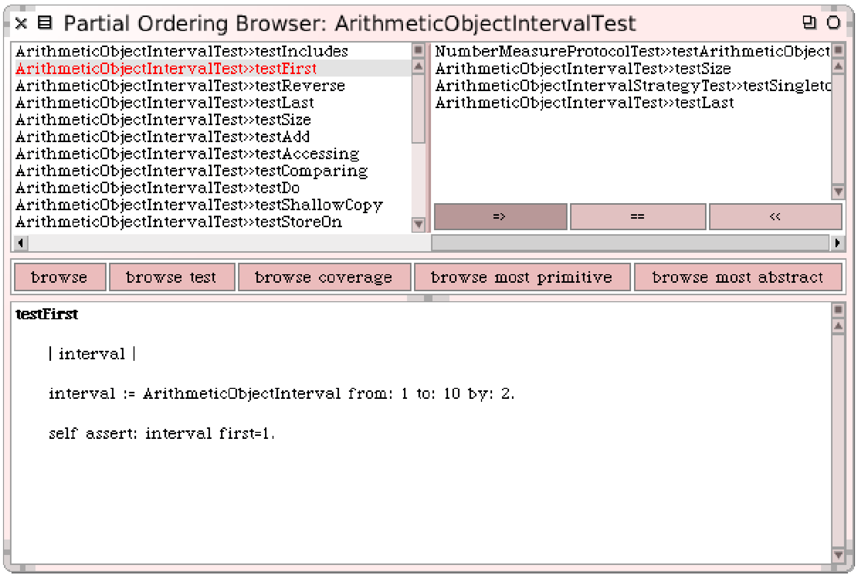
Furthermore, the Coverage Browser provides a lot of other actions through the contextual menu, in particular access to browse and visualize *Partial Order* and various visualizations on sources and tests using GraphViz and Mondrian.

We notice that Christo also has a Coverage Browser called Package Coverage Browser that starts at the package level and not class-categories. Besides, it provides the absolutely identical features and functionality.

### Partial Order Browser

The Partial Order Browser is the entry point for browsing and visualizing the Partial Order of Tests and can be opened through the Coverage Browser or the Squeak main-menu. It is based on the typical look of any Omnibrowser and features a standard browsing-panel (top) the nodes in the graph and a text-panel for displaying the content of the currently selected node.

Although a browser for browsing a Partial Order-graph on tests seems to be quite inconvenient, it does have some advantages over a Mondrian or GraphViz-Visualization. First, it can display a lot of concrete information at once, for example class and method names as well as the code of the test. Second the browser is very compact and you can browse and search more determined as it provides a very specific focus on a small selection of the graph. On the other hand the total overview is missing. However this can be compensated by the provided visualizations.



Partial Ordering Browser: ArithmeticObjectIntervalTest

```
ArithmeticObjectIntervalTest>>testIncludes
ArithmeticObjectIntervalTest>>testFirst
ArithmeticObjectIntervalTest>>testReverse
ArithmeticObjectIntervalTest>>testLast
ArithmeticObjectIntervalTest>>testSize
ArithmeticObjectIntervalTest>>testAdd
ArithmeticObjectIntervalTest>>testAccessing
ArithmeticObjectIntervalTest>>testComparing
ArithmeticObjectIntervalTest>>testDo
ArithmeticObjectIntervalTest>>testShallowCopy
ArithmeticObjectIntervalTest>>testStoreOn
```

```
NumberMeasureProtocolTest>>testArithmeticObject
ArithmeticObjectIntervalTest>>testSize
ArithmeticObjectIntervalStrategyTest>>testSingleto
ArithmeticObjectIntervalTest>>testLast
```

```
testFirst

    | interval |

    interval := ArithmeticObjectInterval from: 1 to: 10 by: 2.

    self assert: interval first=1.
```

Figure B.12: Partial Order Browser showing the Partial Order of tests on a selected test of the package Aconcagua

The Partial Order graph can be browsed by selecting one of the directional buttons >=, << and == and clicking on a node to go into that direction in the graph. This will open another panel to the right side of the selected node showing the nodes of going into that direction in the graph. The direction-buttons have the following meanings:

- => will go downwards in the graph, showing all the tests being covered by or equivalent to the selected test
- << will go upwards in the graph, showing all the tests that cover the selected test
- == will show all the tests that are functional equivalent to the selected test

We notice that depending on the graph and using the directions => and << you could browse in cycles.

There is wide range of actions available for each selected tests. We can visualize the graph using *Mondrian* or export the graph-data into a file to visualize it using *GraphViz*. There are also options available to browse the most or least abstract tests within the current selection or package. The most abstract test is a test which is not covered by any other test, the least abstract is a test which is not covering any other tests. This is in particular useful to browse the *boundary* of a graph. Furthermore each selection can be browsed using a dedicated browser for coding (standard SystemBrowser), coverage (Coverage Browser ) or testing (Testing Browser).

We mention that there is also a Partial Order Class Browser available with the focus and Partial Order-relations on test-classes instead on test-methods. However this browser is only suitable for very large packages including lots of tests.

## B.4 How to install Christo

To use Christo you require a Squeak-3.9 image, release build 7067 at least. If you have another release build, make sure you have exactly the following *OmniBrowser*-packages installed:

- *OB-Standard.39*, cwp.161
- *OmniBrowser.39*, cwp.318

Due the complexity of dependencies it is recommended do load Christo using the *PackageLoader*. This one will automatically load all working and optional dependencies in the right order. Load it from the following repository:

```
MCHttpRepository
  location: 'http://www.squeaksource.com/PackageLoader'
  user: ''
  password: ''
```

Open a workspace and do: CoverageLoader new loadAll. During the installation you will be asked to specify whether or not to install optional packages and which technologies to gather coverage. Further information will be provided during installation.

The installer can be re-run at any time to load further packages or to update the installed packages to the latest versions.

In the following we list some technologies that can be used with Christo to gather coverage information. Read the instructions of those projects for a correct installation – or use the *PackageLoader* which will load them automatically.

- ByteSurgeon
  `http://www.squeaksource.com/ByteSurgeon`
- ObjectsAsMethods
  `http://www.squeaksource.com/ObjectsAsMethodsWrap`
- JMethods / Persephone
  `http://www.squeaksource.com/JMethods`
- (unstable) MethodWrappers
  `http://www.squeaksource.com/MethodWrappers`

# Bibliography

[Baud06a] Benoit Baudry, Franck Fleurey, and Yves Le Traon. "Improving test suites for efficient fault localization". In: *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pp. 82–91, ACM Press, New York, NY, USA, 2006.

[Beck00a] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison Wesley, 2000.

[Beck01a] Kent Beck and Martin Fowler. *Planning Extreme Programming.* Addison Wesley, 2001.

[Beck03] Kent Beck. *Test Driven Development: By Example.* Addison-Wesley, 2003.

[Beck98a] Kent Beck and Erich Gamma. "Test Infected: Programmers Love Writing Tests". *Java Report*, Vol. 3, No. 7, pp. 51–56, 1998.

[Berg07b] Alexandre Bergel, Stéphane Ducasse, Colin Putney, and Roel Wuyts. "Creating Sophisticated Development Tools with Omni-Browser". *Journal of Computer Languages, Systems and Structures*, 2007. To appear.

[Bran98a] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. "Wrappers to the Rescue". In: *Proceedings European Conference on Object Oriented Programming (ECOOP 1998)*, pp. 396–417, Springer-Verlag, 1998.

[Chil94a] J. J. Chilenski and S. P. Miller. "Applicability of modified condition/decision coverage to software testing". *Software Engineering Journal*, Vol. 9, No. 5, pp. 193–200, 1994.

[Choi89a] 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*, pp. 275–284, jan 1989.

[Zell05b] Holger Cleve and Andreas Zeller. “Locating causes of program failures”. In: *ICSE ’05: Proceedings of the 27th international conference on Software engineering*, pp. 342–351, 2005.

[Cock02a] Alistair Cockburn. *Agile Software Development.* Addison Wesley, 2002.

[Coll95a] Dave Collins. *Designing Object-Oriented User Interfaces.* Benjamin/Cummings Publishing, 1995.

[Coop95a] Alan Cooper. *About Face — The Essentials of User Interface Design.* Hungry Minds, 1995.

[Corn06a] Steve Cornett. “Minimum Acceptable Code Coverage”. 2006.

[Corn96a] Steve Cornett. “Code Coverage Analysis”. 1996.

[Corn99a] Steve Cornett. “What is Wrong with Line Coverage”. 1999.

[Deme02a] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns.* Morgan Kaufmann, 2002.

[Denk06a] Marcus Denker, Stéphane Ducasse, and Éric Tanter. “Runtime Bytecode Transformation for Smalltalk”. *Journal of Computer Languages, Systems and Structures*, Vol. 32, No. 2-3, pp. 125–139, July 2006.

[Denk06b] Marcus Denker, Orla Greevy, and Michele Lanza. “Higher Abstractions for Dynamic Analysis”. In: *2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pp. 32–38, 2006.

[Deur01a] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. “Refactoring Test Code”. In: M. Marchesi, Ed., *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pp. 92–95, University of Cagliari, 2001.

[Duca00a] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. “Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems”. In: *Proceedings of CoSET ’00 (2nd International Symposium on Constructing Software Engineering Tools)*, June 2000.

[Duca02v] Stéphane Ducasse. “Refactoring Browser et SmallLint”. *Programmez! Le Magazine du Développement*, Vol. 1, No. 46, Sep. 2002.

[Duca06b] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew Black. "Traits: A Mechanism for fine-grained Reuse". *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 28, No. 2, pp. 331–388, March 2006.

[Duca99s] Stéphane Ducasse, Michele Lanza, and Serge Demeyer. "Reverse Engineering based on Metrics and Program Visualization". In: *Object-Oriented Technology (ECOOP'99 Workshop Reader)*, Springer-Verlag, 1999.

[Eden01a] Amnon H. Eden. "Visualization of Object-Oriented Architectures". In: *International ICSE workshop on Software Visualization*, May 2001.

[Eick01a] 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*, Vol. 27, No. 1, pp. 1–12, 2001.

[Elba00a] Sebastian G. Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. "Prioritizing test cases for regression testing". In: *International Symposium on Software Testing and Analysis*, pp. 102–112, ACM Press, 2000.

[Fowl99a] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[Gael03a] Markus Gaelli. "Test composition with example objects and example methods". Technical Report IAM-03-009, Institut für Informatik, Universität Bern, Switzerland, May 2003.

[Gael03b] Markus Gaelli, Oscar Nierstrasz, and Roel Wuyts. "Partial ordering tests by coverage sets". Tech. Rep. IAM-03-013, Institut für Informatik, Universität Bern, Switzerland, Sep. 2003. Technical Report.

[Gael04a] 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)*, p. 317, June 2004.

[Gael04b] 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)*, pp. 114–123, 2004.

[Gael04c] Markus Gaelli, Oscar Nierstrasz, and Stéphane Ducasse. "One-Method Commands: Linking Methods and Their Tests". In: *OOPSLA Workshop on Revival of Dynamic Languages*, Oct. 2004.

[Gael06b] Markus Gaelli. *Modeling Examples to Test and Understand Software.* PhD thesis, University of Berne, Nov. 2006.

[Girb04a] Tudor Gîrba and Michele Lanza. "Visualizing and Characterizing the Evolution of Class Hierarchies". 2004.

[Girb06a] Tudor Gîrba and Stéphane Ducasse. "Modeling History to Analyze Software Evolution". *Journal of Software Maintenance: Research and Practice (JSME)*, Vol. 18, pp. 207–236, 2006.

[Gree05b] Orla Greevy and Stéphane Ducasse. "Characterizing the Functional Roles of Classes and Methods by Analyzing Feature Traces". In: *Proceedings of WOOR 2005 (6th International Workshop on Object-Oriented Reengineering)*, July 2005.

[Howd82a] W. E. Howden. "Weak Mutation Testing and Completeness of Test Sets". *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 4, pp. 371–379, jul 1982.

[Kuhn06d] Adrian Kuhn and Orla Greevy. "Summarizing Traces as Signals in Time". In: *Proceedings IEEE Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pp. 01–06, IEEE Computer Society Press, Los Alamitos CA, Oct. 2006.

[Lang05a] Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. "Visualization-based analysis of quality for large-scale software systems". In: *ASE*, pp. 214–223, 2005.

[Lanz99a] Michele Lanza. *Combining Metrics and Graphs for Object Oriented Reverse Engineering.* Master's thesis, University of Bern, Oct. 1999.

[Lien06a] Adrian Lienhard, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. "Capturing How Objects Flow At Runtime". In: *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pp. 39–43, 2006.

[YuSe02a] 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*, pp. 352–363, EEE Computer Society Press, Annapolis MD, nov 2002.

[YuSe05a] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. "MuJava : An Automated Class Mutation System". *Journal of Software Testing, Verification and Reliability*, Vol. 15, No. 2, pp. 97–133, jun 2005.

[Mads02b] Per Madsen. "Testing By Contract — Combining Unit Testing and Design by Contract". In: *The Tenth Nordic Workshop on Programming and Software Development Tools and Techniques*, 2002. On-line proceedings: http://www.itu.dk/people/kasper/NWPER2002/.

[Marc02a] Michele Marchesi, Giancarlo Succi, Don Wells, and Laurie Williams. *Extreme Programming Perspectives.* Addison Wesley, 2002.

[Marc03a] Michele Marchesi and Giancarlo Succi, Eds. *Extreme Programming and Agile Processes in Software Engineering.* Springer, 2003.

[Mari01a] Radu Marinescu. "Detecting Design Flaws via Metrics in Object-Oriented Systems". In: *Proceedings of TOOLS*, pp. 173–182, 2001.

[Mari04a] Radu Marinescu. "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws". In: *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pp. 350–359, IEEE Computer Society Press, Los Alamitos CA, 2004.

[Mari99a] Brian Marick, John Smith, and Mark Jones. "How to Misuse Code Coverage". International Conference and International Conference and Exposition on Testing Computer Software, jun 1999.

[Mars06a] Philippe Marschall. *Persephone: Taking Smalltalk Reflection to the sub-method Level.* Master's thesis, University of Bern, Dec. 2006.

[Mesz07a] Gerarde Meszaros. *XUnit Test Patterns - Refactoring Test Code.* Addison Wesley, jun 2007.

[Meye06a] Michael Meyer. *Scripting Interactive Visualizations.* Master's thesis, University of Bern, Nov. 2006.

[Meye06a] Michael Meyer, Tudor Gîrba, and Mircea Lungu. "Mondrian: An Agile Visualization Framework". In: *ACM Symposium on Software Visualization (SoftVis 2006)*, pp. 135–144, ACM Press, New York, NY, USA, 2006.

[Mill63a] Joan C. Miller and Clifford J. Maloney. “Systematic mistake analysis of digital computer programs.”. *Commun. ACM*, Vol. 6, No. 2, pp. 58–63, 1963.

[Moor01a] I. Moore. “Jester – a JUnit test tester”. In: M. Marchesi, Ed., *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, University of Cagliari, 2001.

[Mott06] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. “Mutation Analysis Testing for Model Transformations”. In: *ECMDA-FA*, pp. 376–390, IRISA, Campus Universitaire de Beaulieu, Bilbao, Spain, jul 2006.

[YuSe04a] 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*, Vol. 29, No. 5, pp. 1–4, sep 2004.

[Parn94a] David Lorge Parnas. “Software Aging”. In: *Proceedings 16th International Conference on Software Engineering (ICSE ’94)*, pp. 279–287, IEEE Computer Society, Los Alamitos CA, 1994.

[Parr02a] 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, Eds., *Extreme Programming Perspectives*, pp. 123–140, Addison-Wesley, 2002.

[Putn] Colin Putney. “OmniBrowser, an extensible browser framework for Smalltalk”. http://www.wiresong.ca/OmniBrowser.

[Reic07a] Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. “Rule-based Assessment of Test Quality”. In: *Proceedings of TOOLS Europe 2007*, 2007. To appear.

[Reng06a] Lukas Renggli. *Magritte – Meta-Described Web Application Development.* Master’s thesis, University of Bern, June 2006.

[Romp06a] Bart Van Rompaey, Bart Du Bois, and Serge Demeyer. “Characterizing the Relative Significance of a Test Smell”. *icsm*, Vol. 0, pp. 391–400, 2006.

[Romp06b] Bart Van Rompaey, Bart Du Bois, and Serge Demeyer. “Improving Test Code Reviews with Metrics: a Pilot Study”. Tech. Rep., Lab On Re-Engineering, University Of Antwerp, 2006.

[Roth01a] Gregg Rothermel, Roland Untch, Chengyun Chu, and Mary Jean Harrold. “Prioritizing Test Cases For Regression

Testing". *Transactions on Software Engineering*, Vol. 27, No. 10, pp. 929–948, Oct. 2001.

[Scha02b] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. "Traits: Composable Units of Behavior". Technical Report IAM-02-005, Institut für Informatik, Universität Bern, Switzerland, Nov. 2002. Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.

[Shne98a] Ben Shneiderman. *Designing the User Interface.* Addison Wesley Longman, third Ed., 1998.

[Smit82a] D.C.S. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harlem. "Designing the Star User Interface". *Byte*, Vol. 7, No. 4, pp. 242–282, Apr. 1982.

[VanE02a] Eva van Emden and Leon Moonen. "Java Quality Assurance by Detecting Code Smells". In: *Proc. 9th Working Conf. Reverse Engineering*, pp. 97–107, IEEE Computer Society Press, Oct. 2002.

[Wamp06a] Rafael Wampfler. *Eg – a Meta-Model and Editor for Unit Tests.* Master's thesis, University of Bern, Nov. 2006.

[Zell01a] Andreas Zeller. "Automated Debugging: Are We Close". *Computer*, Vol. 34, No. 11, pp. 26–31, 2001.

[Zell02b] Andreas Zeller. "Isolating cause-effect chains from computer programs". In: *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pp. 1–10, ACM Press, New York, NY, USA, 2002.

[Zell02a] Andreas Zeller and Ralf Hildebrandt. "Simplifying and Isolating Failure-Inducing Input". *IEEE Transactions on Software Engineering*, Vol. SE-28, No. 2, pp. 183–200, Feb. 2002.

[Zell05a] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann, oct 2005.

[Zell99a] Andreas Zeller. "Yesterday, my program worked. Today, it does not. Why?". In: *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 253–267, Springer-Verlag, London, UK, 1999.

[Zhu97a] Hong Zhu, Patrick A. V. Hall, and John H. R. May. "Software Unit Test Coverage and Adequacy". *ACM Comput. Surv.*, Vol. 29, No. 4, pp. 366–427, 1997.