Towards a Taxonomy of SUnit Tests [⋆]

Markus Gälli [a] Michele Lanza [b] Oscar Nierstrasz [a]

[a] *Software Composition Group*
Institut für Informatik und angewandte Mathematik
Universität Bern, Switzerland

[b] *Faculty of Informatics*
University of Lugano, Switzerland

Abstract

Although unit testing has gained popularity in recent years, the style and granularity of individual unit tests may vary wildly. This can make it difficult for a developer to understand which methods are tested by which tests, to what degree they are tested, what to take into account while refactoring code and tests, and to assess the value of an existing test. We have manually categorized the test base of an existing object-oriented system in order to derive a first taxonomy of unit tests. We have then developed some simple tools to semi-automatically categorize tests according to this taxonomy, and applied these tools to two case studies. As it turns out, the vast majority of unit tests focus on a single method, which should make it easier to associate tests more tightly to the methods under test. In this paper we motivate and present our taxonomy, we describe the results of our case studies, and we present our approach to semi-automatic unit test categorization.

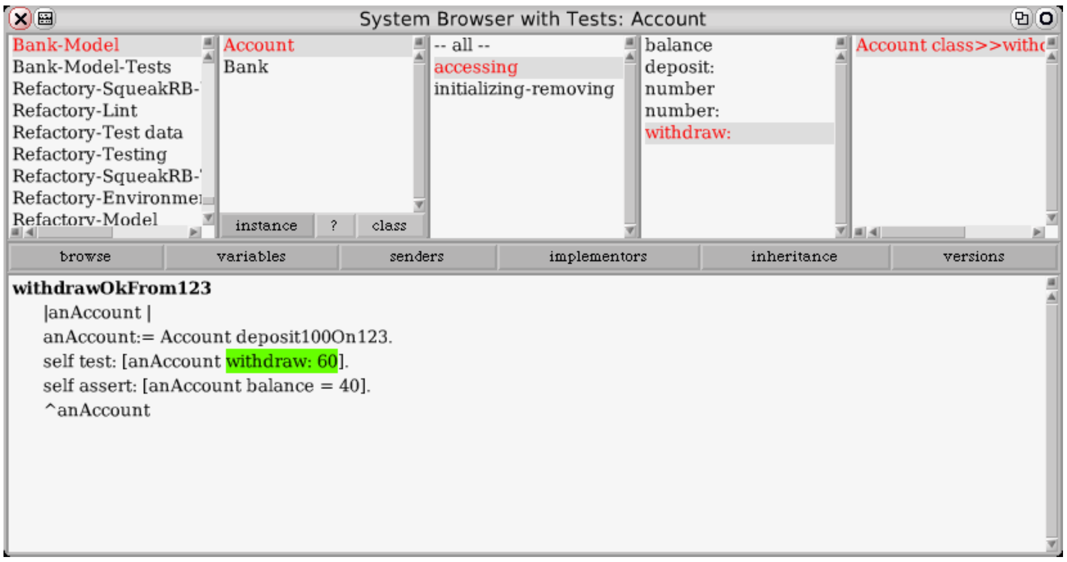
Key words: unit testing, taxonomy, reverse engineering

1 Introduction

XUnit [1] in its various forms (JUnit for Java, SUnit for Smalltalk, etc.) is a widely-used open-source unit testing framework. It has been ported to most

[⋆] We thank Stéphane Ducasse for his helpful comments and gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02).

Email addresses: `gaelli@iam.unibe.ch` (Markus Gälli), `michele.lanza@unisi.ch` (Michele Lanza), `oscar@unibe.ch` (Oscar Nierstrasz).



```
withdrawOkFrom123
    |anAccount |
    anAccount:= Account deposit100On123.
    self test: [anAccount withdraw: 60].
    self assert: [anAccount balance = 40].
    ^anAccount
```

Fig. 1. An enhanced class browser shows methods and their one-method tests site by site. Note that the test returns its result, thus enabling other unit tests to reuse it. We thus store tests like other factory methods on the class site.

object-oriented programming languages and is integrated in many common IDEs such as Eclipse.

Although these development environments help developers to navigate between related methods in a complex software system, they offer only limited help in relating methods and the unit tests that test them.

Our hypothesis is that a majority of unit tests focus on single methods. We call these dedicated unit tests *one-method commands*. If our hypothesis is valid, then we could help the developer in several ways to write and evolve methods together with their tests:

- *Tighter integration of tests and methods in class browsers.* Each *one-method command* could be displayed close to its method, and document a quality-approved usage of the method. (See Figure 1) It then would be also clear if a method has a *dedicated* test case or not. The developer would not have to switch windows for developing tests or methods as they could be naturally displayed site by site.
- *Test case selection.* All *one-method commands* could be executed as soon as their focused method has been changed.
- *Concrete Typing.* The set of tested concrete types of the receiver, parameters and result of the method under test are deducible by executing an instrumented version of its *one-method commands*. Thus *one-method commands* remove the burden of a test-first-driven development of providing the types in a statically typed language or deducing them in a dynamically typed language.
- *Test case refactoring.* If a method is deleted, its corresponding test method

could be deleted immediately too. Renaming a method would not break the brittle naming convention anymore, which is currently the only link between a method and its unit tests. Adding a parameter to a method could be automatically mirrored by adding a factory to its according test [1].

In order to validate our hypothesis we have:

- Developed an initial taxonomy of unit tests by carrying out an empirical study of a substantial collection of tests produced by a community of developers.
- Implemented some lightweight tools to automatically classify certain tests into categories offered by the taxonomy.
- Conducted case studies to validate the generality of the taxonomy.

Our manual experiment supports the hypothesis that a significant portion of test cases have an implicit one-to-one relationship to a method under test or are decomposable into *one-method commands*. Although it is difficult to identify a general algorithm to distinguish this kind of test, our initial heuristics to automate this endeavor succeed in identifying 50% of one-to-one tests without resulting in any false positives.

Structure of the article. In Section 2 we define some basic terms. In Section 3 we present the taxonomy derived from our manual case study. In Section 4 we describe some simple heuristics for mapping unit tests to the taxonomy, and we describe the results of applying these heuristics to two case studies. In Section 5 we discuss some of the problems and difficulties encountered. Section 6 briefly outlines related work. In Section 7 we conclude and outline future work.

2 Basic Definitions

We first introduce some basic terminology, on which our taxonomy builds on.

Assertion: An *assertion* is a method that evaluates a (side-effect free) Boolean expression, and throws an exception if the assertion fails. Unit test assertions usually focus on specific instances whereas assertions of *Design By Contract* are used in post-conditions and are more general.

Package: We assume the existence of a mechanism for grouping and naming a set of classes and methods. In the case of Java this would be packages; in the

[1] Further refactorings [2], which have to be carried out in parallel for the test code and the code under test would be easier too, but this is subject to further research.

case of Smalltalk we use class categories as the smallest common denominator of several Smalltalk dialects. We call these groups *packages*.

Command: Every XUnit Test is a *command* [3], which is a parameter-free method whose receiver can be automatically created. The XUnit Test can thus be automatically executed.

The command receiver in the case of a XUnit test case can be constructed automatically, *e.g.,* `new MyTestCase(myTestSelector)`. The whole command then looks like:

```
(new MyTestCase(myTestSelector)).run()
```

Test package: A *test package* is a package which includes a set of commands.

Package under Test: If a test package tests another package, we call this other package the *package under test*, which may be identified either implicitly by means of naming conventions, or explicitly by means of a dependency declaration.

Candidate method: A *candidate method* is a method of the package under test.

Focuses on one method: We say that a command *focuses on one method*, if it tests the result or side effects of *one* specific method and not the result or side effects of several methods.

3 A Taxonomy of Unit Tests

Initial case study. We derived the taxonomy by manually categorizing 982 unit tests of the Squeak [4] base system [2]. Squeak is a feature-rich, open source implementation of the Smalltalk programming language written in itself and by many developers. It includes network- and 2D/3D-graphics support, an integrated development environment, and a constructivist learning environment for children.

The tests were written by at least 26 different developers. One of the test developers developed 36% of the test cases, two more developed a further 34%, and yet another six developers produced another 19% of tests. Each of the other developers produced less than 3% of the tests. We defined the taxonomy depicted in Figure 2 by iteratively grouping tests into categories and refining the classification criteria. Our manual categorization yielded a distribution of the categories shown in Figure 3.

[2] Version 3.7 beta update 5878, available at `http://www.squeak.org`

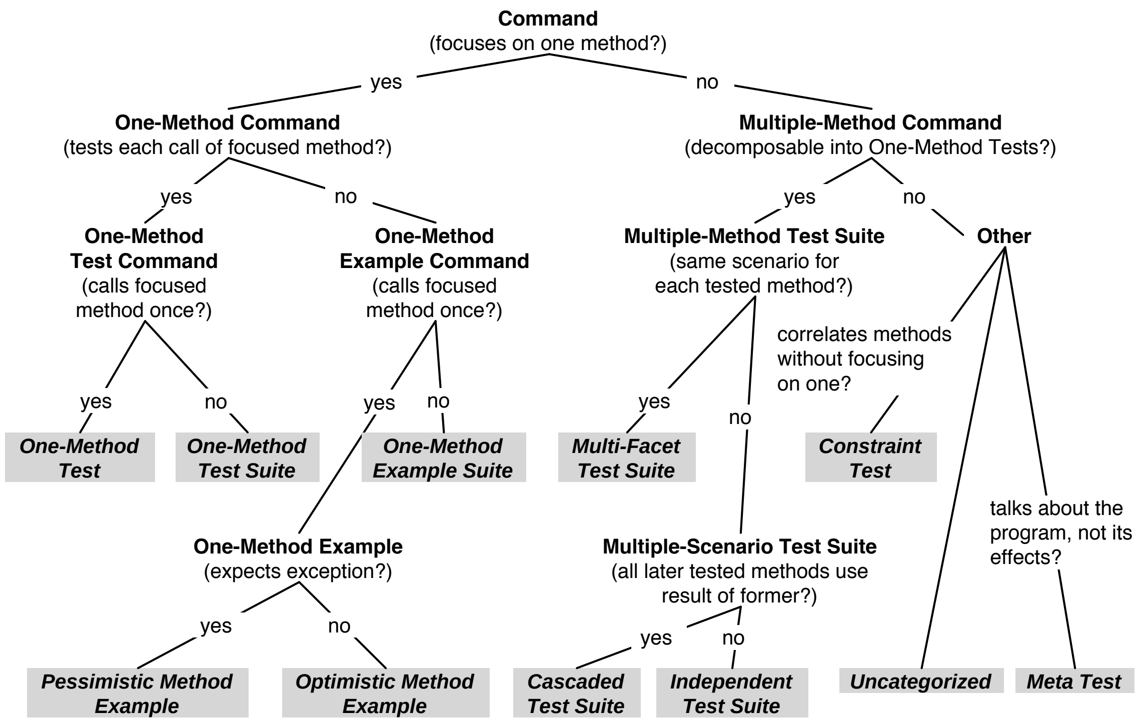

Command
(focuses on one method?)
yes
no
One-Method Command
(tests each call of focused method?)
Multiple-Method Command
(decomposable into One-Method Tests?)
yes
no
yes
no
One-Method Test Command
(calls focused method once?)
One-Method Example Command
(calls focused method once?)
Multiple-Method Test Suite
(same scenario for each tested method?)
Other
correlates methods without focusing on one?
yes
no
yes
no
yes
no
One-Method Test
One-Method Test Suite
One-Method Example Suite
Multi-Facet Test Suite
Constraint Test
talks about the program, not its effects?
One-Method Example
(expects exception?)
Multiple-Scenario Test Suite
(all later tested methods use result of former?)
yes
no
yes
no
Pessimistic Method Example
Optimistic Method Example
Cascaded Test Suite
Independent Test Suite
Uncategorized
Meta Test

Fig. 2. Taxonomy of unit tests. Nodes are gray and denote concrete occurrences of unit tests.

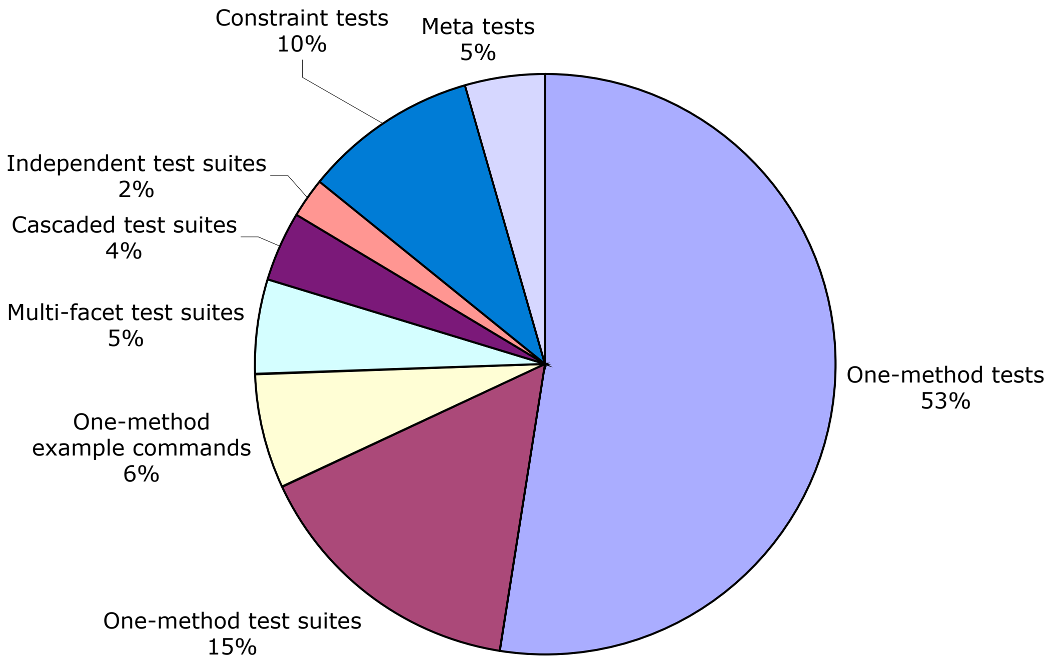


Fig. 3. Manual classification of unit tests for the base Squeak system

We now describe and motivate each of the unit test categories in the taxonomy. For each node of our taxonomy we present a real world example found in the

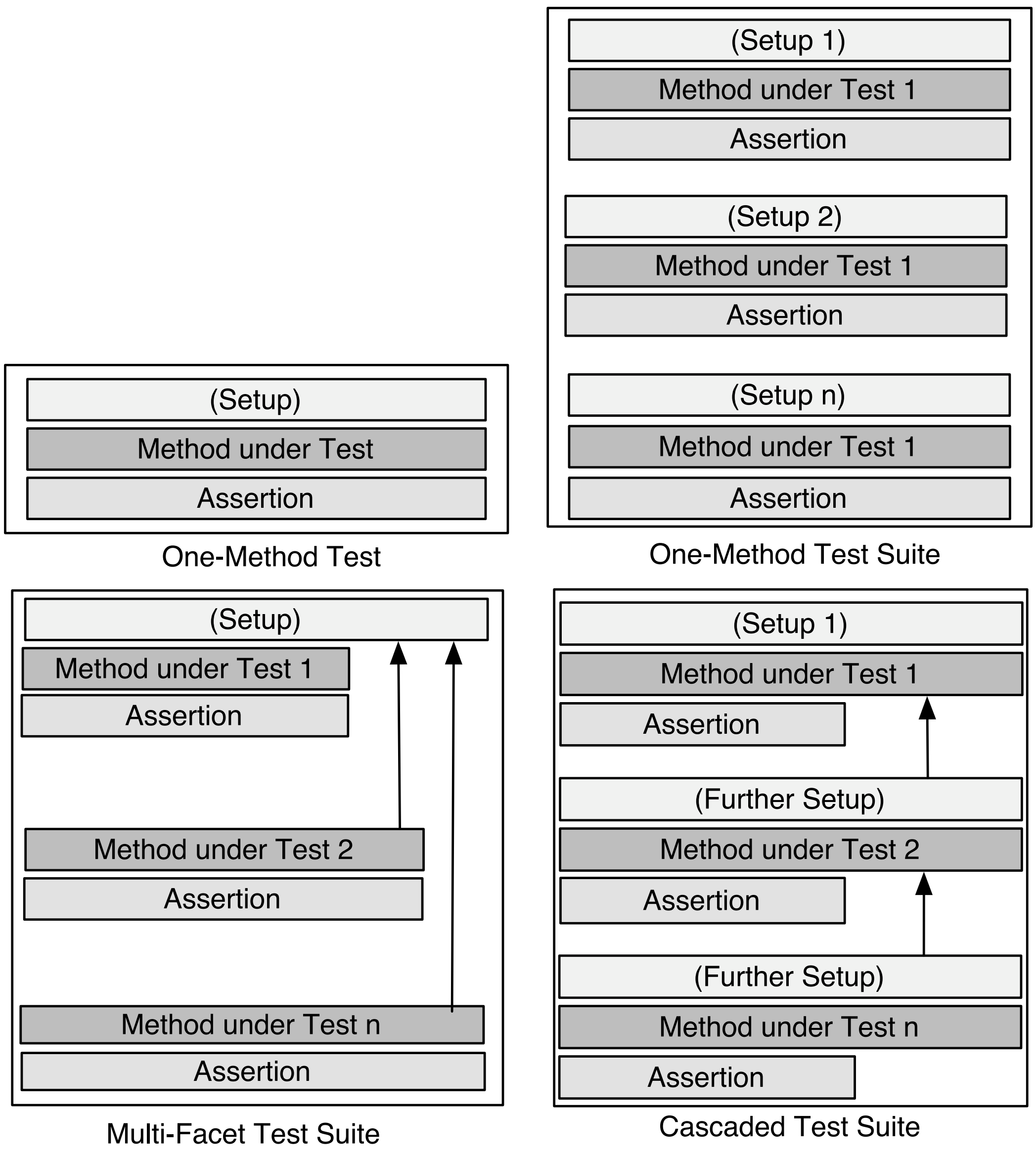


Fig. 4. One-method test suites, multi-facet test suites and cascaded test-suites are decomposable into one-method tests.

Squeak unit tests[3].

We divide our taxonomy tree into two subtrees (Figure 2): (1) *One-method commands*, which are *commands* that focus on single methods, and (2) *multiple-method commands*, which do not focus on a single method. We divided each of these subtrees into two further subtrees, which we will present in the following subsections.

[3] For a short introduction to the Smalltalk syntax see the appendix.

3.1 One-method test commands

A *one-method test comand* is a *one-method command* which has assertions testing the outcome of *each call* of the method under test.

3.1.1 One-method tests

If it tests the outcome of exactly one call of a method under test, we call it a *one-method test*. In the example below the method Week class≫indexOfDay: would be the method under test, and only called once:

```
YearMonthWeekTest≫testIndexOfDay
  self assert: (Week indexOfDay: 'Friday') = 6.
```

3.1.2 One-method test suites

On the other hand a *one-method test suite* tests the outcome of the method under test in several situations:

```
YearMonthWeekTest≫testDaysInMonth
  self assert:  (Month daysInMonth: 2 forYear: 2000) = 29.
  self assert:  (Month daysInMonth: 2 forYear: 2001) = 28.
  self assert:  (Month daysInMonth: 2 forYear: 2004) = 29.
  self assert:  (Month daysInMonth: 2 forYear: 2100) = 28.
```

3.2 One-method example commands

A *one-method example command* is a *one-method command* which does not have assertions for the method under test. So this command does not test the focused method against some desired result, but merely calls it. We detected three concrete instances of these commands:

3.2.1 Pessimistic one-method examples

A *pessimistic method example* is a *one-method example* which checks that an exception is thrown if a method is called in a way which violates a precondition. Beck [5] calls *pessimistic one-method examples “exception tests”*. Here is an example of a *pessimistic one-method example* ensuring that an attempt to create the directory C: on a Windows platform should fail:

```
DosFileDirectoryTests≫testFileDirectoryNonExistence
  "Hoping that you have 'C:' of course..."
  FileDirectory activeDirectoryClass == DosFileDirectory ifFalse:[^self].
  self
    should: [(FileDirectory basicNew fileOrDirectoryExists: 'C:')]
  raise: InvalidDirectoryError.
```

Note that we consider neither shouldnt: raise: nor should: raise: as assertions, because they do test whether something is true or false in a given state, but merely check whether or not an exception is thrown.

3.2.2 Optimistic method examples

An *optimistic method example* is a *one-method example* which expects that no exception is thrown if the method under test is called without violating some preconditions. Again, *optimistic method examples* do not contain *assertions.* The unit test below tests that the invocation of copyBits on a BitBlt in a certain situation does not throw an exception:

```
BitBLTClipBugs≫testDrawingWayOutside2
  | f1 bb f2 |
  f1 := Form extent: 100@100 depth: 1.
  f2 := Form extent: 100@100 depth: 1.
  bb := BitBlt toForm: f1.
  bb combinationRule: 3.
  bb sourceForm: f2.
  bb destOrigin: 0@0.
  bb width: SmallInteger maxVal squared; height: SmallInteger maxVal squared.
  self shouldnt:[bb copyBits] raise: Error.
```

3.2.3 One-method example suites

A *one-method example suite* is a *one-method example command* which calls the method under test more than once. It can be decomposed into several one-method command which call the same focused method once:

```
FractionTest≫testDegreeSin
  self shouldnt: [ (4/3) degreeSin] raise: Error.
  self assert: (1/3) degreeSin printString = '0.005817731354993834'
```

3.3 Multiple-method test suite

A *multiple-method test suite* is a *multiple-method command* which is decomposable into one-method tests. (See Figure 4).

3.3.1 Multi-facet test suites

Multi-facet test suites are *multiple-method test suites* that reuse a scenario to test several candidate methods. In the following example a previously initialized variable time is used to check different methods on Time.

```
TimeTest≫testPrinting
  self
    assert: time printString = '4:02:47 am';
    assert: time intervalString = '4 hours 2 minutes 47 seconds';
    assert: time print24 = '04:02:47';
    assert: time printMinutes = '4:02 am';
    assert: time hhmm24 = '0402'.
```

3.3.2 Cascaded test suites

Cascaded test suites are *multiple-scenario test suites* in which the results of one test are used to perform the next test:

```
Base64MimeConverterTest≫testMimeEncodeDecode
  | encoded |
  encoded ˙ Base64MimeConverter mimeEncode: message.
  self should: [encoded contents = 'SGkgVGhlcmUh'].
  self should:
    [(Base64MimeConverter mimeDecodeToChars: encoded) contents
      = message contents].
```

This *cascaded test suite* first triggers a method Base64MimeConverter≫mimeEncode:, tests its result encoded, and then uses encoded to test Base64MimeConverter≫mimeDecodeToChars:.

3.3.3 Independent test suite

An *independent test suite* is a *multiple-scenario test suite* which tests different methods on different receivers not depending on each other.

In the following example several independent methods are tested:

IslandVMTweaksTestCase≫replaceIn:from:to:with:startingAt: needs a totally different set of parameters than say

IslandVMTweaksTestCase≫nextInstanceAfter: [4]

```
IslandVMTweaksTestCase≫testForgivingPrims
| aPoint anotherPoint array1 array2 |
aPoint := Point x: 5 y: 6.
anotherPoint := Point x: 7 y: 8. "make sure there are multiple points floating around"
anotherPoint. "stop the compiler complaining about no uses"

self should: [ (self classOf: aPoint) = Point ].
self should: [ (self instVarOf: aPoint at: 1) = 5 ].
self instVarOf: aPoint at: 2 put: 10.
self should: [ (self instVarOf: aPoint at: 2) = 10 ].

self someObject.
self nextObjectAfter: aPoint.

self should: [ (self someInstanceOf: Point) class = Point ].
self should: [ (self nextInstanceAfter: aPoint) class = Point ].
```

[4] Actually these tests are calling primitives, which are implemented in the virtual machine and not in the smalltalk image.

```
array1 := Array with: 1 with: 2 with: 3.
array2 := Array with: 4 with: 5 with: 6.

self replaceIn: array1 from: 2 to: 3 with: array2 startingAt: 1.
self should: [ array1 = #(1 4 5) ].
```

3.4 Others

We call all test cases which neither focus on one method nor are decomposable into one-method tests *others*.

3.4.1 Constraint test

A *constraint test* checks the interplay of several methods without focusing on one of them. In the following example a graphic conversion functionality is tested by comparing the original bitmap with the result obtained after encoding the bitmap to the png-format and then decoding it back again.

```
PNGReadWriterTest≫test16Bit
  self encodeAndDecodeForm: (self drawStuffOn: (Form extent: 33@33 depth: 16))
```

3.4.2 Meta test

A *meta test* is a test about the application itself, *e.g.,* its structure, its current state or its implemented or unimplemented methods. For example, the following test checks if the class of Metaclass only has one instance, namely Metaclass:

```
BCCMTest≫test07bmetaclassPointOfCircularity
  self assert: Metaclass class instanceCount = 1.
  self assert: Metaclass class someInstance == Metaclass.
```

3.4.3 Uncategorized

We call all unit tests which do not fall into one of the above categories *uncategorized.*

3.5 First validation: Maven

Using our taxonomy, we manually categorized 50 randomly selected JUnit tests of *Maven* [6], a Java project management and project comprehension[5].

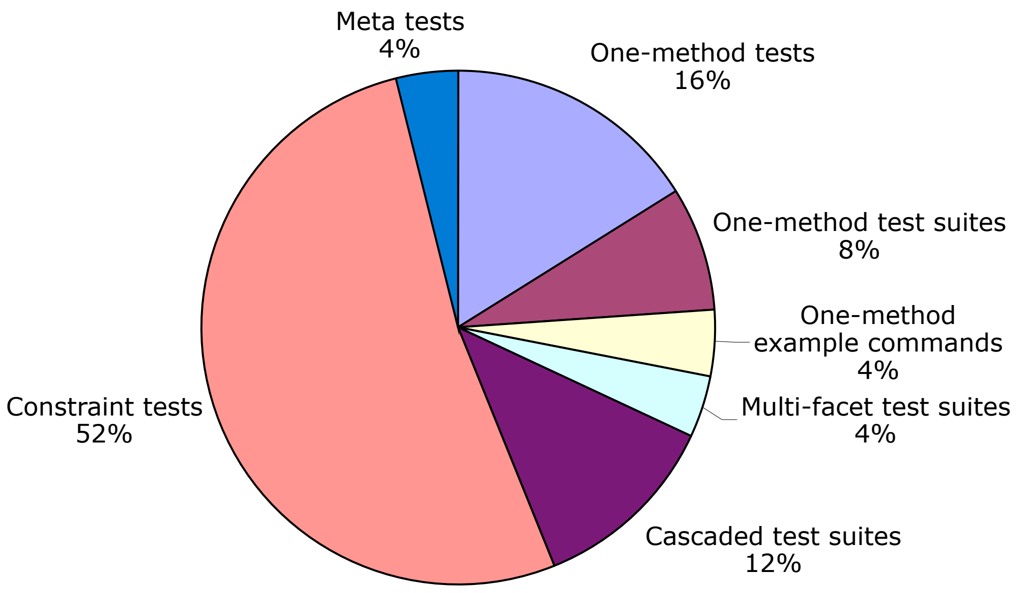


Fig. 5. Manual classification of 50 random unit tests of Maven

25 of these tests merely checked some getter/setter code and were classified as constraint tests. The other sampled tests fell naturally into one of our proposed categories, and if less trivial getter/setter test code had been selected, we could expect again *one-method commands* as the majority of classified tests (See Figure 5).

4 Automatic Classification of Unit Tests

After having manually derived the taxonomy, we developed some lightweight heuristics to automatically detect the feature properties depicted in Figure 2. Our goal is to classify most of the unit tests automatically. Using these heuristics we have been able to automatically classify 52% of the manually classified one-method commands tests, while our average precision rate was 89% (see Table 1). Finally we applied our automatic approach to a new case study and found that more than a third of the unit tests focus on single methods.

[5] See http://www.iam.unibe.ch/~gaelli/mavenUnitTests.html

4.1 Instrumentation

To detect the feature properties we rely on dynamic analysis of the code, as we are dealing with runnable test cases in a dynamically typed environment.

Many of the unit tests of the Squeak base system test low level classes like Arrays *etc.* It is therefore not feasible to use method wrappers [7], because recursion would almost certainly arise when the wrapping algorithm uses a method which is about to be wrapped — thereby bringing our system to a halt. We therefore used the *bytecode interpreter* found in the class ContextPart, which is also used in the debugger of Squeak to step and send through methods.

Using and enhancing the bytecode interpreter of Squeak has the advantage of being more general than *method wrappers* and base level classes can be tested too. However, it comes with the following disadvantages:

- It is slower than current VM optimized method wrapper code.
- Simulation of exception handling code is buggy in the current implementation in the SqueakVM: As a consequence it did not work for exception handling code used by mainly by optimistic or pessimistic method examples.
- Methods which only return a variable are inlined by the Smalltalk-compiler and thus cannot be detected[6].

4.2 Lightweight Heuristics

In the following we present a list of heuristics used to detect the feature properties displayed in the left subtree of the Figure 2. We have not yet developed any heuristics to classify leaves of the right subtree.

The first question in the decision tree is whether a unit test focuses on a single method. Three possible ways to detect this property are:

(1) *Deduction of the focused method from the command name.* One approach to deduce if a command focuses on one method is to examine the method name of the command. Often the developer includes the name of the method under test as part of the test method. A typical unit test looks like FooTest≫testBar which denotes that a method named bar of the class named Foo is tested and thus focused on. The execution of the test method can be simulated with our bytecode interpreter and thus checked, if it calls directly a method of the form Foo≫bar or Foo≫bar:.

[6] On the other hand this might be a welcome side effect as one would normally not focus a test on a method that merely returns a variable.

If the naming convention of the test method name can be decoded and exactly one candidate method matches, then the developer has clearly indicated that this would be the method under focus. More specifically we deleted the first four characters "test" of the command name, and searched for a selector in the trace in the first level, that matches the remaining string, possibly converting the leading character to lower case, and ignoring parameters.

Example: If the test method name is BarTest≫testFoo then we look for an event in which a candidate method foo is called. If there are two selectors called, like foo: and foo, the result is ambiguous and we cannot say on which of them our test would focus.

(2) *Deduction of the focused method by the command structure.* We say that the command focuses on this method, if exactly one candidate method is called directly: A simple way to detect if a unit test focuses on one method is to find out if the test method only calls one candidate method, that is only one method of the package under test. This approach cannot be complete, as many unit tests do the setup of the test scenario not in the extra TestCase≫setUp method, but in the test method itself, and there they often have to call methods of the package under test for the setup. We do not make a distinction whether a candidate method is called only once or more than once, as long as it is the only called candidate method.

(3) *Deduction of the focused method by using historical information.* In incremental test-driven approaches the less complex methods will be built before the more complex ones. To test a more complex method the developer will likely refer to simpler candidate methods, either to build the scenario on which the complex method can be run or to use already existing methods as test oracles. However, in Squeak we do not know if a test case was developed before another test case, as Squeak still relies on a code exchange mechanism which destroys this versioning information.

To determine if a *one-method command* is a *one-method test command* or a *one-method example command* we check if it only calls self should: [] raise: Exception, self shouldnt: [] raise: Exception or friends, and if all the expressions inside the "shoulds" call the same method.

We can distinguish *one-method tests* from *one-method test suites* by simply counting how often the method under test is called. Accordingly we do the further split up in the right subtree, the *one-method example command* and then use the difference between the calls should:raise: and shouldnt:raise: to make the last distinction. With this heuristic we classify any *one-method test* as *one-method test command* which does not call any kind of should:raise: and shouldnt:raise:.

Category	**Manual result**	**Computed Result**	**Hits**	**Recall**	**Precision**
One-method tests	387	207	202	52%	98%
One-method test suites	114	86	57	50%	66%
Pessimistic method examples	11	15	10	91%	66%
Optimistic method examples	15	16	10	67%	63%
One-method example suites	10	1	1	10%	100%
Total	537	334	280	52%	89%

Table 1
Preliminary manual and automatic classifications of one-method commands of the Squeak Unit Tests.

Category	**Manual result**	**Computed Result**	**Hits**	**Recall**	**Precision**
One-method tests	59	19	5	8%	26%
One-method test suites	80	48	37	46%	77%
One method example suites	3	3	3	100%	100%
Total	142	70	45	32%	64%

Table 2
Preliminary manual and automatic classifications of one-method commands of the SmallWiki Unit Tests.

4.3 A First Case Study: Squeak Unit Tests

Having categorized the Squeak Unit Tests before, we could compare the results of our lightweight heuristic with our manual results. (See Table 1). Squeak 3.7 has no notion of packages and relies on a naming convention of class-categories. We only automatically categorized 671 of 982 tests, whose class-category name allowed us to identify their package under test. Our heuristics were able to categorize 52% of the leaves of the left subtree from our taxonomy with a mean precision of 89%, meaning that only 11% of the categorized test cases were put in a different category than by the human reengineer.

4.4 *A Second Case Study: SmallWiki*

After having done a manual categorization (see Figure 6) we automatically categorized the 200 unit tests of *SmallWiki* [8], a collaborative content management tool written in VisualWorks Smalltalk and ported to Squeak. We chose this system as a case study, as it is a medium sized application developed by a single experienced developer in a test-driven way.

A surprising result here was that more tests could be detected as focusing on one method by considering the calls of only one candidate method, rather than by exploiting their naming convention.

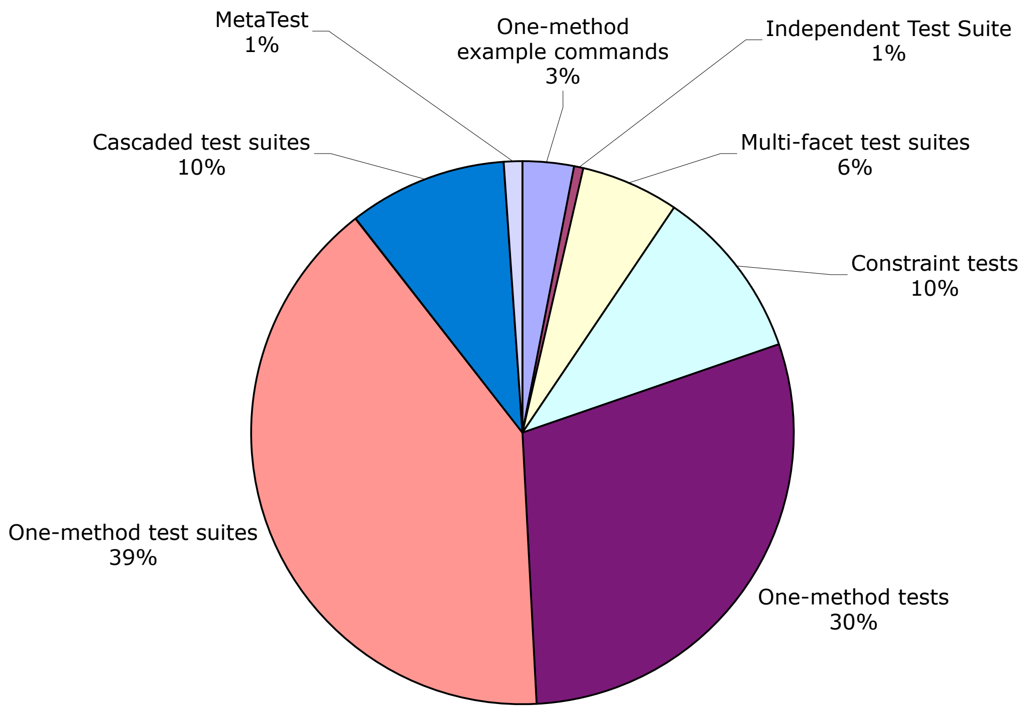


Fig. 6. Manual classification of unit tests for the SmallWiki system

We only programmed the detection for three categories, namely *one-method tests*, *one-method test suites*, and *one-method example suites*. All of them together represented already more than a third of all tests. Figure 6 shows that contrary to the Squeak case study, the developers here wrote more *one-method test suites* than *one-method tests*. The recall and precision for *one-method tests* displayed in Table 2 is only 5% respectively 26% as there have been many tests for getter/setter pairs: The getter-methods of variables are inlined and could thus not be detected by our bytecode interpreter. Only setter methods have been detected leading to false positives.

5 Discussion

Although the taxonomy we have derived appears promising, it is a preliminary result for several reasons:

- Our taxonomy is based on only three case studies. Though it seldom arises that we discover new categories, more case studies need to be conducted.
- We focused on XUnit Tests, as described by Beck et al.[1] so we do not know if developers write other kinds of unit tests while using other testing frameworks.
- We have not addressed the question if unit tests should be considered white-box or blackbox-tests and if they could likewise be used as acceptance, integration, or end-to-end tests.
- Only three of the Squeak Unit Test developers wrote 70% of the test cases making our sample data of this case study less representative.

Developers have complete freedom to write any kind of unit tests — making automatic classification a difficult business. The automatic classification heuristics are similarly preliminary and may fail in the following cases:

*.Ambiguity of the naming convention Using the naming convention for automatic detection of the method under test is unreliable and ambiguous. For example, does the following test focus on Foo≫bar:, on Foo≫bar, or both of them? A similar problem arises in Java, as the naming convention will not differentiate between overloaded methods that take different types of parameters.

```
FooTest≫testBar
  |aFoo|
  aFoo:= Foo new.
  aFoo bar: 1.
  self assert: (aFoo bar = 1)
```

We would manually categorize this one as a *constraint test.*

*.Test framework tests Tests of the test framework may be incorrectly categorized. The following test could be classified as a pessimistic method example of error: but its intent is to be an *optimistic* method example of should:raise:

```
SUnitTest≫testException
  self
    should: [self error: 'foo']
    raise: TestResult error
```

*.Assertions come only after clean up In some tests cleanups are necessary. As the cleanup does not have to influence the test result, developers also write the assertions after the cleanup.

In the following example both assertion statements could be moved two lines up preserving the test case. Thus it is activate and not wait or suspend which is tested.

```
StopwatchTest≫testMultipleTimings
 aStopwatch activate.
 aDelay wait.
 aStopwatch suspend.
 aStopwatch activate.
 aDelay wait.
 aStopwatch suspend.
 self assert: aStopwatch timespans size = 2.
 self assert:
  aStopwatch timespans first asDateAndTime <
  aStopwatch timespans last asDateAndTime
```

***.**Tested method is not the last called of the package under test Some tests are testing methods which are not the last method of the package called before the assertion occurred. Example: Is the method under test removeActionsWith-Receiver: or actionForEvent:? The name of the command indicates the former, but the structure of the test suggests the latter:

```
EventManagerTest≫testRemoveActionsWithReceiver
 | action |
 eventSource
  when: #anEvent
  send: #size to: eventListener;
  when: #anEvent
  send: #getTrue to: self;
  when: #anEvent:
  send: #fizzbin to: self.
 eventSource removeActionsWithReceiver: self.
 action := eventSource actionForEvent: #anEvent.
 self assert: (action respondsTo: #receiver).
 self assert: ((action receiver == self) not)
```

***.**Mock objects The following test is interesting, as it is programmed by an experienced developer (it uses mock principles [9] to deal with program behavior). Here the methods under test in a cascaded scenario are overwritten so that additional information about the number of calls could be transcribed and tested. We currently subsume this kind of test under *meta tests*.

```
MorphTest≫testIntoWorldCollapseOutOfWorld
 | m1 m2 collapsed |
  "Create the guys"
  m1 := TestInWorldMorph new.
  m2 := TestInWorldMorph new.
  self assert: (m1 intoWorldCount = 0).
  self assert: (m1 outOfWorldCount = 0).
  self assert: (m2 intoWorldCount = 0).
  self assert: (m2 outOfWorldCount = 0).

  "add them to basic morph"
  morph addMorphFront: m1.
  m1 addMorphFront: m2.
  self assert: (m1 intoWorldCount = 0).
  self assert: (m1 outOfWorldCount = 0).
  self assert: (m2 intoWorldCount = 0).
  self assert: (m2 outOfWorldCount = 0).
   (...)
```

*.Naming convention indicates one-method test, but it is not Which is the method under test here, weeks: or days? Days are computed too so it is also an interesting method to test. Our heuristic would detect Duration≫weeks as the method under test. We would manually categorize this one as a *constraint test.*

```
DurationTest≫testWeeks
 self assert: (Duration weeks: 1) days= 7.
```

*.Developers do not agree on method under test Consider the two following tests written by two different developers: They both check if two different kinds of instantiations yield the same result. The name of the first indicates that it is testing =, the name of the second indicates that it tests the creation of instances. Both tests have at least two candidate methods, namely the instance creation methods and the = method.

```
IntervalTest≫testEquals4
  self assert: (3 to: 5 by: 2) = #(3 5).
  self deny: (3 to: 5 by: 2) = #(3 4 5).
  self deny: (3 to: 5 by: 2) = #().
  self assert: #(3 5) = (3 to: 5 by: 2).
  self deny: #(3 4 5) = (3 to: 5 by: 2).
  self deny: #() = (3 to: 5 by: 2).
```

```
MonthTest≫testInstanceCreation
  | m1 m2 |
  m1 := Month fromDate: '4 July 1998' asDate.
  m2 := Month month: #July year: 1998.
  self assert: month = m1.
  self assert: month = m2.
```

Any meaningful definition of *focuses on one method*, where at least two different candidate methods are involved, is likely to be dismissed by at least one of those developers. As a compromise they could categorize both of them as *constraint tests.*

6 Related Work

Binder [10] discriminates between methods under test (*MUT*) and classes under test (*CUT*) but he does not discriminate between unit tests which focus on one or on several *MUTS*.

Beck [5] argues that isolated tests would lead to easier debugging and to systems with high cohesion and loose coupling. *One-method commands* are isolated tests, whereas *multiple method-commands* execute several tests and in the case of *cascaded method test suites* or *multi-facet test suites* depend on each other or on a common scenario.

Eclipse [11] provides a Search≫Referring Tests menu item which allows one to navigate from a method to a JUnit Test that executes this method. However no distinction is made between methods used for setting up the test scenario and those actually under test.

Jézéquel [12] discusses how testing can rely on the *Design by Contract principle* [13] and classes are seen as self-testable entities as much as possible by embedding unit test cases with the class. We found that developers write many tests we could categorize as *one-method commands.* The concept of *one-method commands* even makes methods self-testable. Squeak version 3.7 had almost 900 unit tests but only 24 assertions in the non test code. Associating *one-method examples* with assertion containing methods yields highly abstract *and* executable tests.

Van Deursen et al.[14] talk explicitly about unit tests that focus on one method and start to categorize them using bad smells like *indirect testing*, which describe tests that we would categorize as independent tests. In another paper [15] Van Deursen and Moonen explore the relationships between testing and refactoring, they suggest that refactoring of the code should be followed by refactoring of the tests. Many of these dependent test refactorings could be automated or at least made easier, if the exact relationships between the unit tests and their methods under test would be known.

Bruntink et al.[16] show that classes which depend on other classes require more test code and thus are more difficult to test than classes which are independent. Using *cascaded test suites*, where a test of a complex class can use the tests of its required classes to set up the complex test scenario, should improve the testability of complex classes.

Thomas [17] argues that the message-centric view deserves more attention. *One-method tests*, *optimistic and pessimistic method examples* are all reifications of messages and are the atoms of all *one-method commands* and *multiple-method test suites.*

Edwards [18] is making a claim for *example centric programming*:

> In general, examples are standalone snippets of code that call the code under observation. Unit tests (...) are a good source of examples, and should be automatically recognized as such.

Our taxonomy should help us to link the different kinds of unit tests to the code they are exemplifying.

Test cases are implemented in XUnit using the "pluggable selector" pattern, which avoids the need to create a new class for each new test case at the cost of using the reflection capabilities of the system, thus making the *"code hard to analyze statically"* [5].

7 Conclusions and Future Work

We have developed a taxonomy which categorizes the relations

- between unit tests and methods under test and
- between unit tests and other unit tests.

Knowing these relations can help the developer to refactor, compose and run the program together with the tests, and thus to speed up their co-evolution. It can also help the reengineer to assess if a given method is adequately tested.

We have given initial evidence that the "unit" under test in object-oriented programs is most often a method and that most other kinds of unit tests can be decomposed into *one-method tests*.

We have started to develop some lightweight heuristics to automate this categorization. Our simple heuristics can identify a relevant portion of categories with a high precision rate. We have given evidence why complete automatic classification of unit tests using our taxonomy is impossible for all our suggested algorithms.

We have also discovered that developers write tests which do not have any assertion at all, but only establish whether a given method should or should not throw an exception: 5% of the tests in our manual case study and 2% in the automatic one fell into this category.

In the future we want to explore the following axes of research:

- We want to make the relationships between unit tests and methods under test explicit: First experiments show that if *one-method tests* also delivered the result of their *focused method* as a return value, one could parse the *one-method test* and clearly identify the *focused method*. This link also allowed the composition of tests, and would be stable to refactorings like renaming. Methods in statically typed languages can be void, thus we want to return a complex result object consisting of the receiver, parameters and possibly the return value of the *focused method*. We want to research the pros and cons of alternative denotations of the *focused method* using method comments, specific method sends or in case of Smalltalk bracketing blocks as markers.
- We want to evaluate if an optional 5-pane Smalltalk browser for navigating between tests and methods will be accepted by the Squeak community [19].
- We want to come up with heuristics to automatically categorize multiple method commands.
- We have previously proposed a partial order of unit tests by means of *coverage sets* — a unit test A *covers* a unit test B, if the set of method signatures invoked by A is a superset of the set of method signatures invoked by B [20].

In the four case studies we conducted, 75% of the unit tests were comparable to at least one other unit test in terms of that partial order. These results indicate that unit tests could be refactored into composed one-method tests leading to lower testing time and easier scenario building. We plan to enhance the IDEs of Squeak and Eclipse, so that developers can compose new tests from existing tests.

- We also plan to exploit this overlapping of many tests to identify focused methods under tests: If two tests TestA≫testOne and TestA≫testTwo directly call a method Foo≫foo but TestA≫testOne in addition calls only a method Bar≫bar, chances should be high, that TestA≫testOne is focusing on Bar≫bar.

We see this work as the beginning of the work on classifying unit tests and hope to spawn a discussion about this subject. For this reason we decided to put our taxonomy together with a nomenclature on our web site[7], so that we can easily integrate new kinds of unit tests we find or you report to us.

References

[1] K. Beck, E. Gamma, Test infected: Programmers love writing tests, Java Report 3 (7) (1998) 51–56.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999.

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Reading, Mass., 1995.

[4] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future: The story of Squeak, A practical Smalltalk written in itself, in: Proceedings OOPSLA '97, ACM Press, 1997, pp. 318–326.

[5] K. Beck, Test Driven Development: By Example, Addison-Wesley, 2003.

[6] Maven, http://maven.apache.org.

[7] J. Brant, B. Foote, R. Johnson, D. Roberts, Wrappers to the Rescue, in: Proceedings ECOOP '98, Vol. 1445 of LNCS, Springer-Verlag, 1998, pp. 396–417.

[8] L. Renggli, Smallwiki: Collaborative content management, Informatikprojekt, University of Bern (2003).

[9] T. Mackinnon, S. Freeman, P. Craig, Endotesting: Unit testing with mock objects (2000).

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

[7] http://kilana.unibe.ch/nomenclatureofunittests/

[11] Eclipse Platform: Technical Overview, http://www.eclipse.org/whitepapers/eclipse-overview.pdf (2003).

[12] J.-M. Jézéquel, Object-Oriented Software Engineering with Eiffel, Addison Wesley, 1996.

[13] B. Meyer, Object-Oriented Software Construction, 2nd Edition, Prentice-Hall, 1997.

[14] A. Deursen, L. Moonen, A. Bergh, G. Kok, Refactoring test code, in: M. Marchesi (Ed.), Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001), University of Cagliari, 2001, pp. 92–95.

[15] A. Deursen, L. Moonen, The video store revisited - thoughts on refactoring and testing, in: M. Marchesi, G. Succi (Eds.), Proceedings of the 3nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002), 2002.

[16] M. Bruntink, A. van Deursen, Predicting class testability using object-oriented metrics, in: Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), IEEE Computer Society Press, 2004.

[17] D. Thomas, Message oriented programming, Journal of Object Technology 3 (5) (2004) 7–12.

[18] J. Edwards, Example centric programming, in: OOPSLA 04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, ACM Press, 2004, pp. 124–124.

[19] M. Gälli, O. Nierstrasz, S. Ducasse, One-method commands: Linking methods and their tests, oOPSLA Workshop on Revival of Dynamic Languages (Oct. 2004).

[20] M. Gälli, M. Lanza, O. Nierstrasz, R. Wuyts, Ordering broken unit tests for focused debugging, in: 20th International Conference on Software Maintenance (ICSM 2004), 2004, pp. 114–123.