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 , a tool that allows developers to (i) visually explore test suites and (ii) quantify test smelliness. We present the feature set of this tool as well as its architecture, and demonstrate its use on a C++ test suite of considerable size.

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, instances of well known anti-patterns (*test smells*) for test code harm the understanding and modification of test cases [2,3]. In this work we propose a tool called TestQ [1] to (i) visually explore the design of test suites and (ii) quantify the presence of static test smells (i.e. smell instances that can be identified by inspecting the design and source code of a test suite). As such, this tool is intended to

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

assessment of the maintainability of the test *code*, seeking to answer the question *How maintainable is my unit test code?* Test engineers and regular developers alike can use this information to spot refactoring 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 implementations variants of xUnit.

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. We continue with describing the architecture of the tool and the lessons we learned as tool builders in Sections 5.1 and 5.2.

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 [4]. Neukirchen and Bisanz composed a catalogue of code smells for TTCN-3 test suites, and offer tool support [5]. In previous work we introduced a formalism for xUnit tests [6]. We proposed and evaluated a set of metrics to detect two test smells, General Fixture and Eager Test.

Considering reverse engineering and visualizing test suites, Agrawal et al. introduce a set of techniques to enhance program understanding, debugging and testing [7]. Among others, the χSuds tool suite assist developers in achieving high test coverage, locating errors as well as minimizing regression sets. Via source code coloring, the developer perceives the coverage level, erroneous locations or execution frequency. Gaelli et al. observe that not all unit tests are alike [8]. Therefore, a taxonomy that distinguishes unit tests based on the focus on one or more methods, type of expected outcome, etc. Their automated classification approach for SUnit tests using heuristics achieves a high overall precision (89%) and a moderate recall (52%). One of the steps the authors identify as future work involves making explicit the relationship between unit tests and methods under test. Van Geet and Zaidman hypothesize that unit tests covering multiple units are less suited as documentation as such tests are harder to understand [9]. In a case study involving the Ant project, the median number of methods executed by a test command is more than 200, which make them conclude that the test suite of this particular project is not well suited for documentation purposes. To gain knowledge about the inner working of a software system, Cornelissen et al. use sequence diagrams obtained from test execution [10]. The use of abstraction, separation of test stages and stack depth limitations make such diagrams scalable.

Some other studies investigated the co-evolution a test suite has to undergo in order to remain up-to-date with the evolving code. Elbaum demonstrated how small changes to the system resulted in major coverage drops [11], while Moonen et al. describe how refactorings can even invalidate tests [12]. As such, the larger the test suite, the more 'regressions' can be expected.

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, then we present the set of test smells that we target and the proposed metrics to detect and quantify them.

We summarize the terminology and xUnit concepts as described in [6]:

Production code – Code developed by the project team that will end up in the released product.

External libraries – The set of software libraries a system uses, but that are not developed by the project. One of the external libraries is the testing framework.

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

The test code contains a set of test cases, typically implemented as classes in an object-oriented implementation of 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. These smells were introduced by Van Deursen et al., Meszaros and Reichart [2,4,3]. We interpreted these informal descriptions and translated them into a formalism based upon set theory. In appendix A, we detail how we formally define these metrics. In TestQ , the user can configure metrics thresholds, i.e. indicate from which value on a particular test entity exhibits a test smell.

In general no agreement exists about this set of test smells, and there may well exist reasons to design tests what others consider as a smell. As an example, consider the Eager Test example. Van Deursen et al.use the term *Eager Test* to refer to a test method checking several methods of the object to be tested [2]. They say that dependencies between the enclosed implicit tests make such tests harder to understand and maintain. The xUnit family of testing frameworks, as exemplified by JUnit, advises its users to avoid dependencies between tests. Test methods are supposed to be independent artifacts, sharing at most a (re-initialized) fixture constituting the unit under test. Fewster and Graham state that the efficiency benefit of such long tests (where setup and tear-down is only performed once) is far outweighed by the inefficiency of identifying the single point of failure [13]. On the other hand, Meszaros uses the term *ChainedTests* to point to this test design [3], motivating that it may be a valid strategy for overly long, incremental tests. Test frameworks such as TestNG [14] and JExample [2] even provide facilities to make dependencies between tests explicit. Kuhn et al. present a case study where the introduction of dependencies between tests reduce the number of failed tests on average by 90% for single defects introduced in the system under test [15].

[2] http://www.iam.unibe.ch/~scg/Research/JExample/

Name	Description	Metric	Why
Assertionless	Test commands that do not invoke asserts.	Number of invoked framework asserts.	Absence of verification results in useless tests.
AssertionRoulette	Test commands with lots of asserts without description	Number of invoked description-less asserts.	Negatively influences defect localization and readability.
DuplicatedCode	Sets of test commands that contain the same invocation and data access sequence.	Length of similar sequences of invocations and accesses.	Directly affects maintenance cost
EagerTest	Test commands that exercise too much at once.	Number of invoked production methods.	Multiple tests inside one test command make it harder to identify test objectives and introduce implicit dependencies. Hard to track single point of failure.
EmptyTest	Test commands without a body.	Number of invocations and accesses.	Indicates forgotten, stubbed or commented-out code.
ForTestersOnly	Production methods or functions which are introduced specifically to make the unit under test testable.	Invocation of production entities only in test code.	Exposure of internals results in fragile tests.
GeneralFixture	A test case fixture that is too large.	We use metrics introduced in [6], characterizing the fixture size.	Hard to understand logic and objective of test case
IndentedTest	Overuse of loops and conditionals in test code.	Number of decision points.	Tests should be simple and linear.
IndirectTest	Test commands that exercise components via other components.	Number of production types.	Impacts defect localization.
MysteryGuest	Use of external resources in test commands.	Invocation of a standard set of I/O entities.	Harms stability and isolation.
SensitiveEquality	Verification by dumping an object's characteristics to string.	Number of invocations of *toString* methods (typical Java implementation).	Is easy and fast, yet makes tests fragile to small changes.
VerboseTest	Long test command bodies	SLOC	Affects readability.

Table 1
Smells with detection strategy

4. Tour of TestQ

In this section, we highlight the two main features of TestQ . The first feature, called Test Suite Topology, targets test suite wide structural analysis, thereby already providing an indication of some size characteristics of individual test cases. An integrated set of views realize this feature. A second set of views enriches test entities with raw metric values and smell information (i.e. interpreted metrics) to constitute the Test Smell Detection feature.

The environment facilitates switching between both features (and their views) as well as the level of detail, by offering menu entries, navigational tree views, zooming and right-click context menus. The views can be manipulated and queried both through the GUI as well as through custom scripts executed by an interpreter. This interpreter furthermore eases the process of creating new views.

We demonstrate both features here by means of a running example. Poco [3] is an

[3] http://pocoproject.org/

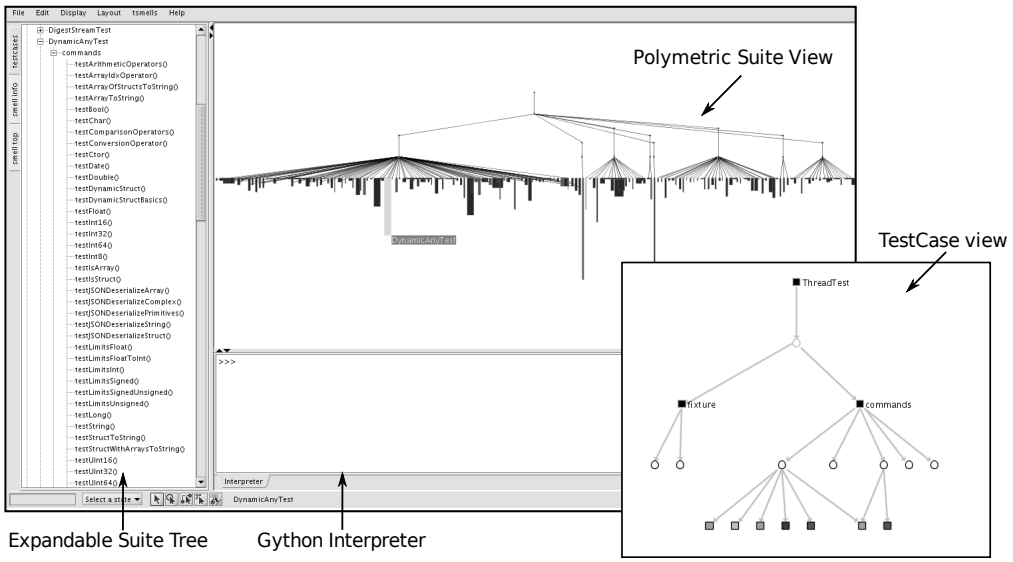

Fig. 1. TestQ visualizing the test suite's topology.

industrial strength C++ class library (192 kSLOC) with an extensive CppUnit test suite of 190 test case classes (55 kSLOC).

4.1. *Test Suite Topology*

Motivation 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 Suite Tree is a vertical text-based tree panel that contains the test suites, their test cases and test methods (test commands, fixture methods setup and tear down, and helpers). As such it maps the source tree structure to the more abstract, graph based representations.

The leaves in the Polymetric Suite View [16] 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 case SLOC divided by number of commands; and (iii) the coloring is based on the presence of helper and fixture methods. As such, exceptional test cases immediately strike out as refactoring candidates: the *Extract Test Case Class* refactoring operation applies to the long nodes, while the wide ones may qualify for e.g. *Extract Helper Method*. Test cases without fixture are an indicator for low cohesion, or a lot or redundant code in individual test commands.

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 test 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. A quick inspection reveals that *Foundation::testsuite* and *net:testsuite* hold the bulk of the test cases, while *Data::SQLite:testsuite* only contains a single test case *SQLiteTest*. Yet, this test case appears to contain 70 small test commands, hence the exceptionally long but slim node. Splitting this test case in multiple test cases with logically related commands would increase 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 *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 test commands (see Table 2). 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 (directly accessible in TESTQ):

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

Table 2
DynamicAnyTest metrics

4.2. *Test Smell Detection*

Motivation Explore hot spots of test smells; quantify individual instances.

Description This set of views reveals code level test smells. Each smell is represented as a node, connected to the test entity that is impacted. By hovering over a node, more detailed information on each of the smell instances becomes accessible. This information includes (i) the name of the one or more owner entities; (ii) corresponding source file(s) and line number(s); and (iii) associated metric values. At any point a table of the smelliness metrics for a specific entity can be printed, resembling table 2.

The SMELL FLOWER VIEW is a collection of graphs which show test cases as separate 'flowers'. The center node of such a flower represent the test case itself, surrounded by its test methods and attached smell instances. This way test cases that are smell hotspots strike as large colored graphs. The user can also focus on a particular smell by either colorizing smell nodes or by removing smell-types. While most smells stay in a

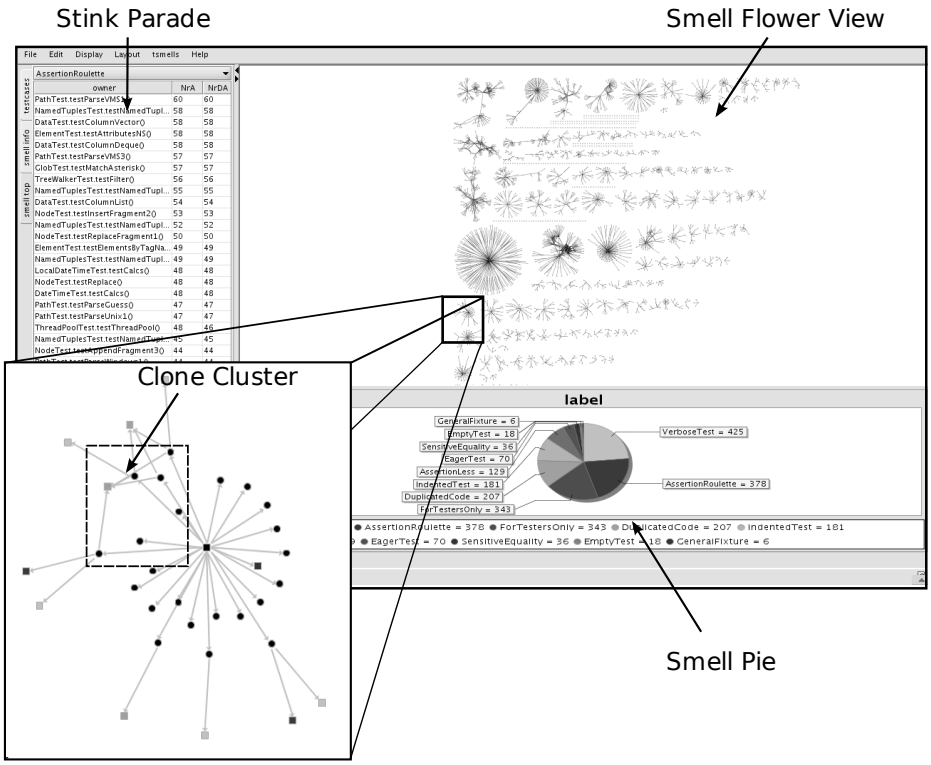

Fig. 2. Test Smell Detection for Poco with focus on *Foundation::EventTest*

single case, DuplicatedCode typically spans over multiple methods that may belong to different flowers. This results in so called 'Clone Clusters'.

By default the complete suite and all smell types are shown. This often result in a crowded view. The tool's zoom options and interactivity allows the user to shift between the overall view and particular test cases. For example, assume that we are only interested in VerboseTests located in Poco's *Foundation::testsuite*. Narrowing the view for this purpose requires five lines of Gython code (a domain specific extension of Jython [4]), injected at run-time. This results in the custom view shown in Figure 3. The second shot was constructed identically but based on the AssertionRoulette smell. The corresponding code is presented in Listing 1.

Listing 1. A custom view using Gython scripting

```
1 remove((entity == 'smell') & (label != "VerboseTest"))
2 suites = (entity == 'package') & (name != 'Foundation::testsuite')
3 remove(suites + suites.successors + suites.successors.successors)
4
5 for vb in (label == 'VerboseTest'):
6         vb.width = int(metricDict['VerboseTest'][vb.name]['LOC'])
```

[4] http://www.jython.org

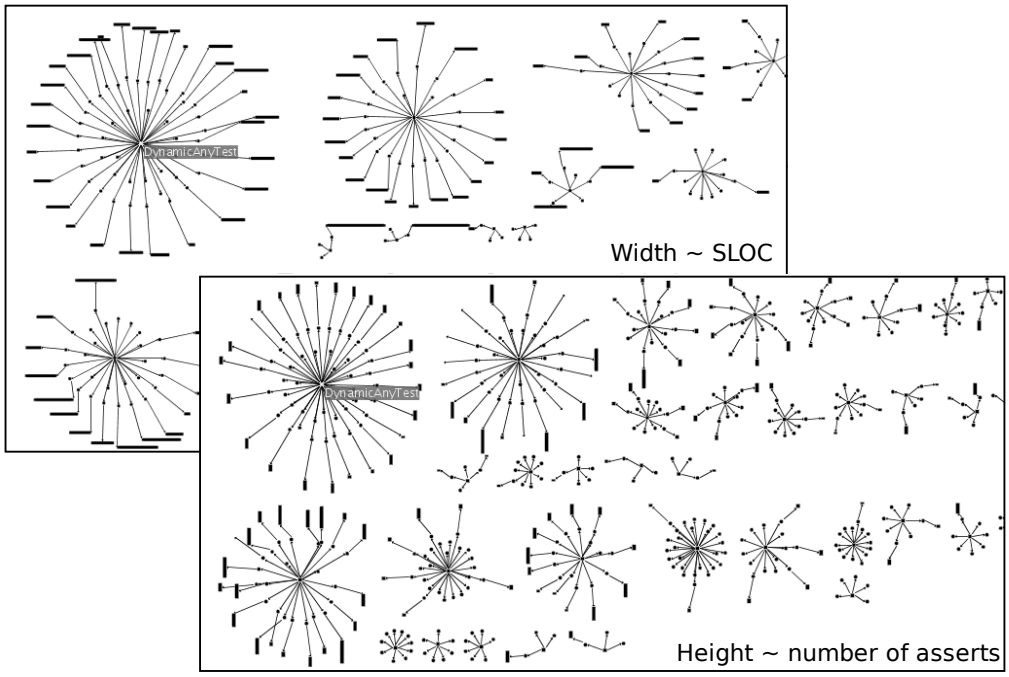

Fig. 3. Customized Smell Flowers

Since the significance of these smells is quite subjective we have introduced the Stink Parade. For example, one might argue that a high number of asserts does not stink but is, on the contrary, a sign of strong verification. This Stink Parade is a table that ranks the top-scoring test entities and their associated metric values for a smell of choice. As such, developers can focus on the worst offenders for a selection of smells of their interest. We see this as a pragmatic approach considering the lack of studies to compare the impact or interaction of test smells.

The Smell Pie shows the ratio and absolute count of the different code smells in a pie chart. Selecting a smell in the chart will highlight all the occurrences in a view. Colorizing or removing the nodes of a certain smell-type is a couple of mouse clicks away.

Case Study When investigating Poco for the AssertionRoulette smell with the aid of the Stink Parade, several high rollers present themselves. At the top of the list stands *PathTest.testparseVMS1()*, with 60 assert invocations without descriptive messages. Such high AssertionRoulette numbers are often the symptom for another smell, EagerTest, i.e. multiple test scenarios in a single command. Browsing the source code, presented in Listing 2, confirms this assumption. Smell instance concentrations in Poco become visible in the Smell Flower View and custom derivatives. The upper left flower in Figure 3's both screenshots is *DynamicAnyTest*, which was identified before. 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 2. Part of the source code of PathTest.testparseVMS1 confirming the presence of AssertionRoulette DuplicatedCode and VerboseTest smell instances

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

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

5. Tool Building

In this section, we present the architecture of TESTQ and the lessons we learned as experimental tool builders.

5.1. TESTQ *Architecture*

Summarizing, the tool is build as an extension of the Fact Extraction Tool CHain (Fetch) [5] , a reverse engineering tool chain. The graph exploration environment Guess [6] is customized as visual front-end for TESTQ .

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 [17]. Designed as a pipes and filters architecture, Fetch chains a set of open source components together, most notably:

- SourceNavigator: a multilanguage integrated development environment with parsers for multiple languages. We use these parsers in batch processing mode to extract structural information from the source code [7] .

[5] http://lore.cmi.ua.ac.be/fetchWiki/
[6] http://graphexploration.cond.org/
[7] http://sourcenav.berlios.de/

– pmccabe: McCabe-style function complexity and line counting for C and C++ [8].
– JavaNCSS: the equivalent of pmccabe for Java [9].
– snavtofamix: unifies the output of the above tools, creates cross-referencing links between structural components and generates a FAMIX model in Case Data Interchange Format (CDIF) [18].
– CDIF2RSF: translates the CDIF to a Rigi Standard Format (RSF) style fact base.
– Crocopat: a graph query engine for RSF with a Prolog-like language called RML [19].
– Guess: a graph exploration and visualization environment.

TestQ relies on Fetch to build an RSF model of the source code. Using Crocopat this general purpose object-oriented model is then refined with test entities following xUnit concepts as introduced in [6], making abstraction from language and actual xUnit implementation. Currently TestQ has model constructors for multiple versions of CppUnit, JUnit and QTestLib. However, the extension to other frameworks is straightforward, provided the language constructs used in the xUnit implementation are present in the composed meta-model. Listing 3 contains a sample RML query, identifying test cases and test methods written using QTestLib, the testing framework of the popular Qt C++ application framework. Essentially, it identifies classes that belong to a file that includes the QtTest header file as test cases, and methods of that class as test methods.

Listing 3. RML script identifying test case classes and methods among the classes in the RSF model

```
1  TestCaseId(tcid) :=
2       EX(fid, qtestid, qtest,
3               Class(tcid,_) &
4               ClassBelongsToFile(tcid, fid, _) &
5               Include(_,fid, qtestid) &
6               File(qtestid, qtest) &
7               @"QtTest"(qtest));
8
9  TestMethodId(x) :=
10     EX(y,
11          TestCaseId(y) &
12          MethodBelongsToClass(x,y));
```

Next, this test-aware model is queried for the presence of test smell instances. The outcome is loaded in the Guess environment, or can alternatively be processed separately with e.g. a spreadsheet for trend analysis. [20] is an interactive graph visualization tool for software exploration, amongst others. It features graph layout, navigation and manipulation operations. Within Guess, multiple graph views of the 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.

Guess contains a scripting environment for customization purposes. Gython is a domain specific language derived from Jython, a Java implementation of python. Through an

[8] http://www.parisc-linux.org/~bame/pmccabe/
[9] http://www.kclee.de/clemens/java/javancss/

integrated console, Gython scripts can be executed on the fly. This allows the user to morph graphs, change visualizations and even modify the GUI. Gython scripts can be passed on to Guess from the command line as well, to customize the tool at start-up.

5.2. *Lessons Learned*

Reuse. The visualizations offered by TestQ are the result of a chain of tasks that process the source code via multiple intermediate data representations into the eventual end-user format. Many of these tasks are well known in software development environments, and efficient algorithms and implementations have been studied extensively for tasks such as parsing source code [21], creating object-oriented models [18], visualizing software systems [16], querying [19] and layouting graphs [22].

Pipes and Filters. Architectures built around pipes and filters have long been praised for their ease of extension and replacement of components. The use of files as exchange mechanism between components facilitates debugging the tool chain at various points. Moreover, it allows us to share these intermediate results with other tools that use a same file format (One of our intermediate formats representing a model for an object-oriented system is FAMIX CDIF [18], a file format that was used in earlier versions of Moose [23]). In the context of Fetch, a success story of this architecture is the switch in code metric tool from cccc [10] (used in [24]) to pmccabe [11] . This switch was not only implemented in a reasonable amount of time, it moreover only required a limited amount of local changes.

Open Source. Using open source components in TestQ offered two advantages: (i) the freedom to adapt the reusable components to our needs and (ii) an easier distribution scheme that encourages the interested to try (and even modify) the tool. In many cases, we slightly changed components to better serve our needs and recompiled the source code into optimized binaries for the platforms we support.

Testability. Due to the application of non-main stream technologies, the use of existing test frameworks to verify TestQ was limited. More specifically RML/CrocoPat, which makes up for a large percentage of the code, does not have proper testing facilities, nor is writing a unit testing framework for it straightforward. To remedy this we resorted to scripted input-output tests run inside a home grown framework. This results in increased defect localization time and maintenance time. For the Guess extension, we decided not to write developer tests, as:

- Guess extensions are written in this graph specific jython dialect.
- Numerous interfaces were test-unfriendly.
- Mostly of the code being GUI-related.
- The exploratory nature and short development time of the tool.

The Fetch model extractor *snavtofamix*, however, is accompanied with copious PyUnit tests and an input-output suite.

Robustness. When choosing a parser, we had to choose between the criteria accuracy and robustness, as observed in a extraction tool contest reported by Sim et al. [25]. Reasoning that we would encounter multiple variants of C++ (gcc, Visual C++, .NET C++, plain C, etc.), we opted for a robust parser. Inevitably, this results in a certain amount of noise in the extracted data that we had to cope with further on. Moreover,

[10] http://cccc.sourceforge.net
[11] http://www.parisc-linux.org/ bame/pmccabe/

it became clear that giving in to accuracy would lead to models that are not necessarily complete, a side effect the user should be well aware of. At the bright side, we expect the tool to deliver results, whatever C++ source code it is presented. This resulted in successful deployments of Fetch in industrial settings as reverse engineering and internal quality monitoring tool [17].

Performance. At one side, the tool chain benefits from selecting existing components that were perceived as efficient. For example, Beyer et al. compared the performance of his Crocopat tool with binary relations in Prolog and a relational database approach citebeyer05. The pipes and filters architecture, however, does not really promote performance. A considerable amount of CPU cycles is lost in (i) preparing the right output for the next component and writing it to disk, and (ii) in the relatively slow interpreters executing the many shell scripts that we used to glue the tool chain together. Moreover, as each component fulfills exactly one task without awareness of other components, the required model nor visualization can be built up incrementally. This results in a cycle of multiple minutes to process source code into the graph visualizations, even for small systems. As a consequence, TestQ is too slow to become usable in a forward engineering context where the tool could be used during coding and testing.

Portability and Distribution. Due to the variety of components developed in many programming languages, we quickly noticed that porting and distributing Fetch and TestQ was not going to be easy. Still, the software requirements do not appear as the hardest challenge, as almost all operating system distributions have pre-installed shell, Java, Python, etc environments. The major challenge appeared to be the installation of components that required compilation (Source Navigator, pmccabe). One of the solutions lies in the redistribution of binaries for the platforms we support, something that the open source licenses allow us to do.

6. 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 a C++ case study and discussed some of the findings. We conclude that the use of TestQ enables us to inspect the design of a test suite at a high level, as well as quickly identify test smell hot spots. 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.

Considering the architecture of the tool, we described how TestQ is realized as an extension of the pipes and filters architecture of Fetch and a customization of the Guess graph exploration environment. This architecture was perceived by the authors as flexible to extend and suited for experimentation with the meta-model, the metrics, visualizations and tool composition. The robustness of Fetch paid of in several past industrial case studies. The lack of integration in development environments and the poor overall

performance make it unlikely that TestQ , in its current setting, proves to be useful in rapid code-test-refactor cycles.

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] Stefan Reichart. Assessing test quality. Master's thesis, Universitat Bern, 2007.

[5] Helmut Neukirchen and Martin Bisanz. Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites. In *Proceedings of the 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES 2007). Lecture Notes in Computer Science*, pages 228–243. Springer, Heidelberg, June 2007.

[6] 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.

[7] Hira Agrawal, James L. Alberi, Joseph R. Horgan, J. Jenny Li, Saul London, W. Eric Wong, Sudipto Ghosh, and Norman Wilde. Mining system tests to aid software maintenance. *Computer*, 31(7):64–73, 1998.

[8] Markus Gaelli, Michele Lanza, and Oscar Nierstrasz. Towards a taxonomy of SUnit tests. In *Proceedings of 13th International Smalltalk Conference (ISC'05)*, September 2005.

[9] Joris Van Geet and Andy Zaidman. A lightweight approach to determining the adequacy of tests as documentation. In *Proc. of the 2nd Workshop on Program Comprehension through Dynamic Analysis*, pages 21–26, October 2006.

[10] Bas Cornelissen, Arie van Deursen, Leon Moonen, and Andy Zaidman. Visualizing testsuites to aid in software understanding. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR'07)*, 2007.

[11] Sebastian Elbaum, David Gable, and Gregg Rothermel. The impact of software evolution on code coverage information. In *Proceedings of the 17th International Conference on Software Maintenance*, pages 170–179, 2001.

[12] A. van Deursen and Leon Moonen. The video store revisited – thoughts on refactoring and testing. In *Proceedings of the 2nd eXtreme Programming and Flexible Processes Conference*, pages 71–76, 2002.

[13] M. Fewster and D. Graham. *Software Test Automation*, chapter 7: Building maintainable tests. ACM Press, 1999.

[14] C. Beust and H. Suleiman. *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley, 2007.

[15] Adrian Kuhn, Bart Van Rompaey, Lea Hänsenberger, Oscar Nierstrasz, Serge Demeyer, Markus Gaelli, and Koenraad Van Leemput. Jexample: Exploiting dependencies between tests to improve defect localization. In *Proceedings of 9th International Conference on Agile Processes and eXtreme Programming in Software Engineering (XP 2008)*, pages 72–83. Springer.

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

[17] 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.

[18] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. FAMIX 2.1 - the FAMOOS information exchange model. Technical report, University of Bern, 2001.

[19] 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.

[20] Eytan Adar. Guess: A language and interface for graph exploration. In *Proceedings of CHI*, 2006.
[21] Susan Elliott Sim, Richard C. Holt, and Steve Easterbrook. On using a benchmark to evaluate c++ extractors. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, page 114, Washington, DC, USA, 2002. IEEE Computer Society.
[22] A.K. Frick, H. Mehldau, and A. Ludwig. A fast adaptive layout algorithm for undirected graphs. In *Proceedings of Graph Drawing '94, LNCS 894*, pages 388–403. Springer, 1994.
[23] Oscar Nierstrasz, Stphane Ducasse, and Tudor Grba. he story of moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, 2005.
[24] Tim Littlefair. *An Investigation into the Use of Software Code Metrics in the Industrial Software Development Environment*. PhD thesis, Faculty of communications, Health and Science, Edith Cowan University, June 2001.
[25] Susan Elliott Sim, Richard C. Holt, and Steve Easterbrook. On using a benchmark to evaluate c++ extractors. In *Proceedings of the Tenth International Workshop on Program Comprehension*, pages 114–123, 2002.
[26] Lionel C. Briand, John W. Daly, and Jurgen K. Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.
[27] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

Appendix A

This appendix lists formal definitions for the metrics used in this work. The notation is an extension of the framework for coupling measurement in object-oriented systems by Briand et al. [26], and the subsequent refinement for reasoning about xUnit tests by Van Rompaey et al. [6].

In the following a couple of symbols are, unless overridden, inherently bound:

- tc is a test case, ie $tc \in TC$
- tm is a test command, ie $tm \in TM$
- th is a test helper, ie $tm \in TH$
- te is a test command or helper, ie $te \in TM \cup TH$

Assertionless A test command is assertion-less if it does not invoke framework checker methods, either direct or indirect. These commands are useless and potentially misleading, thus should be avoided, tagged or at least enumerated. $TTH(tm)$ is the set of all test helpers invoked by command tm, directly or nested in other helpers. $TIM_c(tm)$ is the set of all framework checker method invocations in commmand tm, either directly or indirectly through test helpers.

$$TTH(tm) = \bigcup_{i=0}^{\infty} TH_i(tm)$$

$$TIM_c(tm) = IM_c(tm) \bigcup \bigcup_{t \in TTH(tm)} IM_c(t)$$

$$ALESS = \{ tm \mid TIM_c(tm) = \phi \}$$

Assertion Roulette High numbers of description-less checker invocations make for hard to read tests. In case of failure manual intervention and reruns may be required. These description-less assertions are counted for a test command and all its helpers. $TCFM$ is partitioned in a set containing checker methods with a description, and one without.

$$TFCM = TFCM_{descr} \cup TFCM_{nodescr}$$

$$TIM_{cnd}(te) = TIM_c(te) \cap TFCM_{nodescr}$$

$$n \in \mathbb{N}_0,\ AROUL(n) = \{\ te \mid\ \mid TIM_{cnd}(te) \mid \geq\ n\}$$

Duplicated Code Code clones in unit tests have a bad effect on maintainability, since modifications to the UUT may result in a multitude of changes. Duplication is considered a strong smell since regression testing is one of the main goal of automation. Duplicate statements should be refactored to setup, teardown or helper methods.

Detecting clones is accomplished by comparing the contents of (test) methods against one another. Each method gets partitioned in sequences of adjacent accesses and invocations. These accesses and invocations are identified on the type and declaration level. Common partitions between methods are reported. The minimum size of these reported partitions is configurable. We do not provide a formal definition for this smell, as the used formalism has no concept of ordering.

Eager Test This smell was described thoroughly in [6]; metrics included.

Empty Test Tests without an implementation serve no use. It might indicate that the test is commented out or on the contrary a stub that was never filled. The total number of invocations and accesses in a command is used here. When this total equals zero a test is flagged.

For Testers Only Methods only used by test code do not belong in the production class. One can move these methods to a subclass in test code. Detecting FTO can result in a fair share of false positives, eg when the UUT is a library. A modifiable white list WL of methods should be used.

$$WL = \{pm \in M(PROD) \mid pm\ is\ whitelisted\}$$

$$FTO = (M(PROD) \cap IM(TEST)) \setminus (WL \cup IM(PROD))$$

General Fixture This smell was described thoroughly in [6]; metrics included.

Indented Test Loops and conditionals break the linear character of a test, and might make it too complex. Who's going to test the test? To fight duplication Indented Test is flagged for commands and helpers separately.

COND(m) and LOOP(m) denote the sets of conditionals and loops used in the implementation of method m.

$$INDENT = \{te \mid COND(te) \cup LOOP(te) \neq \phi\}$$

Indirect Test Testing business logic through the presentation layer is an example of an Indirect Test. A test case should test its counterpart in the production code. However, pinpointing the 'tested class' is not trivial. Instead a heuristic based on the number of production types used (NPTU) is employed, a metric defined in [6].

$$n \in \mathbb{N}_0, INDIR(n) = \{tm \mid NPTU(tm) \geq n\}$$

Mystery Guest The use of external resources in unit tests is considered not done. It lowers a tests' documentational value. Also, the extra dependency might introduce subtle circumstantial failures. And last but not least, I/O operations such as file access or database connections have a negative effect on speed.

To make static detection feasible, the system should be taught about unwanted methods. Direct or indirect invocations of such blacklisted methods $\epsilon\ MYST$ in commands and helpers will be flagged. $IM_i(te)$ stands for the set of all invoked methods at level i of indirection in helper or command te.

$IM_0(te) = IM(te)$

$$i\epsilon\ \mathbb{N},\ IM_{i+1}(te) = \bigcup_{t\epsilon IM_i(te)} SIM(t) \cup PIM(t)$$

$$TIM(te) = \bigcup_{i=0}^{\infty} IM_i(te)$$

$$MYST = \{m\ \epsilon\ M(C) |\ m\ is\ blacklisted\}$$

$$MGUES = \{te \mid TIM(te)\ \cap\ MYST\ \neq\ \phi\}$$

Sensitive Equality Verification by dumping an object's characteristics to string is easy and fast. However by doing so a dependency on irrelevant details like formatting characters is created. Whenever the toString implementation changes, tests will start failing. Detecting this in Java code boils down to the usage of 'toString' in a test framework checker method, either nested or indirect. For other languages a method blacklist $SEBL$ is needed. As a heuristic for 'linked to a checker method', all invocations in a helper or command are taken into account.

$$SEBL = \{m\ \epsilon\ M(C) \mid m\ dumps\ to\ string\ and\ was\ blacklisted\}$$

$$SEQUAL = \{te |\ IM(te)\ \cap\ SEBL\ \neq\ \phi\}$$

Verbose Test Closely related to the Long Method code smell introduced by Fowler [27], Verbose Tests have a negative influence on readability. We use SLOC to address this smell.