TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites

Manuel Breugelmans and Bart Van Rompaey

manuel.breugelmans@student.ua.ac.be, bart.vanrompaey2@ua.ac.be
Lab On REengineering,
University of Antwerp

Abstract

The increasing interest in unit testing in recent years has resulted in lots of persistent test code that has to co-evolve with production code in order to remain effective. Moreover, poor test design decisions, and complexity introduced during the course of evolution harm the maintenance of these test suites – making test cases harder to understand and modify. Literature about xUnit – the *de facto* family of unit testing frameworks – has a fairly clear set of anti-patterns (called test smells). In this paper we present TestQ which allows developers to (i) visually explore test suites and (ii) quantify test smelliness. We present the feature set of this tool as well as demonstrate its use on test suites for both C++ and Java systems.

Key words: software maintenance, test quality, refactoring

1 Introduction

Both the rise of agile development methodologies as well as the need to find defects earlier in the development cycle has resulted in a rise in interest in unit testing – and the xUnit family of testing frameworks in particular [1]. However, testing also has an associated cost in the form of continuous maintenance, as test code needs to co-evolve with the production system.

Just like for production design, well known anti-patterns (*test smells*) for test code exist that harm the understanding and modification of test cases [2,3]. In this work we propose a prototype tool TestQ [1] to (i) visually explore the design of test suites and (ii) quantify the presence of static test smells. As such, it is important to note that this tool is intended as an assessment

[1] http://tsmells.googlecode.com

of the maintainability of the test *code*. Thus, the question *How maintainable is my unit test code?* is answered but *How well does my suite excercise and verify the SUT?* remains unanswered. Test engineers and regular developers alike can use this information to spot refactor opportunities. As TestQ is based on a formalism for the xUnit family of testing frameworks [1], it targets language-independent analysis. The case studies in this work use the popular JUnit and CppUnit frameworks.

After a summary of related work in Section 2, we expand upon the detection strategy for test smells in Section 3. Next, we describe the main features of TestQ by means of a running example in Section 4. In Section 5, we discuss the findings for two more systems. Finally, we describe the architecture of TestQ and formulate a conclusion.

2 Related Work

We identified the following work in the domain of test suite analysis, with a focus on test suite design and maintainability aspects. Several authors have been describing and cataloging test smells. Van Deursen et al. introduced the concept of a test smell as a poorly designed test [2]. Meszaros broadens the scope of the concept, by describing test smells that act on a behavior or a project level, next to code-level smells [3]. Reichhart et al. propose TestLint, a rule-based tool to detect static and dynamic test smells in Smalltalk SUnit code. In previous work we introduced a formalism for xUnit tests [4]. We proposed and evaluated a set of metrics to detect two test smells, General Fixture and Eager Test.

3 Test Smell Detection Strategy

In this section we expand upon the detection strategy for test smells used in TestQ . First, we reflect upon terminology and some important concepts of xUnit, summarized from [4]. We assume a system that contains production code, test code and libraries. The testing framework belongs to the libraries. The test code contains a set of test cases, typically implemented as classes in xUnit. Every test case has a fixture, a set of instance variables of the test case describing the unit under test as well as data objects. Before every test command, a container for an individual test (typically a method of the test case), the fixture is initialized in the test case setup method. Test helpers are methods that support test commands by abstracting e.g. recurring verification behavior. In this work, we consider the static test smells in Table 1 and a metrics-based detection strategy in terms of xUnit concepts. The thresholds

to configure smell presence are configurable. During development we defined these detection-metrics rigorously, by means of a formal model based on set theory. [2]

Name	Description and Detection
Assertionless	A test command that does not invoke asserts. We count the number of invoked framework asserts.
Assertion Roulette	A test command with a high number of descriptionless checker invocations. We count the number of invoked, descriptionless asserts.
Duplicated code	Pairs of test commands that contain the same method invocation and data access sequence. We count the length of sequences of invocations and accesses.
Eager Test	A test command that exercises too much at once. We count the number of invoked production methods.
Empty Test	Test commands without a body. We count the number of invocations and accesses.
For Testers Only	Production methods or functions which are introduced specifically to make the unit under test testable. We tag production methods that are only called by test code.
General Fixture	A test case fixture that is too large, with test commands only using part of the fixture. We use a set of 3 metrics introduced in [4], characterizing the fixture size and the usage rate of the fixture elements by the test commands.
Indented Test	Loops and conditionals break the linear character of a test, and might make it too complex. We count the number of decision points.
Indirect Test	test commands that exercise components via other components. We count the number of production types as an indicator for this smell.
Mystery Guest	External resources used by a test harm stability and isolation. Test commands that use a standard set of such resource libraries (such as I/O) are marked.
Sensitive Equality	Verification by dumping an object's characteristics to string is easy and fast, yet makes tests fragile to small changes. We tag tests that invoke a *toString* method (the typical Java implementation).
Verbose Test	Long test negatively influencing readability. We use SLOC as an indicator.

Table 1
Smells with detection strategy

[2] http://fenix.cmi.ua.ac.be/ p035120/formspec.pdf

4 Tour of TestQ

In this section, we highlight the two main features of TestQ . A first set of views are more oriented towards the structure of the test suite as a whole, but provide already an indication of some size characteristics of individual test cases. The second feature encompasses the annotation of test cases with test smell presence. The environment facilitates the switching between views and level of detail, by offering menu entries, navigational tree views as well as right-click context menus.

As a running example, Poco [3] will be used. Poco is an industrial strength C++ networking framework (192 kSLOC), with an extensive CppUnit suite (55 kSLOC). Table 2 quantifies this system in terms of production and test artifacts.

Production	
#classes	840
#methods	9527
#functions	912
Test Cases	
#test cases	190
#test commands	1094
#helpers	93
#fixture	356

Table 2
Poco size metrics (trunk 08-02-28)

4.1 Test Suite Topology

Motivation To allow developers to explore a test suite's structure, with a focus on exceptional entities via size annotations.

Description Three complementary views elaborate upon the suite's structure. The Expandable Test Suite Tree is a vertical text-based tree panel that contains the test suites, their test cases and test methods divided in commands, fixture methods (setup and tear down) and helpers. As such it maps the source tree structure to the abstract graph based representations.

The leaves in the Polymetric Suite View [5] tree represent test cases grouped by suite. Three metrics are used: (i) the number of commands in a test case determines the height, (ii) the width is relative to the ratio of test

[3] http://sourceforge.net/projects/poco

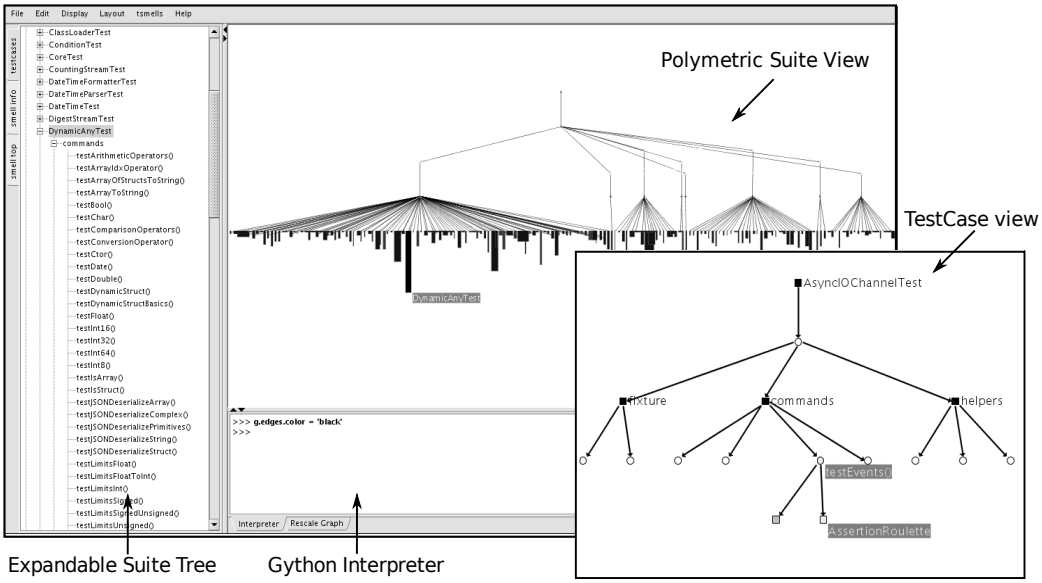

Fig. 1. TestQ visualizing the test suite's topology.

case SLOC divided by number of commands, (iii) while the coloring is based on the presence of helper and fixture methods. As such, exceptional test cases immediately strike out as refactoring candidates: long nodes can use an *Extract Test Case Class*, while the wide ones are prime candidates for *Extract Helper Method*. Test cases without fixture are an indicator for low cohesion, or a lot or redundant code in individual test commands. This view contains a wealth of information, often you can spot the different developers test coding style.

The Test Case View is a hierarchical graph representation of a single test case. Its methods are grouped by test commands, test helpers and fixture . Test smells are shown as separate nodes attached to the originating method(s) (or case) and colored per type. Hovering over the nodes pops up some extra information such as the associated metric value(s).

Case Study From the Expandable Test Suite Tree we learn that Poco's test code is decomposed in eight test suites that each correspond to a different production module. Expanding any of these module nodes reveals the test cases. After a quick inspection we see that *Foundation::testsuite* and *net:testsuite* hold the bulk of the test cases, while *Data::SQLite:testsuite* only contains a single test case *SQLiteTest*. Further investigation reveals that this test case contains 70 small testcommands, hence the exceptionally long but slim node. Splitting this test case in multiple test cases with logically related commands increases the readability.

In the Polymetric Suite View, a set of massive test cases from the *Foundation::testsuite* suite strike the attention. These wide test case nodes indicate VerboseTests. Investigation of the *DynamicAnyTest*, one exceptionally wide

and long node shows that this is indeed a candidate for refactoring, as 64 code smells where found in the 40 commands (see Table 3). Especially the smells VerboseTest and AssertionRoulette are everywhere. Moreover, we found instances of DuplicatedCode and IndentedTest. This test case contains a huge amount of assertions in long commands. Browsing the source (right click → toSource) proves this, as the tests check multiple scenarios in a single command and thus does not convey the intent clearly. If one of these tests start failing, the manual inspection of the source becomes a challenging task. The metrics numbers for this test case are:

metric	value	metric	value
SLOC	1792	#smells	64
SLOC/mtds	40.72	#AssertionLess	5
min(SLOC)	0	#VerboseTest	29
max(SLOC)	104	#EmptyTest	1
#commands	40	#smells/mtd	1.45
#helpers	2	#AssertionRoulette	25
		#DuplicatedCode	2
		#IndentedTest	2

Table 3
DynamicAnyTest metrics

4.2 Test Smell Detection

Motivation Detect occurrences, significance and frequency of test smells.

Description Test entities are ranked in a Stink Parade, based upon a selected test smell and the metric values associated with this smell. As such, developers can focus on the worst offenders for a selection of smells of their interest. We see this a pragmatic approach considering the lack of studies to compare the impact or interaction of test smells.

The Suite Smell View shows all the test cases as separate graphs in a single view. The center nodes represent the test case, nodes on distance one the test methods and attached to them the smells. This way the smelliness hot-spots are crystal clear. While most smells stay in a single testcase, DuplicatedCode typically spans over multiple methods. This view shows these cases stringed together in clone clusters, possible ranging over multiple test cases.

The Smell Pie shows the ratio and absolute count of the different code smells in a pie chart. Interesting here is the high level of interactivity. Selecting a smell in the chart will highlight all the occurences in a view. Colorizing or removing a certain smell is a couple of mouse clicks away.

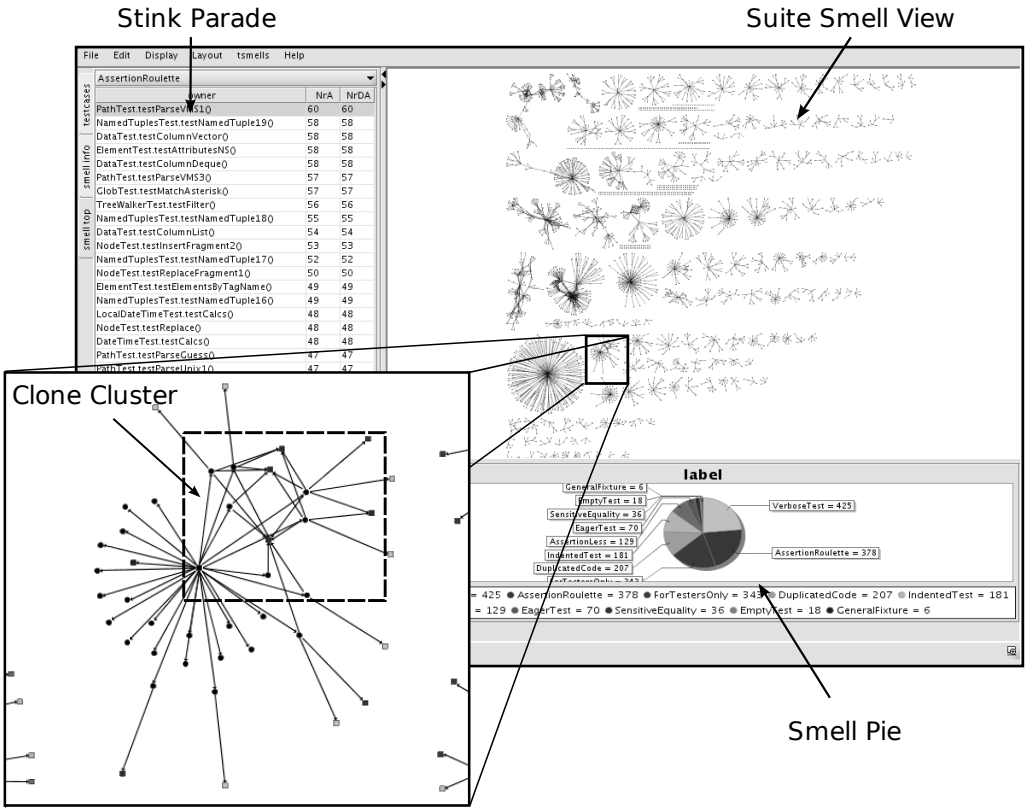

Fig. 2. Test Smell Detection for Poco with focus on *Foundation::EventTest*

Case Study When investigating Poco for the AssertionRoulette smell with the aid of the Stink Parade, several high rollers present themselves. Top of the list is *PathTest.testparseVMS1()* with 60 assert invocations, all without descriptive messages. These high numbers are often the symptom for another smell, i.e. multiple test scenarios in a single command. Browsing the source, as shown in Listing 1, confirms this assumption. Smell concentrations in Poco are visually obvious from the Suite Smell View, the magnitude of the testcase-flowers serve as a smelliness barometer. The clone cluster in Figure 2 shows a handful of methods linked together by DuplicatedCode smells. When looking at the source we identify candidates for *Extract Helper Method* indeed. The Smell Pie shows a high ratio of VerboseTests and AssertionRoulettes, as well as ForTestersOnly. The accuracy of the detection strategy for the latter smell depends upon the completeness of the composed model, i.e. are the clients of the methods under test part of the system. In the case of Poco, a network library, API methods used by the test code are false positives.

Listing 1. AssertionRoulette DuplicatedCode and VerboseTest in PathTest

```
1  void PathTest::testParseVMS1() {
       Path p;
3      p.parse("", Path::PATH_VMS);
       assert (p.isRelative());
5      assert (!p.isAbsolute());
       assert (p.depth() == 0);
7      assert (p.isDirectory());
       assert (!p.isFile());
9      assert (p.toString(Path::PATH_VMS) == "");

11     p.parse("[]", Path::PATH_VMS);
       assert (p.isRelative());
13     assert (!p.isAbsolute());
       assert (p.depth() == 0);
15     assert (p.isDirectory());
       assert (!p.isFile());
17     assert (p.toString(Path::PATH_VMS) == "");
       ...
19 }
```

5 Case Studies

ArgoUML is an open source UML CASE tool implemented in Java. We consider version 0.20, containing 100 kSLOC production code and 10 kSLOC test code. The JUnit 3.x test suite entails 120 test cases. The polymetric test suite view shows a homogenous topology, small test cases 2.6 commands on average, with reasonably sized methods averaging 18 SLOC per command. A couple of smelly outliers require further attention. Interestingly most of those were written by authors other than the main tester. This shows clearly in the polymetric view. A particular smelly case is *targetmanager::TestTargetManager* with 25 instances of DuplicatedCode, 11 VerboseTests (44 SLOC on average), 7 control structures a GeneralFixture and more. Prime candidate for refactoring! In general this is a clean test suite with a couple of extremely stinky cases.

Checkstyle is a development tool which automates the process of checking Java code. We evaluate release 4.4, containing 20 kSLOC production code and 10 kSLOC test code (157 test cases). The average SLOC per command equals 6 which is surprisingly low. This is due to a project specific test language implemented in root test cases, for example *BaseCheckTest*.The suite is well balanced with 3.5 commands per case, save some prohibitively

large cases including *IndentationCheckTest* with 30 commands, *RegexpTest* 21, *JavadocMethodCheckTest* 25. Remarkable is the low percentage of fixture methods, only 15% of the testcases has an explicit setUp() method. Code level smells are scarce. There is, for example only a single AssertionLess command on a total of 546 commands. Few AssertionRoulettes, top offenders are commands in *DetailASTTest* and *UtilsTest* with 9 assertions, nothing spectacular. The DuplicatedCode instances found consist mostly of false positives caused by the project specific test language. These can either be removed swiftly in guess or prevented altogether by adapting the model constructor.

6 TestQ Architecture

The tool is build as an extension of the Fact Extraction Tool CHain (Fetch) [4], a reverse engineering tool chain. The graph exploration environment Guess [5] is customized as visual front-end. We first shortly describe Fetch and Guess, then explain how TestQ is realized on top of these tools.

Fetch is a tool chain for software analysis targeting the exploration of large C/C++/Java software systems for (i) dependency analysis; (ii) pattern detection; (iii) visualization; (iv) metric calculation and similar types of static analysis [6]. Fetch builds up a model of a system from the source code. Crocopat [7] is used as a query engine.

Guess [8] is an interactive graph visualization tool for software exploration, amongst other. It features graph layout, navigation and manipulation operations.

TestQ builds an xUnit specific model (introduced in [4]) from the model extracted by Fetch, making abstraction from language and actual xUnit implementation. Currently model constructors for multiple versions of CppUnit, JUnit and QTest. Metrics and smell instances are then deduced from this xUnit model. These can be loaded in the Guess environment or separately processed with e.g. a simple spreadsheet. Next, multiple graph views of a test suite enriched with metric information are composed, making use of the interactivity and extensibility features of Guess. For example, hovering over a node or edge will give access to all kinds of information and operations for that entity. Selecting test entities in the tree pane highlights them in the graphs. Context menu actions on test cases and methods allow for instant source browsing.

[4] http://lore.cmi.ua.ac.be/fetchWiki/
[5] http://graphexploration.cond.org/

7 Conclusion

In this paper we introduced a dual-purposed reverse engineering tool for xUnit test suite analysis. First of all, we present a visual approach to explore the structure of a test suite. Annotated with size metrics, developers can identify relevant test cases for further exploration. As a second feature, TestQ contains a test smell detection engine detecting 12 static test smells, presenting the results both in a quantitative manner (in a sorted table) as well as using visual markers on the test suite's topology. As an evaluation, we applied the tool on three case studies and discussed some of the findings. Configurable metric thresholds allow the user to customize the detection process as well as prioritize smells that are deemed important. Indeed, more research is needed beyond the few empirical studies to further characterize test smells, their interaction and impact on maintainability.

References

[1] P. Hamill. *Unit Test Frameworks*, chapter Chapter 3: The xUnit Family of Unit Test Frameworks. O'Reilly, 2004.

[2] Arie van Deursen, Leon Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In *Proc. Extreme Programming and Flexible Processes (XP)*, pages 92–95, 2001.

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

[4] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800–817, December 2007.

[5] Michele Lanza and Ducasse Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, 2003.

[6] Bart Du Bois, Bart Van Rompaey, Karel Meijfroidt, and Erik Suijs. Supporting reengineering scenarios with fetch: an experience report. *Electronic Communications of the EASST Volume 8: ERCIM Symposium on Software Evolution*, (8), 2007.

[7] Dirk Beyer. Relational programming with CrocoPat. In *Proceedings of the 28th ACM/IEEE International Conference on Software Engineering (ICSE 2006, Shanghai, May 20-28)*, pages 807–810. ACM Press, New York (NY), 2006.

[8] Eytan Adar. Guess: A language and interface for graph exploration. In *Proceedings of CHI*, 2006.