Restructuring Unit Tests with TestSurgeon

Pablo Estefó

Department of Computer Science (DCC)
University of Chile, Santiago, Chile

1. CHALLENGES WHEN REFACTORING TESTS

Agile methodologies prohibit the addition of a software feature without carefully accompanying it with explicit tests [1]. In his seminal book, "Uncle Bob" [10] wrote

"The iteration between writing test cases and code is very rapid [...]. As a result, a very complete body of test cases grows along with the code."

In practice, new tests are typically added when new features are added. During the refactoring and cleaning phase, tests are frequently run to make sure that nothing is broken (*i.e.,* tests remain green).

The software engineering community has produced great techniques and methodologies to maintain the design of a software clean. Coding practices and design patterns are often mentioned. However, little pressure is put on a software engineer to keep the design and structure of unit tests clean [4].

As a consequence, tests are often duplicated and poorly structured. The reason stems from the fact that unit tests are rarely reused as a standard software component is indented to be. In practice, it appears that less effort is dedicated to have unit tests modular and extensible.

To illustrate the problem of such an observation, we describe a situation found in Roassal, an agile visualization engine.

1.1 Running Example

Roassal consists of 290 classes, in which 64 of them are unit tests (following the xUnit architecture); it also defines 2,201 methods, in which 554 are test methods. Each unit test is related to a particular feature of Roassal (*e.g.,* events, interaction, shapes, rendering). However, one of these unit tests appears to contain significantly more test methods than other unit tests. The unit test `ROMondrianViewBuilderTest` contains about 50% of all the test methods. Moreover, those tests are rather chaotic and do not appear to test one single feature.

A discussion with the authors of Roassal revealed that at the beginning of the development of Roassal, a large majority of the tests were contained in `ROMondrianViewBuilderTest`. Later on, other unit tests appeared and new test methods were properly located. However, `ROMondrianViewBuilderTest` was left as it is, without being restructured. The problem illustrated with Roassal is not anecdotic. A large body of research is dedicated to better structure unit tests [3, 7, 15, 2].

Although the problem is widely acknowledged, restructuring unit tests is still known to be challenging [2]. Current programming environments do not propose criteria or metrics to assess the similarity of tests, useful to guide their refactoring.

1.2 Unit Test Similarity

A criteria of similarity is essential to properly understand differences and commonalities within a group of tests. In particular, two similar tests may be redundant. This is the case if the two tests actually test the same part of the code or if a test is subsumed by another.

We informally define the notion of *similarly* between two tests if both tests are redundant in the way they test the application.

Similarity as code duplication. Our first choice to measure the redundancy between two tests is to statically compare the source code of these tests. The intuition is that if two tests have nearly the same definition, then they are likely to test the same code. As your experiment described below shows, this intuition simply does not hold in our case.

Tracking portion of duplicated source code is known to be effective at identifying opportunities for code refactoring [6]. In addition, many tools are available to efficiently compare and report clones in software source code [13].

In order to find out whether duplicated tests are effective to identify redundant tests, we run the following experiment on the Roassal application:

1. each test is associated with another test with the closest source code (*i.e.,* greatest amount of duplicated lines of code). We obtain a list of pairs (t_1, t_2).
2. we consider each pair $(f(t_1, t_2), g(t_1, t_2))$, were f computes the portion of duplicated code between t_1 and t_2 and g computes the amount of application methods that are executed by both t_1 and t_2.

Scatter-plotting the last group of pairs visually indicates whether a correlation between the f and g metrics exist

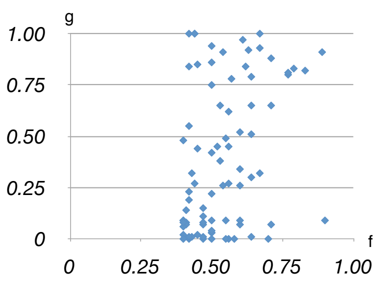

Figure 1: Comparing test code duplication with test coverage

(Figure 1). We find out that this last group of pairs *are not* linearly correlated. As a consequence, we deduce that two tests having a large portion of duplicated code do not necessarily test the same piece of code. Consider the example of the two test methods in Roassal `testModel` and `testNestedElements`. These tests are 10 and 12 lines of code long, respectively. Whereas 9 lines are identical between the two tests, only 30% of the application methods executed by these two tests are in common. This example reflects an overall tendency among Roassal's tests.

Since we have run this experiment on Roassal only, it may happen that a correlation between the duplication of test source codes and their coverage exist in other applications. However, the case of Roassal indicates that this is not the case in general. As a conclusion, code duplication between tests cannot be used to estimate if these tests are redundant. This small experiment makes a good case why comparing test has to be carried out using dynamic analysis.

Similarity as test coverage. We will compare test methods by reasoning on what they effectively test, which is measurable by profiling their execution. Test coverage [5] is an effective technique to identify which portion of a software is covered.

The traditional measure of test coverage [11, 12] relies on marking methods that are covered during the test execution. Whereas marking methods is useful to identify which portion of the code is actually covered or not by the tests, it poorly indicates whether the two tests are actually executing the same test scenario.

We compute two metrics for each covered method: (i) the number of times the method has been executed and (ii) the amount of different objects the method has been executed on. These two metrics capture an aspect of the computation that is relevant when comparing two executions. In addition, these metrics are technicaly cheap to compute, with a low overhead.

Each test is associated to the application methods the test execute and their respective metric values. We define the similarity of two tests t_1 and t_2 if the methods and their respective metric values are comparable. We use a visual support to estimate this similarity, as described in the next section.

2. TESTSURGEON

TestSurgeon is a profiler for unit tests. It monitors the execution of unit tests and collects data on what is being tested and how. TestSurgeon approximates the similarity between tests methods by offering a visual representation of their difference. TestSurgeon is intended to help software engineers structure and refactor their unit tests based on the difference of executions between analyzed test methods.

An analysis with TestSurgeon begins with a particular test method t_r. Test methods that are similar to t_r are first identified with a dedicated metric. Among the multitude of test methods that an application may have, determining which other tests are semantically close to t_r guides the software engineer on where to focus next.

Identifying similarities between test executions involves a significant amount of data. We use polymetric views [8], a visualization technique to let software reengineers visually identify particular patterns in a large amount of gathered data (Section 2.1).

Once a test t_b similar to t_r is selected, then our visual representation of the differences is used to affine the analysis and take further actions against one of the three scenarios we have identified (Section 2.2).

2.1 Test difference blueprint

Once two similar test methods are selected, a visual blueprint reveals the differences of execution between a blue t_b and a red t_r test methods. The blueprint is obtained by successively running t_b and t_r.

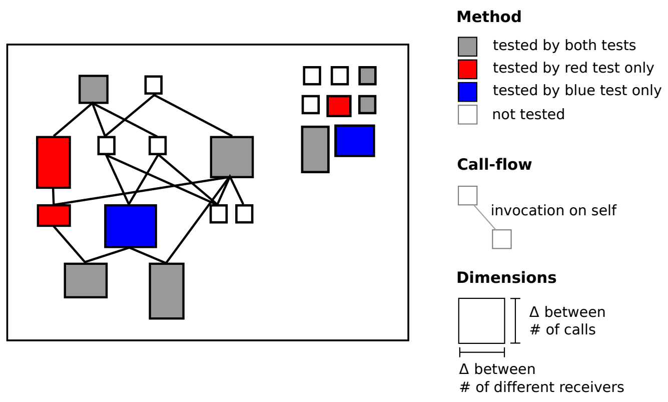

Figure 2: Test difference blueprint

The blueprint is a visual representation of the difference of the test executions in which each large box represents a class. Figure 2 shows one class that contains 20 methods. Inner boxes are methods. Edges between methods represent a call-flow on self variable, that means, upper method has a self invocation which leads to a call of lower method. Each method has a particular color and shape to indicate the difference of execution between the two tests. A blue box represents a method that is executed during the execution of t_b only. A red box means that the method is executed solely during the execution of t_r. A method that is executed by both t_r and t_b is painted in gray. Figure 2 shows that seven methods of the depicted class are executed by the two tests, three by the red test and two by the blue test.

The size of a method indicates whether the method has been commonly used by both tests: a small method means the method has been similarly executed by both test, and a large method means the method has been executed differently. More specifically, the height of a method is linear to the difference between the number of times the method is executed by both tests. The width is linear to the number of different object receivers that executed the method during the execution of t_r and t_b.

We have refactored a significant portion of Roassal unit tests. During our experience, we have identified four common scenarios that are typically involved when restructuring test methods.

2.2 Refactoring Scenarios

We have identified three scenarios in which identifying similar tests leads to an improvement of the test suite.

Defining fixture initialization. It frequently appears that each test defines its own test initialization before asserting some invariant. This initialization of the test corresponds to the code portion located before the first assertion.

In the case of Roassal, we have identified a significant proportion of test methods that uses a similar scenario. Consider the two test skeletons defined in the same test `UnitTest`:

```
UnitTest>>test1
   "Portion of code A"
   self assert: expr1

UnitTest>>test2
   "Portion of code B"
   self assert: expr2
```

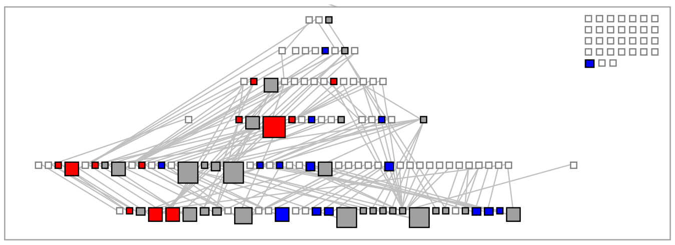

Figure 3: Gray methods are common in both tests

Figure 3 illustrates the situation by having the two portions of code in gray. The code contained in `expr1` and `expr2` are indicated with the red and blue color.

If the code portion A and B are semantically equivalent, then a `setUp` method can be defined and the two tests are shortened as follows:

```
UnitTest>>setUp
   "Portion of code A or B"

UnitTest>>test1
   self assert: ...

UnitTest>>test2
   self assert: ...
```

The execution that may be moved in a `setUp` method is indicated with the gray methods.

Moving misplaced test method. Sharing a significantly amount of common methods means that t_r and t_b are related. In the case (i) that t_r is defined in a unit test different than the one that defines t_b and (ii) t_r is not similar to other tests defined in its unit test, then t_r may be moved next to t_b, in the same unit test.

Appending or removing test methods. We have encountered cases where two tests are highly similar. In that case, the two tests may either be turned into one unique test or one of the two tests moved into a test suite that is occasionally exercised.

Consider a situation where two tests have a different source code but test the same base code.

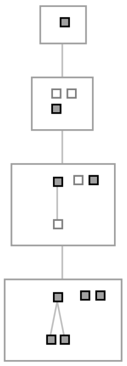

Figure 4: Minor difference between two tests

Figure 4 shows a class hierarchy composed of four classes that are tested by two similar tests. The edge between classes indicates class inheritance (a superclass is above its subclasses). All the covered methods are similarly executed by the two tests, shown by the presence of small gray boxes. Since both the red and blue tests actually test the same code portion, this situation indicates an opportunity for removing one test.

Another situation is when one test is superset of another.

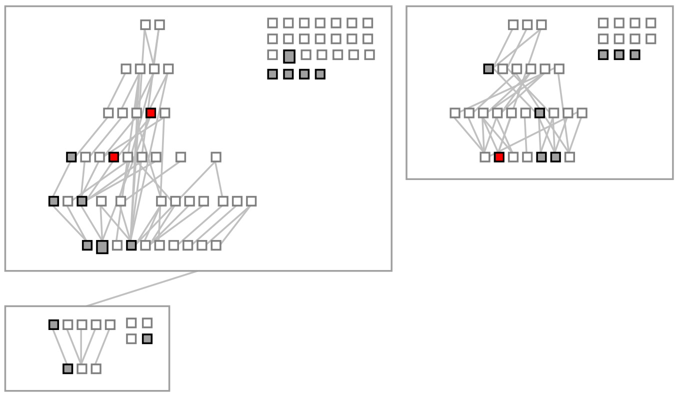

Figure 5: The red test is a superset of the the blue test

Graphically, this situation is represented with a large amount of gray squared methods accompanied with either some red or blue methods, but not both, as illustrated in Figure 5. The red test is a superset of the blue test by exercising slightly more methods of the base application.

3. RELATED WORK

This section briefly revises the literature related to test comparison and test visualization.

Test prioritization and clustering is a popular approach for handling the high cost of running large amount of tests. Shin Yoo *et al.* have presented a mechanism for reducing pair-wise comparison for clustering in order to ease the human task of prioritizing test. They cluster tests by grouping similar tests. Each test execution is represented by a binary-chain where each digit (1 or 0) represents if a certain method was performed during execution, then this coverage similarity is measured with a binary distance.

Vangala *et al.* [14] created a mechanism that uses static and dynamic analysis to optimize testing activities such as test selection, redundancy elimination and test prioritization. Nevertheless their metrics follows the same spirit of TestSurgeon, they are defined for a procedural programming context which is incompatible in an object-oriented environment.

Greiler *et al.* [4] address the problem of test suite understanding by proposing a Test Similarity Correlator framework. Their framework produces a trace execution which characterizes the test execution through three types of events: execution of a test method, set-up and tear-down event, method execution and exception thrown. Test suite understanding is supported by comparing test execution traces.

Tests code as a software artifact has multiple uses. One of them is as documentation [7]. Specially for new developers who face up legacy systems, tests are highly relevant as code examples. Following this perspective, Lienhard *et al.* [9] propose a tool which eases the task of writing new tests for those systems by providing a visualization called Test Blueprint as a guide for test understanding and maintenance.

4. CONCLUSION AND FUTURE WORK

Restructuring unit tests is a problem that remains largely unaddressed. Our work innovates in this field by providing an expressive and intuitive visualization to understand difference of test execution. TestSurgeon is a help for restructuring unit tests. It profiles the execution of unit tests and indicates how similar tests are.

As a future work, we plan to give a perspective to our current results by analyzing other software applications.

5. REFERENCES

[1] Kent Beck. *Test Driven Development: By Example.* Addison-Wesley Longman, 2002.

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

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

[4] Michaela Greiler, Arie van Deursen, and Andy Zaidman. Measuring test case similarity to support test suite understanding. In *Proceedings of the 50th international conference on Objects, Models, Components, Patterns*, TOOLS'12, pages 91–107, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] Susan Horwitz. Tool support for improving test coverage. In *Proceedings of the 11th European Symposium on Programming Languages and Systems*, ESOP '02, pages 162–177, London, UK, UK, 2002. Springer-Verlag.

[6] Rainer Koschke. Large-scale inter-system clone detection using suffix trees and hashing. *Journal of Software: Evolution and Process*, 2013. accepted for publication.

[7] 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 P. Abrahamsson, editor, *Extreme Programming and Agile Processes in Software Engineering, 9th International Conference, XP 2008*, Lecture Notes in Computer Science, pages 73–82. Springer, 2008.

[8] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[9] Adrian Lienhard, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz. Test blueprints — exposing side effects in execution traces to support writing unit tests. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08)*, pages 83–92. IEEE Computer Society Press, 2008.

[10] Robert Cecil Martin. *Agile Software Development. Principles, Patterns, and Practices.* Prentice-Hall, 2002.

[11] Audris Mockus, Nachiappan Nagappan, and Trung T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 291–301, Washington, DC, USA, 2009. IEEE Computer Society.

[12] Paul Piwowarski, Mitsuru Ohba, and Joe Caruso. Coverage measurement experience during function test. In *Proceedings of the 15th international conference on Software Engineering*, ICSE '93, pages 287–301, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[13] Matthias Rieger, Stéphane Ducasse, and Michele Lanza. Insights into system-wide code duplication. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 100–109. IEEE Computer Society Press, November 2004.

[14] Vipindeep Vangala, Jacek Czerwonka, and Phani Talluri. Test case comparison and clustering using program profiles and static execution. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 293–294, New York, NY, USA, 2009. ACM.

[15] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production and test code. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 220 –229, April 2008.