Towards a Taxonomy of Unit Tests

Markus Gälli, Michele Lanza, Oscar Nierstrasz
Software Composition Group, University of Bern
www.iam.unibe.ch/∼scg

Abstract

Not all unit tests are alike. Some tests are simple one-liners, while others contain a battery of assertions. Certain tests focus on a single method, while others test interactions between methods. There are even tests that do not contain assertions at all. This can make it difficult for a developer to understand which methods are tested by which tests, to what degree they are tested, and what to take into account while refactoring. We have manually analyzed the test base of a large 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 it to two case studies. Beside explaining our taxonomy, we report on our initial results using it, namely that a majority of unit tests focus on single methods and that our lightweight automatic categorization could already classify more than 50% of these single method commands.

Keywords: [unit testing, taxonomy, classification, reverse engineering, dynamic program slicing, unit test composition, traceability]

1 Introduction

Although modern object-oriented programming environments help developers to navigate between related methods in a complex software system, they offer no means to relate methods and the unit tests that test them. As a consequence, developers lack support for:

- navigating between methods and their tests,
- knowing which methods are tested by a given unit test,
- refactoring tests together with the program,
- and composing tests from simpler tests.

In particular, a developer has no way, short of reading individual unit tests, of knowing what the relationship is between a set of unit tests and the methods being tested. Furthermore, these relationships can assume various forms, depending on how complex the individual test is. Test methods may invoke one or several methods under test, and may perform any number of assertions at various points during the test.

In order to better support developers in keeping track of the relationships between units tests and methods under test, we have:

- Developed an initial taxonomy of unit tests by carrying out an empirical study of a substantial collection of tests produced by different developers.
- Implemented some lightweight tools to automatically classify certain tests into categories offered by the taxonomy.
- Applied these tools to 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 1:1 relationship to a method under test, but suggests that a general algorithm to distinguish this kind of tests from the others is difficult to find if not impossible. Yet our initial heuristics to automate this endeavor could identify 50% of these 1:1 tests without selecting any false positives.

Structure of the paper. In Section 2 we present the taxonomy derived from our empirical study. In Section 3 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 4 we discuss some of the problems and difficulties encountered. Section 5 briefly outlines some related work. In Section 6 we conclude with some remarks about future work.

2 A Taxonomy of Unit Tests

Initial case study. We derived the taxonomy by manually categorizing 671 unit tests of of the Squeak [10] base

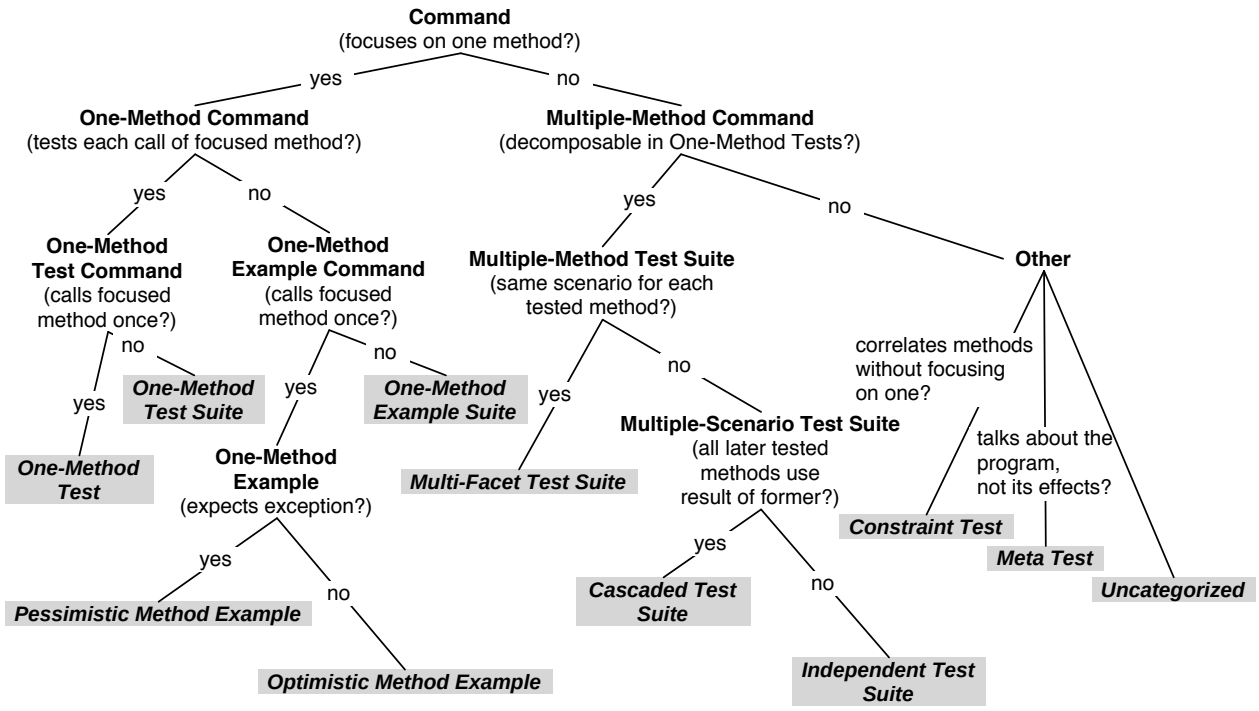

Figure 1. Taxonomy of unit tests.

system[1]. In order to reduce the effort of our initial investigation, we did not categorize roughly 300 tests which did not clearly identify the package under test. 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. The remaining developers each produced less than 3% of the tests.

We defined the taxonomy depicted in Figure 1 by iteratively grouping tests into categories and refining the classification criteria. Our manual categorization yielded a distribution of the categories shown in Figure 2.

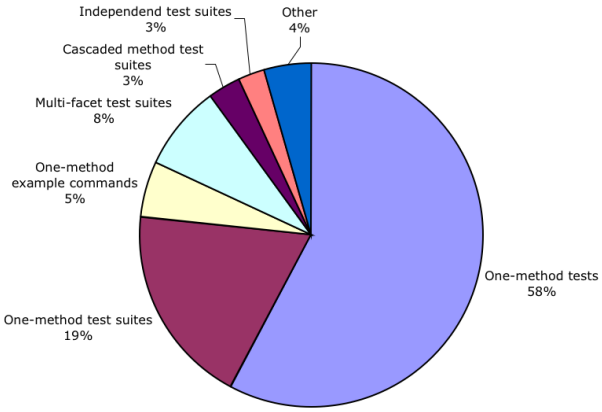

Figure 2. Manual classification of unit tests for the base squeak system

2.1 Basic Definitions

First we introduce some basic terminology, and then we describe the categories of unit tests appearing in the taxonomy.

- *Assertion:* An *assertion* is a method that evaluates a (side-effect free) boolean expression, and throws an exception if the assertion fails.
- *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 case of Smalltalk we use class-categories as the smallest common denominator of several Smalltalk dialects.
- *Command:* A *command* is a parameter-free method whose receiver can be automatically created. The method can thus be automatically executed.

[1] Version 3.7 beta update 5878, also available at www.squeak.org

- *Test package:* A package which includes a set of commands.
- *Package under Test:* If a test package is tested by 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 method of the package under test.
- *Dynamic program slice:* The *dynamic program slice* presented by Korel *et al.* [**?**] is a part of a program that affects the computation of a variable of interest located in a statement during program execution on a specific program input.
- *Assertion slice:* An *assertion slice* of an assertion in a command is an intersection of all candidate methods called directly by the command and the dynamic program slice of the command up to the evaluation of the assertion and with respect to all variables used in the assertion. In the following example the program slice up to the first assertion with respect to variable c would not include the fourth statement and thus no call of the subtraction operator.

```
testSumAndSubstract
  a=1;
  b=1;
  c=a+b;
  d=c-1;
  assert c==2;
  assert d==a;
```

- *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.

2.2 Categories of Unit Tests

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

- *Command*: Unit tests are commands: The command receiver in the case of a JUnit test case can be constructed automatically, *e.g.*, new MyTestCase(myTestSelector). The whole command then looks like this:
  (new MyTestCase(myTestSelector)).run()
- *One-method command*: A *one-method command* is a command that focuses on a single method.

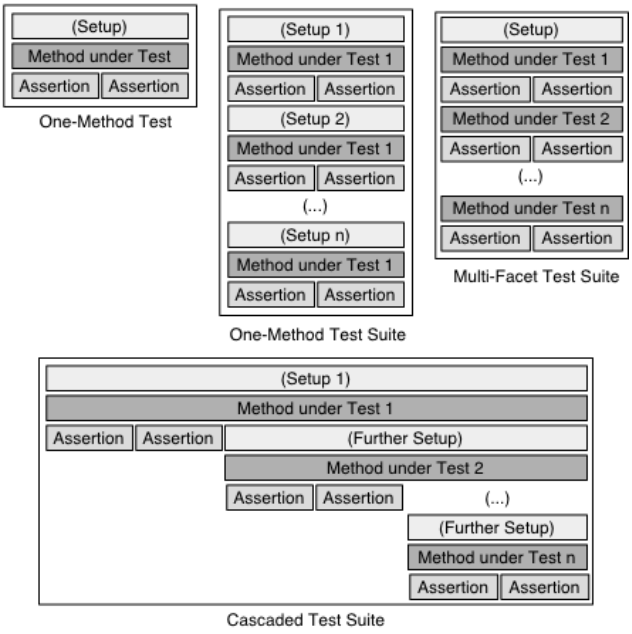

Figure 3. One-method test suites, multi-facet test suites and cascaded test-suites are decomposable into one-method tests.

- *One-method test command:* A *one-method test command* is a one-method command which has assertions testing the outcome of *each occurrence* of the method under test.
- *One-method test:* A *one-method test* is a one-method test command which tests the outcome of one occurrence of a method under test. Example:

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

- *One-method test suite:* A *one-method test suite* is a one-method test command which tests the outcome of the method under test in several situations. Example:

```
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.
```

- *One-method example command:* A *one-method example command* is a one-method command which does not have assertions for all calls of the method under test.

- *One-method example:* A *one-method example* is a one-method example command which calls the method under test exactly once.

- *Pessimistic method example:* A *pessimistic method example* checks that an exception is thrown if a method is called in a way which violates a precondition. Example: The unit test below ensures that there is only one instance of the class True, *i.e.,* new may not be invoked. Beck [**?**] calls them *"exception tests"*.

```
TrueTest>>testNew
  self should: [True new] raise: Error.
```

  A precondition in an implementation could look like

```
Object class>>new
  self deny:
    (self includesBehavior: Boolean).
  (...)
```

  Note that we consider neither shouldnt: raise: nor should: raise: as assertions, because they do not construct states where something is true or false, but merely check whether an exception is thrown.

- *Optimistic method example:* An *optimistic method example* is an one-method example which expects that no exception is thrown if the method under test is called without violating some preconditions. The unit test below tests that the invocation of initialize does not throw an exception:

```
TextMorphTest>>testInitialize
  "For now, just make sure initialization
  doesn't throw exception"
  self
    shouldnt: [TextMorph initialize]
    raise: Error.
```

- *One-method example suite:* 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 commands which call the same focused method once.

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

- *Multiple-method command:* A *multiple-method suite* is a command which does not only focus on a single method.

- *Multiple-method test suite:* A *multiple-method test suite* is a multiple-method command which is decomposable into one-method tests. (See Figure 3).

- *Multi-facet test suite:* A *multi-facet test suite* is a multiple-method test suite which reuses a scenario to test several candidate methods. Example:

```
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'.
```

- *Multiple-scenario test suite:* A *multiple-scenario test suite* is a multiple-method test suite which does not use the same scenario for testing each method.

- *Cascaded test suite:* A *cascaded test suite* is a multiple-scenario test suite, in which the results of one test are used to perform the next test. Example: Below we see a cascaded test suite that builds a scenario, triggers a method, tests its outcome, and then recursively uses the result to trigger another method and test its outcome.

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

- *Independent test suite:* An *independent test suite* is a multiple-scenario test suite, which tests different methods on different receivers, but which do not depend on each other.

- *Others:* All test cases which neither focus on one method nor are decomposable into one-method tests.

  – *Constraint tests:* A *constraint test* checks the interplay of several methods without focusing on one of them. In the following example one cannot tell, if the setter or the getter methods are tested. Consequently the developer chose *testExpAccessors* as the command name.

```
SWCacheTests>>testExpAccessors
  exp ttl: 1111.
  self assert: exp ttl = 1111.
  exp check: 2222.
  self assert: exp check = 2222.
```

– *Meta tests:* Tests which are about the application ifself, *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.
```

– *Uncategorized:* All unit tests that could not be assigned to one of the above categories. After the second manual iteration to classify the Squeak unit tests we had eight uncategorized tests, after the last iteration all Squeak unit tests could be put into a real category so we cannot display an example here.

3 Towards an automatic classification of unit tests

After having manually derived the taxonomy, we started to develop some lightweight heuristics to automatically detect the feature properties depicted in Figure 1. 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 (see Figure 4) of the unit tests focuses on single methods.

3.1 Approach: Instrumentation

To detect the feature properties we relied 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 [3], 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 yielded the following advantages and disadvantages:

- It is slower than current VM optimized method wrapper code.
- It is more general than *method wrappers* as also base level classes can be tested.
- 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. 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.

3.2 Lightweight heuristics for automatic classification

In the following we present a list of heuristics used to detect the feature properties displayed in the left subtree of the Figure 1. 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:

  – *Deduction of the focused method from the command name.*

  One lightweight 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 XUnit 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:.

  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 we would look for an event in which a candidate method foo is called. If there are two selectors called, like foo: and foo, the result would be ambiguous and we could not say on which of them our test would focus on.

– *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 XUnit-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 set up. Note that 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.
  * the set of candidate methods called by each *assertion slice* consists of only one method. (See also Section 2)

– *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.

Calls only one candidate method:

We did not yet implement the assertion slicing algorithm and we did not have any historical data. 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.

So we used a combined approach: We exploited the naming convention and checked if only one candidate method was called.

- *Further classification in the left subtree:* To determine if a *one-method command* is a *one-method test command* or a *one-method example command* we currently simply 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 commands* which does not call any kind of should:raise: and shouldnt:raise:.

3.3 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) 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.

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.

3.4 A second case study: SmallWiki

As a next step we automatically categorized the 200 unit tests of *SmallWiki* [14], a collaborative content management tool written in VisualWorks Smalltalk and ported to Squeak. A surprising result here was that more tests could be detected as focusing on one method by using the first part of our hybrid approach, namely considering the calls of only

one candidate method, than by exploiting their naming convention. The categorization using our hybrid approach is shown in Figure 4.

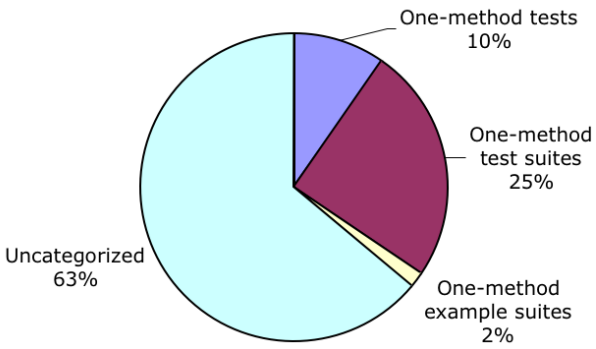

Figure 4. Automatic classification of unit tests for the Smallwiki system

We only detected three kinds of 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. Having achieved a precision of around 90% in the previous case study, but only a recall of 52% suggests that a manual classification or an enhanced automatic one would yield again a majority of *one-method tests*. Figure 4 shows that contrary to the Squeak case study, the developers here wrote more *one-method test suites* than *one-method tests*. This result is backed by our personal experience with the two case studies.

4 Discussion

Although the taxonomy we have derived appears promising, it should be considered only a preliminary result for several reasons:

- Our taxonomy is based on the manual classification of only a single case study.
- We have only carried out a single case study in which we compared our manual and automatic classification.
- We focused on XUnit Tests, as they are described by Beck *et al.* in [1] so we do not know if other frameworks make developers write other kinds of unit tests.
- We have addressed the question if unit tests should be considered whitebox 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 less representative.
- We only worked in Smalltalk, which is dynamically typed. It is conceivable that different styles of unit tests are more common in statically typed languages, or that other categories of tests arise in other languages.

The automatic classification heuristics are similarly preliminary for the following reasons:

- Using the naming convention for automatic detection of the method under test is unreliable as it is ambiguous in the case of XUnit. For example, does the following test focus on Foo≫bar:, on Foo≫bar or even on 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)
```

- Developers have complete freedom to write any kind of unit tests – making automatic classification a difficult business. Here some examples why automatic classification is hard:
  - Tests of the test framework may be incorrectly categorized. This test could be classified as 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
```

  - Scenario with cleanup, method under test happens earlier. 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
```

– Some tests were testing methods which were not the last method of the package called before the assertion occurred. Example: Is the method under test removeActionsWithReceiver: or actionForEvent:? The name of the command indicates the former, but the structure of the test suggests the latter:

```
EventManagerTest>>testRemoveActionsWith\-
Receiver
 | 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)
```

– The following test is interesting, as it is programmed by an experienced developer and as it uses mock principles [12] 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 collapsed |
  "Create the guy"
  m1 := TestInWorldMorph new.
  self assert: (m1 intoWorldCount = 0).
  self assert: (m1 outOfWorldCount = 0).
  (...)
```

– Which is the method under test here, weeks: or days? Days could be computed so it could be an interesting method to test. Our heuristic would detect Duration≫weeks as the method under test.

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

– 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 =.

```
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;
    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.

– A special method test: Not the result of the method is checked but rather whether it throws a specific exception.

```
UndefinedObjectTest>>testHaltIfNil
  self should: [ nil haltIfNil]
  raise: Halt.
```

– A counter example why assertion slicing cannot always work

```
AccountTest>>testSetMaxCredit
  anAccount:= Account new balance: 100.
  anAccount setMaxCredit: -500.
  self assert: (anAccount maxCredit == -500).
  self assert: (anAccount  balance == 100)
```

The last assertion just states that a change of a maximum credit would not change the balance of the account. If the slicing algorithm checks on the base of touched variables and not only on the base of touched objects, the slice of the last assertion would not include Account≫setMaxCredit:, as this method never touched the balance variable in the account, thus Account≫setMaxCredit: would not be the method under test.

5 Related Work

IEEE defines *unit testing* as "*Testing of individual hardware or software units or groups of related units.*" [9] –

our results indicate that object-oriented programmers most often focus on methods as their units under test.

Binder [2] 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 [**?**] 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. Having had a big influence by his eXtreme programming methodology and by the development of the XUnit framework, it is imaginable that developers have widely adopted his views, leading to a high percentage of *one-method commands*, at least when developed within the XUnit-framework.

Van Deursen *et al.* [6] 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 [5] 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.

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

6 Conclusions and Future Work

In this paper we developed a taxonomy of unit tests which assess the relations between unit tests and methods under test and between unit tests and other unit tests. Knowing this relations can help the developer to refactor, compose and run the program with the test and thus to speed up their co-evolution. It can also help the reengineer to assess if a given method is rigorously tested.

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

We started to develop some lightweight heuristics to automate this categorization and proposed the idea of *assertion slicing* as a more accurate technique. Our simple heuristics can identify a relevant portion of categories with a high precision rate.

We gave evidence, why complete automatic classification of unit tests using our taxonomy is impossible for all our suggested algorithms, which is why we call them heuristics.

We also learned that developers write tests which do not check any assertion at all, besides that a method should or should not throw an exception: 5% of the tests in our manual case study and 2% in the automatic one were of that kind.

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

- We want to implement heuristics to detect one-method tests by
  - using *assertion slicing* based on a *program slicing* ([11]).
  - or alternatively exploiting versioning data
- We want to show that decomposing tests into post-conditions and optimistic method examples is always possible and yields flexibility to composing new tests.
- We want to come up with heuristics to automatically categorize multiple method commands.
- 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 then want to enhance the IDEs of Squeak and Eclipse, so that developers can easily navigate between linked tests and methods.
- 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 [8]. In the four case studies we conducted there 75% of the unit tests were comparable to at least one other unit test in terms of that partial order. This 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 out of existing tests.

Acknowledgments

We 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, Oct. 2002 - Sept. 2004).

References

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

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

[3] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. In *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

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

[6] A. v. Deursen, L. Moonen, A. v. d. Bergh, and G. 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.

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

[8] M. Gälli, M. Lanza, O. Nierstrasz, and R. Wuyts. Ordering broken unit tests for focused debugging. In *20th International Conference on Software Maintenance (ICSM 2004)*, 2004. to appear.

[9] A. Geraci, F. Katki, L. McMonegal, B. Meyer, and H. Porteous. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. IEEE, 1991.

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

[11] L. Larsen and M. Harrold. Slicing object-oriented software. In *Proceedings ICSE '96*, pages 495–505. IEEE, 1996.

[12] T. Mackinnon, S. Freeman, and P. Craig. Endotesting: Unit testing with mock objects, 2000.

[13] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

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

[15] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[16] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings ICSM 1999*, pages 179–188, Sept. 1999.

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