One-Method Commands: Linking Methods and Their Tests

Markus Gälli
Oscar Nierstrasz
Stéphane Ducasse
Software Composition Group
University of Bern

E-mail: {gaelli,nierstrasz,ducasse}@iam.unibe.ch

Abstract

Although unit testing is essential for programming, current languages only barely support the developer in composing unit tests into new ones or in navigating between unit tests and their corresponding methods under test.

We have taken several Smalltalk programs and analyzed the relationships between unit tests and methods under test, and the relationships amongst unit tests. First results indicate that most unit tests can be seen or at least decomposed into commands which focus on single methods, and that large portions of unit tests overlap each other. But these relationships between unit tests and methods under test are not reflected in current languages.

We therefore first conceptually extend the meta-model of Smalltalk with one-method commands *so that unit tests become both composable and navigable. Then we introduce a first lightweight implementation of this meta model using method comments to differentiate between the several test phases of existing XUnit test case methods.*

1. Introduction

In the popular unit test framework XUnit [4] the link between the method under test and its unit tests is only established by means of a naming convention. As a consequence a programmer looking at a method cannot easily tell if it has any dedicated unit tests, and can thus neither be sure of its quality nor navigate directly to a dedicated testcase of this method.

We first motivate our position that typical XUnit tests indeed focus on a single method or can be decomposed into ones which do so.

We then propose to conceptually enhance the meta model of object-oriented languages by linking methods under test explicitly to their corresponding tests. We also want to reflect this change in the Class Browsers as seen in Figure 4.

We describe a lightweight implementation of this meta model in which the different test phases like setup, execution and assertion phase are annotated with method comments.

Another problem of XUnit is that test scenarios cannot always be shared between different test cases: Although complex objects typically are built out of simple ones, the XUnit tests of more complex objects cannot naturally be built from test scenarios of the simpler ones.

We have evidence that a relevant portion of executed methods of different XUnit tests overlap [8] and thus many unit tests could be simplified, be run in shorter time, and be better focused on possible errors. We therefore suggest to use *one-method commands*, which in contrast to current unit tests, return a modified test object to build more complex test scenarios.

2. Differences to existing approaches

Romain Robbes, while working on integrating SUnit into the Smalltalk Browser [13], faced the problem of relating methods under test to their test methods. In the first version of his tool, he exploited the naming convention of SUnit to establish that relation. Due to the fact that it is difficult to keep the naming convention in sync or to establish it all in a consistent manner, in the second version he just browses to all test cases, which directly send the method. Eclipse [7] takes a similar approach using a "Search≫Referring Tests " menu entry, which is not well integrated in the user interface of Eclipse.

Every test can be seen as a *command* [9], as is also stated in the comment of TestCase class of XUnit. [5]

Closest to our approach of *one-method commands* are instance specific methods, which are methods that can be bound to specific instances as their receivers. Matsumoto implemented Singleton methods in Ruby [12], and Beck describes [2, 3] how and why instance-specific methods can be added to Smalltalk. Our one-method commands differ substantially from these instance specific methods as one-method commands also provide explicit construc-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA 2004 Vancouver, Canada
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

tors for creating the receivers and parameters of the method, and thus are composable. (See Figure 3).

3. Developers write unit tests which overlap

> If the quality assurance of factories worked like XUnit Tests, every subcomponent would be thrown away after having been tested, making it impossible to reuse the tested subcomponent in another component.

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, 75% of the unit tests were comparable to at least one other unit test in terms of that partial order.

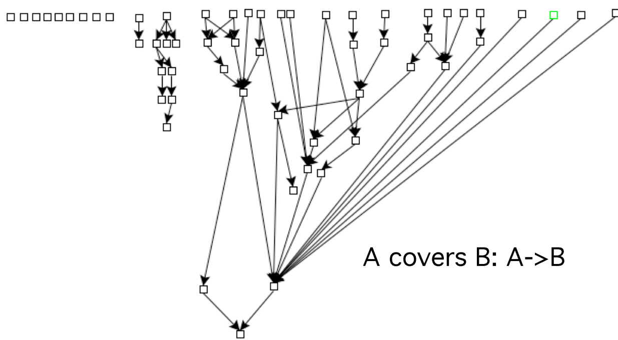

Figure 1. The coverage hierarchy of the Code Crawler tests visualized with Code Crawler.

In Figure 1 an arrow from one test node to another indicates that the first *test node covers* the second. We see a typical coverage hierarchy obtained in the first part of our experiment: Most of the unit tests either covered or were covered by some other unit test and only 5% to 16% of them were standalone nodes.

These results suggest that unit tests could be refactored into composed one-method tests leading to lower testing time and more compositional scenario construction. See also the composed bank tests in the appendix as an example.

4. Using one-method commands

Some definitions:

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

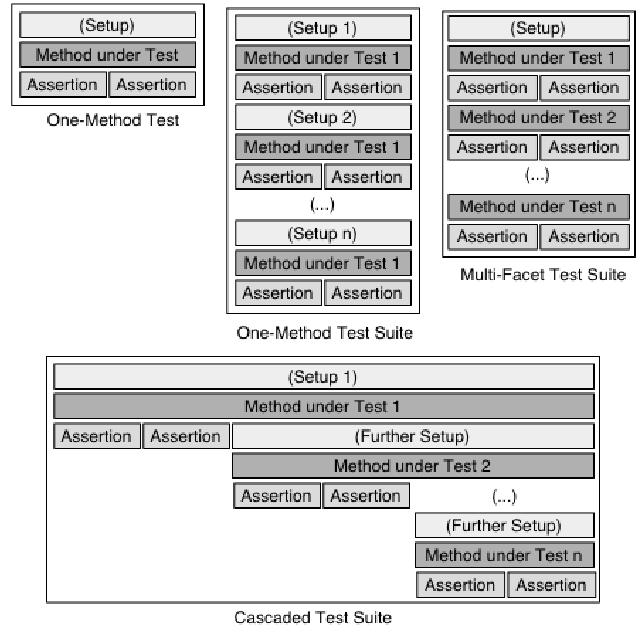

Figure 2. One-method test suites, multi-facet test suites and cascaded test-suites are decomposable into one-method tests.

63% of 650 SUnit tests of the Squeak base system in version 3.7 [10] could be (manually) categorized as *one-method commands*.

Another 35% of the Squeak unit tests were of one of the kinds depicted in Figure 2 and could be decomposed into *one-method commands*. Only 2% were not decomposable into one-method tests as they tested existing properties of the system like subclass relationships or talked about the program structure itself.

Benefits of being able to navigate between tests and methods under test:

- There is no browser dichotomy, only one tool is needed to program *and* test.
- Methods can be immediately seen in action.
- Tests serve as documentation for the methods.
- Better support for test-first programming.
- In principle it is then possible to test private methods as the tests belong to the methods.

5. Enhancing the meta-model for unit tests

As a prerequisite for our meta model we make the following observation:

The receiver or any parameter of a method under test is either a value object or the result or side effect of a combination of some other commands. These other commands always have a last method called, so they could always be treated as a one-method command of this last method called.

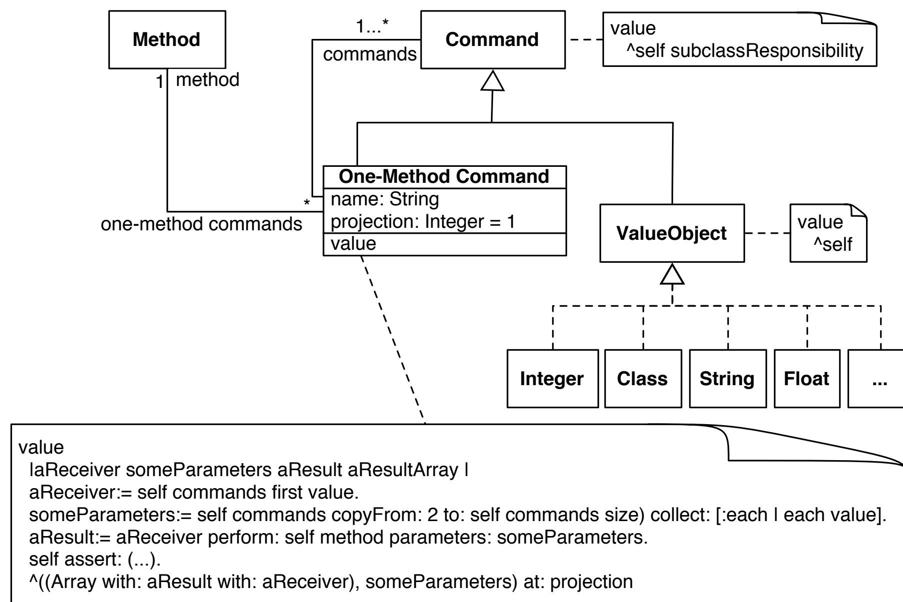

Figure 3. A meta model extension for unit tests making them navigable and composable: Each method knows its one-method commands. The receiver and parameters for new one-method commands are created either with value objects (like integers, strings) or the result of existing one-method commands.

We reflect this observation by the meta model depicted in Figure 3. We plan to implement this meta model in Squeak with the following constraints in mind:

- It should be easy to convert existing XUnit tests to the new model.
- The test cases should be readable and writable as whole methods as they are now.
- The user interface (class browser) should reflect the common one-to-one relationship between methods under test and test methods. (See Figure 4)
- There should not be any parallel hierarchy of test case classes.

As a first lightweight approach we establish this link by separating the different test phases, namely the setup, the execution of *one* method under test, the assertion and the cleaning up phase (see Figure 2), by using method comments. (Figure 4). We can then detect the methods under test by simply parsing the test case and extracting the method called after the comment for test execution.

6. Discussion

One-method commands suggest the following lines of investigation:

- Can a true test-first developer drag and drop the scenarios for new methods out of existing *one-method commands* or value objects? Is it possible to offer the test-driven developer an interface like EToys [1], where only the name of a new method has to be typed and the rest can be dragged and dropped out of existing building blocks?
- Can such a test-driven development make static typing superfluous or at least less important, as any concrete type of both the receivers, parameters and result of method, which has a dedicated *one-method command* could be derived easily out of the command? It is no wonder that XUnit originates from the dynamically typed Smalltalk, since having tests makes static typing less and less important. Testing brings former apologists of statically typed languages also to this conclusion. [11, 6]
- Can instance-specific methods be used for pre-computing and caching the results of some common entries, thus speeding up the process?
- Though one can easily find out how these tests are composed, it is not so easy to automatically detect which method they focus on. We will first investigate some heuristics how this can be automatically detected, and then consider how the connection could be made explicit.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects “Tools and Techniques for Decomposing and Composing Software” (SNF Project No. 2000-067855.02, Oct. 2002 - Sept. 2004) and “RECAST: Evolution of Object-Oriented Applications” (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006).

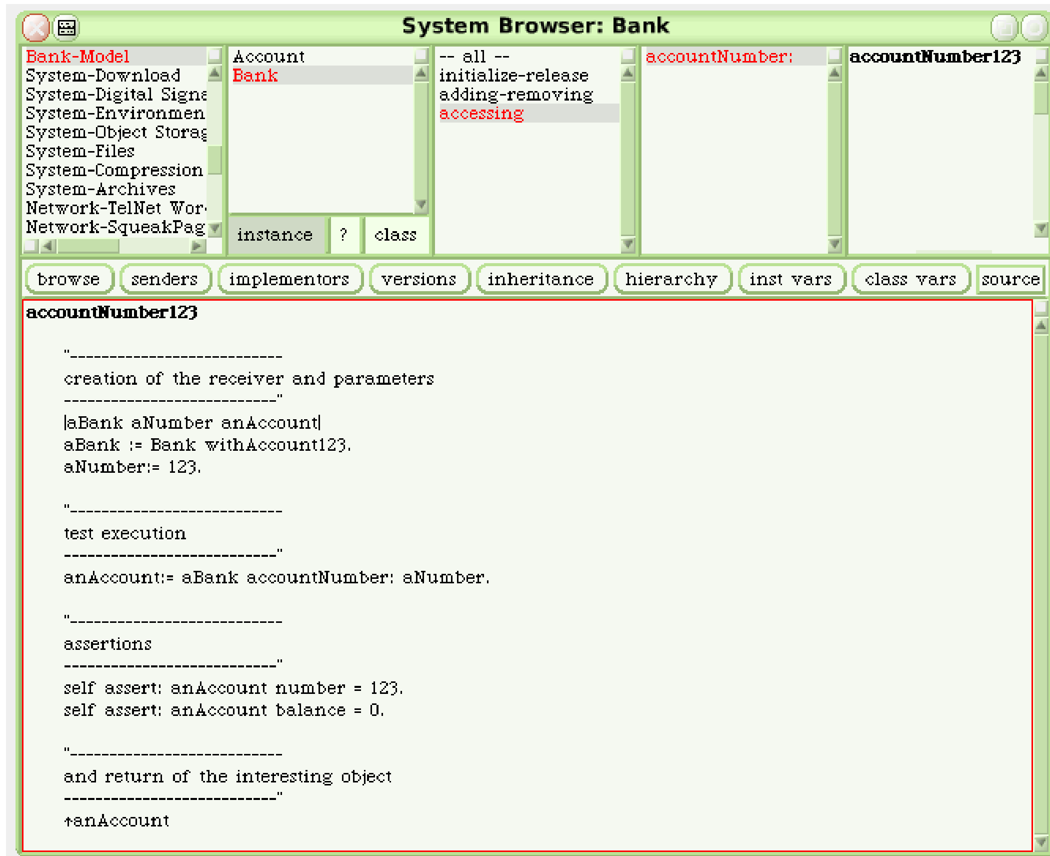

Figure 4. A Squeak class browser reflecting the new meta model using a fifth pane for the according test of a method.

7. REFERENCES

[1] B.J. Allen-Conn and Kimberly Rose. *Powerful Ideas in the Classroom*. Viewpoints Research Institute, Inc., 2003.

[2] Kent Beck. Instance specific behavior: Digitalk implementation and the deep meaning of it all. *Smalltalk Report*, 2(7), May 1993.

[3] Kent Beck. Instance specific behavior: How and Why. *Smalltalk Report*, 2(7), May 1993.

[4] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.

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

[6] Bruce Eckel. Strong Typing vs. Strong Testing. http://www.mindview.net/WebLog/log-0025.

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

[8] Markus Gälli, 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.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[10] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97*, pages 318–326. ACM Press, November 1997.

[11] Robert C. Martin. Are Dynamic Languages Going to Replace Static Languages? http://www.artima.com/weblogs/viewpost.jsp?thread=4639.

[12] Yukihiro Matsumoto. *The Ruby Programming Language*. Addison Wesley Professional, 2002. To appear.

[13] Romain Robbes. Browse Unit: Integrating SUnit into the Smalltalk Browser. http://minnow.cc.gatech.edu/squeak/3113.

APPENDIX

We present some test cases of the canonical bank example which have been refactored to introduce parsable testing comments and to let them return the "interesting" resulting object. Note that we store the test cases on the class side of the resulting object, as the tests also serve as factories.

A smart testing tool would only execute

- Bank class≫withdrawOkFrom123
- Bank class≫withdrawTooMuchFrom123 and
- Bank class≫deleteAccount123

as these test cases include all the others.

```
Bank class >> withAccount123
  "creation of the receiver and parameters"
  |aBank anAccountNumber|
  aBank := Bank new.
  anAccountNumber := 123.

  "test execution"
  aBank createAccount: anAccountNumber.

  "assertions"
  self assert: aBank accounts notEmpty.

  "and return of the interesting object"
  ^aBank

Account class >> accountNumber123
  "creation of the receiver and parameters"
  |aBank anAccount|
  aBank := Bank withAccount123.

  "test execution"
  anAccount:= aBank accountNumber: 123.

  "assertions"
  self assert: anAccount number = 123.
  self assert: anAccount balance = 0.

  "and return of the interesting object"
  ^anAccount

Account class >> deposit100On123
  "creation of the receiver and parameters"
  |anAccount|
  anAccount:= Account accountNumber123.

  "test execution"
  anAccount deposit: 100.

  "assertions"
  self assert: anAccount balance = 100.

  "and return of the interesting object"
  ^anAccount

Account class >> withdrawOkFrom123
   "creation of the receiver and parameters"
  |anAccount |
  anAccount:= Account deposit100On123.

  "test execution"
  anAccount withdraw: 60.

  "assertions"
  self assert: anAccount balance = 40.

  "and return of the interesting object"
  ^anAccount

Account class >> withdrawTooMuchFrom123
  "creation of the receiver and parameters"
  |anAccount |
  anAccount:= Account deposit100On123.

  "test execution"
  self shouldRaiseException: [anAccount withdraw: 160].

  "assertions"
  self assert: anAccount balance = 100.

  "and return of the interesting object"
   ^anAccount


Bank class >> deleteAccount123
  "creation of the receiver and parameters"
  |aBank anAccount |
 aBank:= Bank withAccount123.
 self assert: (aBank accountNumber: 123) notNil.

 "test execution"
 aBank deleteAccount: (aBank accountNumber: 123).

 "assertions"
 self assert: (aBank accountNumber: 123) isNil.

 "and return of the interesting object"
 ^aBank
```