Modeling Examples to Test and Understand Software

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Markus Gälli

von Tübach (SG)

Leiter der Arbeit:
Prof. Dr. Stéphane Ducasse
Prof. Dr. Oscar Nierstrasz
Institut für Informatik und angewandte Mathematik

Modeling Examples to Test and Understand Software

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Markus Gälli

von Tübach (SG)

Leiter der Arbeit:
Prof. Dr. Stéphane Ducasse
Prof. Dr. Oscar Nierstrasz
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 13.11.2006

Der Dekan:
Prof. P. Messerli

Abstract

One of the oldest techniques to explain abstract concepts is to provide concrete examples. By explaining an abstract concept with a concrete example people make sure that the concept is understood and remembered.

Examples in software can be used both to test the software and to illustrate its functionality. Object-oriented programs are built around the concepts of classes, methods and variables, where methods are the atoms of the functionality. But the meta-models of object-oriented languages do not allow developers to associate runnable and composable examples with these concepts and particularly not with methods.

Unit tests on the other hand, assure the quality of the units under test and document them. Not being integrated into the language, unit tests are not linked explicitly to their units under test which makes it unnecessarily difficult to use them for documenting, typing and debugging software. In addition they are not composable making it hard to develop higher level test scenarios in parallel with higher level objects.

In this thesis we analyze unit tests to learn about implicit dependencies among tests and from tests to the methods under test. We develop a technique to partially order unit tests in terms of their covered methods, which reveals possible redundancies due to the lack of composability. We show how partial orders can be used to debug and to comprehend software. We then develop a taxonomy based on several case studies revealing that a high fraction of unit tests already implicitly focuses on single methods. We show that the rest of the tests can be decomposed into commands focusing on single methods.

We build a meta-model based on our findings of analyzing test interdependencies which establishes how tests can be explicitly linked to their method under test and how they can be composed to form higher-level test scenarios. We explain how the problems of missing links between tests and units under test are solved using our meta-model. Furthermore, we implemented the meta-model and a first user interface on top of it to give first evidence of how our model supports the developer.

Acknowledgments

First and foremost, I would like to thank my advisors Oscar Nierstrasz and Stéphane Ducasse. They gave me the opportunity to work in an inspiring environment based on trust. They provided me lots of good examples of how to collaborate in a scientific environment – how to do research, how to work on papers and how to give presentations and lectures. Oscar and Stéph, it was a great pleasure to work with you during all these years.

Many thanks also for giving me some time to play around with Etoys and letting me present two papers in Berkeley, which were not directly related to my thesis. I was glad to be able to go there.

I would also like to thank the other member of my Ph.D. committee, Robert Hirschfeld. Thanks for inviting me to Potsdam. You gave me very detailed and good critique of my thesis! All: Keep on Squeaking!

Very special thanks go to my parents, you helped me a lot during the hard times and always trusted in me. Thanks for your love and for invoking the playful child in me.

I would also like to thank my friends for all the adventures and the good times we have experienced together and for the support and advice you have given me. Best regards especially go to the most venerable members of the "Zieschtigsclub", to the "french connection" and to all I ever had a cool beer and good talk with – was it in the "Musigbistrot" or in the "Chine".

Many thanks also to all the colleagues who have helped me and collaborated with me during my work, especially to Doru and Orla, who always have an open ear and a helpful advice. Thanks for reviewing my thesis and for being an integral part of the social center at the Software Composition Group.

Many thanks go to Marcus for being part of our common Squeak adventures.

Next I would like to thank all the former and current members of the Software Compo-

sition Group. It was a pleasure to work (and party) with you : Gabriela Arevalo, Alexandre Bergel, Frank Buchli, Philipp Bunge, Juan Carlos Cruz, Marcus Denker, Adrian Kuhn, Michele Lanza, Adrian Lienhard, Philippe Marschall, Michael Meyer, Laura Ponisio, Stephan Reichhart, Lukas Renggli, Matthias Rieger, David Röthlisberger, Nathanael Schärli, Andreas Schlapbach, Therese Schmid, Florian Thalmann, Sander Tichelaar, Rafael Wampfler, and Roel Wuyts.

I would also like to thank all the people that supported and hosted me during my travels and stays abroad and send my best to all the great people I have met and talked to at conferences, meetings, workshops, online and elsewhere.

Denise, thanks for your great support during the difficult months of the thesis. I am happy we found each other – you are key.

Markus Gälli

November, 13th 2006

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Main Entry: **[1]ex·am·ple**
Pronunciation: `ig-'zam-p&l`
Function: *noun*
Etymology: Middle English, from Anglo-French *essample*, *example*, from Latin *exemplum*, from *eximere* to take out, from *ex-* + *emere* to take -- more at REDEEM
1 : one that serves as a pattern to be imitated or not to be imitated <a good *example*> <a bad *example*>
2 : a punishment inflicted on someone as a warning to others; *also* **:** an individual so punished
3 : one (as an item or incident) that is representative of all of a group or type
4 : a parallel or closely similar case especially when serving as a precedent or model
5 : an instance (as a problem to be solved) serving to illustrate a rule or precept or to act as an exercise in the application of a rule
synonym see INSTANCE, MODEL
- for example `/f&r-ig-'zam-p&l, frig-/` **:** as an example <there are many sources of air pollution; exhaust fumes, *for example*>

Figure 1.1: "Example" defined in the Myriam Webster Online Dictionary. Note how dictionaries use examples to explain words in their context: "There are many sources of air pollution, exhaust fumes, for example." The concept of examples is currently not used in object-oriented programming languages which makes it hard to test and document software.

People learn new concepts by example. They *e.g.,* learn new words by looking them up in dictionaries and understand them in an exemplified context. Dictionaries like the Oxford Dictionary of Current English [Soanes, 2001] or the Webster Online Dictionary Figure 1.1 (p.1) all present *exemplified* usages of words for different kinds of contexts where they are used. Cognitive psychologist Jerome Bruner *et al.* state that all learning means to compare examples and counter examples in order to derive an intentional description of the subject at hand [Bruner *et al.*, 1956] while Einstein even believed, that "Example isn't another way to teach. It is the only way to teach.".

But the concept of examples is not established within current object-oriented programming languages.

1.1 The Problem

Context Software developers have to illustrate their abstract programs to two audiences: The computer, which is supposed to run and quality-assure their programs on its own by using tests, and developers, who are supposed to understand the program while building it and at a later time while extending it. In traditional software development the process of writing documentation is different from the process of writing tests. This dichotomy is reflected *or caused* by existing concepts of object-oriented programming languages. There are concepts which *either* test *or* document programs but there are no concepts which test *and* document software at the same time (Figure 1.2 (p.3)). All programming languages used in industry today provide means to document code whereas none provides means to automatically test the programs out of the box by integrating automated testing into the development cycle. On the other hand developers prefer to write code over documentation, which explains the rise of unit testing frameworks being an integral part of current agile software methodologies like eXtreme programming.

Agile methodologies aim to reduce the number of superflous artifacts used in traditional software development and rely more on executable and well factored code and tests than on written documentation. Agile developers aim at self documenting code, as the code is written with meaningful class, variable and method names, functionality is not duplicated, and as much functionality as possible is tested. As a consequence written documentation rarely exists in these projects and tests become the most useful resource for understanding the software. (See Figure 1.3 (p.4))

Unit test frameworks are an ad-hoc addition to current programming environments and thus not well integrated into them: Though they help developers to write repeatable automated tests, they leave them alone both with exploiting these tests as means of

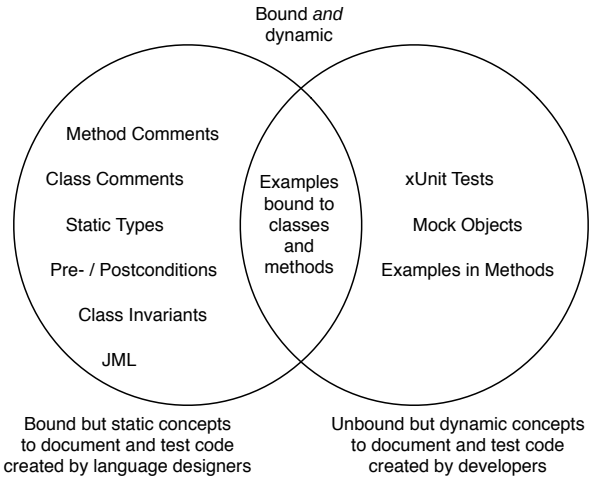

Figure 1.2: Both language designers and developers came up with a list of independent techniques and concepts to ease the process of translating requirements into tests, of documenting and of testing code.

documentation and with composing scenarios for higher level unit tests - which is the most time consuming part of writing tests as mentioned by Zeller [Zeller, 2005].

The main reason for unit tests not being integrated into current programming languages and environments is that the relationship of unit tests to other code artifacts – including other unit tests – is left open by the unit test framework designers.

No distinction of possible units under tests are made. As a consequence developers write all kinds of unrelated unit tests: Some of those tests focus on single methods and execute them only in one scenario, some of them test several methods in different scenarios, and none of them can be reused by yielding their scenarios to create higher level scenarios. Agile communities have renamed “unit tests” into “developer tests” and “acceptance tests” into “customer tests”. This reveals on the one hand the similarity of those two kinds of tests, but on the other hand focuses more on who is writing the tests and even less on what the tests are testing.

Not having an explicit link between a unit test and its unit under tests, browsers cannot display them side-by-side. Thus developers are not reminded to write tests, they cannot see tests side by side to the unit under test to better understand a unit, they do not know if a unit has a dedicated test, and they cannot automatically run the unit test when the unit has changed. Not being able to compose unit tests out of other unit tests, developers have to write redundant tests, which leads to code duplication and

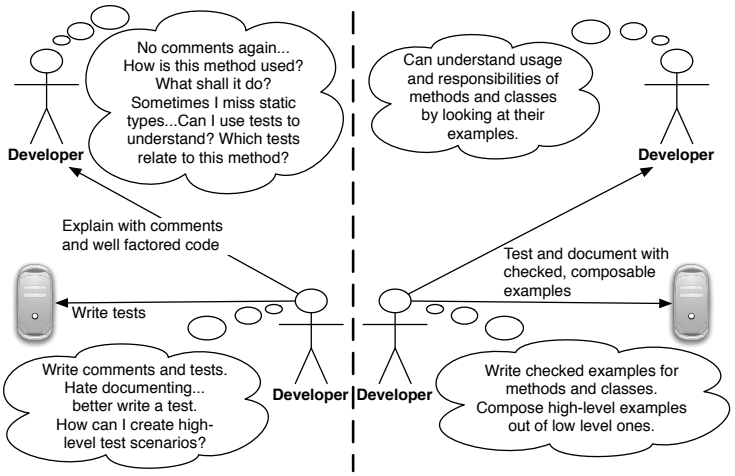

Figure 1.3: Agile developers favor writing tests over documentation, but these tests are not well integrated into languages and thus not browsable. Examples on the other hand can be used both to test software and to explain it.

unnecessarily long testing times.

Unit tests can also serve as templates of how to use the units under test. As I am writing this thesis, I do not follow an abstract thesis writing process, but I rather adapt a successful example of a thesis of our group [Schärli, 2005]. Having a template or example of a thesis at hand facilitates the task of writing a thesis: Whereas my template thesis explains the problems of multiple inheritance and how to solve them with traits, my task is to explain the problems of missing examples of how to run and test object-oriented programs and how to solve them with commands.

There is no rigorous definition of a unit test, how it is structured or what level of granularity it should address: IEEE defines unit testing as

> Testing of individual hardware or software units or groups of related units. See also: component testing; integration testing; interface testing; system testing.

[ANS, 1983]

Hence developers have written all kinds of tests with different levels of granularity and scope. But not denoting the unit under test explicitly makes unit tests unusable as examples, thus developers cannot use them to understand how the units under test should be used.

In addition to not being able to compose unit tests out of other unit tests, developers have to write redundant tests, which leads to code duplication and unnecessarily long testing times.

Furthermore unit tests are not orthogonal to static typing: Test-driven developers using a statically typed language have to provide the types twice: When they develop the test they need to come up with concrete instances to build the scenario for the unit under test. If the unit under test is a method, they then have to again statically type that method, though these types could have been inferred out of the test scenario.

1.2 Our Approach in a Nutshell

Our approach consisted of several iterations of analyzing existing unit test suites, designing the problems we found away into a meta-model and implementing according prototypes – always guided by the metaphor of examples.

Metaphor. We are relying on the metaphor of *examples* which is not yet fully exploited in the context of object-oriented software engineering. The metaphor of examples not only suggests a strong connection between the concept to be exemplified and the example itself, but also to create high-level examples out of low-level ones: The idea of composing tests is awkward, whereas it is natural to describe large examples out of smaller ones.

Analysis. To make a transition from the perspective of tests towards the perspective of examples, we first need to know how current unit tests are implicitly related to each other, and how they implicitly relate to the concepts of object-oriented languages.

— To understand the implicit relationships of unit tests to other unit tests we are introducing a new technique of dynamic analysis: We partially order unit tests by the sets of signatures of the methods called by the tests. Our experimentation with different case-studies has revealed that a high percentage of current unit tests is comparable to other unit tests using this technique. The partial order supports our understanding of how unit tests correlate and suggests ways in which they can be refactored. Moreover, our technique supports software debugging activities: Using this scheme we can present the developer the most relevant failing test case first together with the according methods causing the error.

— To understand the implicit relationships of unit tests to the concepts of object-oriented languages such as classes and methods, we researched case studies, some

of them consisting of more than thousand unit tests, and iteratively built a categorization scheme based on the relationship of the tests to the atomic unit of object-oriented programming, the method.

Design. We can model the relationships found in our analysis-phases explicitly by exploiting our metaphor of examples. This model is called a meta-model, as we are not modeling outer-world concepts but introduce new concepts of object-oriented programming languages themselves. Our meta-model centers around the concept of composable *method commands* which serve as dedicated exemplifications of one method. In Figure 1.4 (p.6) you can see a condensed version of our meta-model based on the concept of method commands. As you can see *method commands* serve the purpose of exemplifying methods and classes, they can be stored into readable code or reified from this code, and they are composable to form higher-level commands.

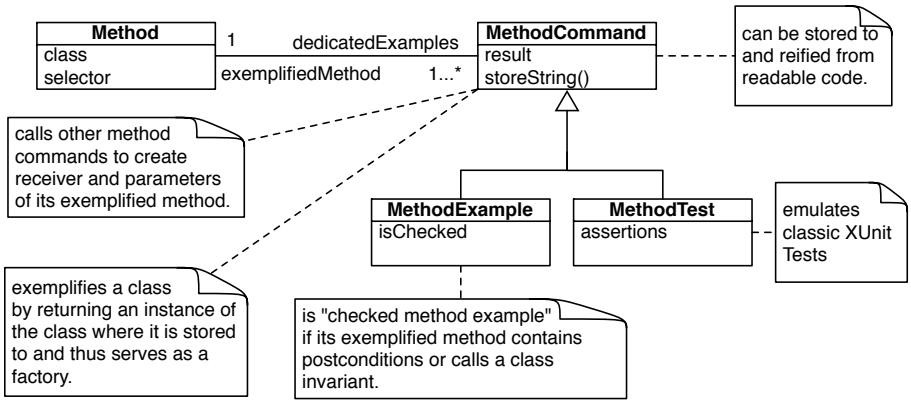

Figure 1.4: A condensed version of our meta-model: Each method can have an example stored in a method command. These commands deliver example instances of their classes and thus can call each other to build higher level test scenarios. Commands can also be stored into and reified from source code.

We explain how commands can be used to exemplify classes, methods, and variables. We also show how they can be composed, and that every object can either be expressed as a literal or as a result of a method command.

Implementation. We present a light-weight implementation of this meta model in Smalltalk and finish by showing how our prototypes using these meta model can help developers to test and understand software.

Thesis statement. *Tests are neither composable nor explicitly bound to the concepts of current object-oriented languages. The reason is that tests are only linked implicitly to their unit under test and to other unit tests. Detecting these implicit links and making them explicit using the guiding metaphor of examples helps developers to document, type, debug and test object-oriented programs.*

1.3 Contributions

The main contributions of this thesis can be summarized as follows:

1. A detailed illustration and analysis of the various problems that are associated with the absence of linkable and composable examples for the building blocks of object-oriented programs.
2. A detailed analysis of unit tests and their interdependencies with each other and with the code under test:
   (a) A new and simple trace summarization technique: First we flatten the signatures of the traced methods of unit tests into sets. Then we use a partial order mechanism to compare these sets. It turns out that a high percentage of unit tests becomes comparable using this simple mechanism. We show [Gaelli *et al.*, 2004a] how the resulting partial order can be used to understand and refactor the code with its test suites and to prioritize failing tests.
   (b) A taxonomy of unit tests with respect to their units under test and to their composability. Our case studies reveals that most unit tests are either method commands focusing on one method or are decomposable into such method commands. [Gaelli *et al.*, 2005b]
3. Based on this analysis a suggestion of a new meta model to include unit tests from the very beginning into the development cycle and environment:
   (a) A light-weight implementation for making missing test links explicit [Gaelli *et al.*, 2004b] [Gaelli, 2004] [Gaelli *et al.*, 2005a] together with an implementation in Smalltalk.
   (b) The development of a meta-model abstracting from our implementation.
   (c) The development and evaluation of a programming methodology around this meta-model exemplified by a tool called “Eg”.
   (d) Suggestions how to implement this meta-model in light ways in other languages such as Java, Ruby and Python.

1.4 Thesis Outline

This dissertation is structured as follows:

Chapter 2 (p.9) identifies and illustrates problems of missing examples in object-oriented programs by establishing a bank example which will be used throughout this thesis.

Chapter 3 (p.29) analyzes the interdependencies among unit tests. It presents a technique to partially order unit tests in order to understand, debug, and refactor existing test suites.

Chapter 4 (p.45) analyzes the interdependencies of unit tests with their units under tests using several case studies, one of them consisting of more than 1000 tests and establishes a taxonomy of unit tests. It also introduces first heuristics to categorize unit tests automatically.

Chapter 5 (p.67) establishes our meta model *Eg* for linking unit tests with the code under test and with other unit tests. It first describes a light way implementation of the meta-model in Smalltalk. It then introduces the meta-model in detail by explaining the responsibilities of the classes of our meta-model – illustrated by use cases to test, debug and understand the Bank example. It finishes by pointing out how the meta-model solves the problems described in Chapter 2 (p.9) and gives an outlook how to implement Eg in languages like Ruby and Java.

Chapter 6 (p.97) summarizes how the main results of our work support the statement of the thesis, and we look forward to future work related to *Eg*.

Chapter 2

Problems in Understanding and Testing

Software understanding and testing is a wide field and so in this thesis we are researching problems of understanding and testing software only in the context of agile development. Agile developers aim for simplicity in order to translate the problem domain into tested and understandable code. Coming from an object-oriented background, they heavily rely on automated tests while doing so. The main software related tasks for agile developers are to write new tests and to enhance and to debug the the software. For writing new tests developers spend most of their time in creating tests scenarios, for debugging they spend most of their time in isolating the failure-inducing change and for developing they spend their time in understanding the existing code-base. But current object-oriented programming environments neither support agile developers to create scenarios for high-level tests out of existing ones nor to debug or understand code by using tests as composable examples. The reason for this lack of help from programming environment is that tests are neither composable nor explicitly bound to the concepts of current object-oriented languages. In this chapter we define the scope of this thesis not only by specifying the problem, but also the "owners" of the problem – agile developers – and the context they are living in. We do so by

— *explaining the motivations and constraints of lightweight agile methodologies,*

— *examining the minimal set of concepts agile developers are relying on, namely tests and code,*

— *defining a set of constraints that code and tests have to fulfill to keep software evolvable in an agile environment,*

— *examining the problems arising from implicit interdependencies of tests and code.*

2.1 Agile Development aligns Viewpoints of Customers and Coders

The quest for fighting unnecessary complexity continues. Though Brooks argues that there is no "silver bullet" in software engineering [Brooks, 1987] and Nierstrasz [Nierstrasz, 2002] emphasizes that "as long as industry remains focused on short-term goals, and maintains a technology-centric view of software development, no progress will be made", industry continues to build new technology-oriented solutions and to buzz these solutions to be "the next great thing" Figure 2.1 (p.10).

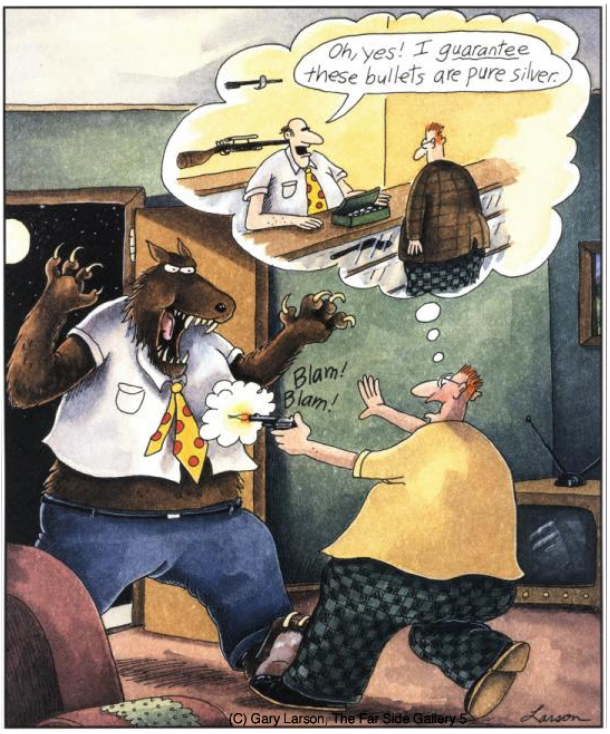

Figure 2.1: *"Moments before he was ripped to shreds, Edgar vaguely recalled having seen that same obnoxious tie earlier in the day."*

> Any problem in computer science can be solved with another level of indirection. But that usually will create another problem. — David Wheeler [Wikipedia, 2006]

Often only the first sentence is getting quoted. This observation fits well with another observation, namely that

> Software people tend to favor the joy of complexity, yet we should strive for the joy of simplicity. — Alan Kay

The agile development movement as explained by Cockburn [Cockburn, 2002] can be seen as the result of technically overcomplex solutions: Agile developers and their customers focus on a rapid feedback cycle between them and the code by removing as many unnecessary "silver bullets" or artificial artifacts imposing unnecessary levels of indirections as possible. [Fowler and Highsmith, 2001] They strive for the "Joy of Simplicity". Accordingly one of the mantras of the agile movement is "Do the simplest thing, that could possibly work"[1].

Many recent efforts like spreadsheet testing [Rothermel *et al.*, 2001], Dabble DB[2], Etoys [Allen-Conn and Rose, 2003], Test-Driven Development or "Naked Objects" [Pawson and Matthews, 2002] are agile in the sense that they focus on "end-user programming". All these efforts not only aim to align the viewpoints of end-users with the viewpoints of developers, they even want to empower end-users with abilities to do the real programming (see Figure 2.2 (p.11)).

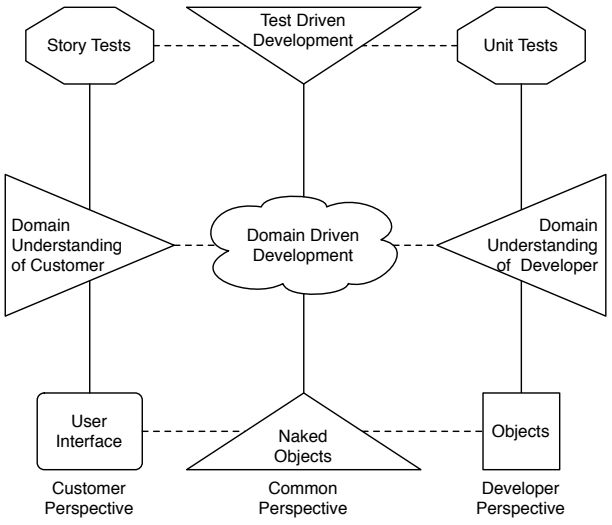

Figure 2.2: Recent methodologies align the viewpoints of the customers and developers to the *same model* using the *same tests* and the same user interface.

[1] The Simplest Thing that Could Possibly Work – A Conversation with Ward Cunningham, Part V, by Bill Venners, www.artima.com/intv/simplest.html

[2] http://www.dabbledb.com

Let us have a more detailed look into the two complementing agile methodologies, namely domain-driven design and test-driven development.

Domain-Driven Design. Domain-driven design (DDD) as described by Evans [Evans, 2003] tries to bridge the existing gap between domain understanding and software implementation decisions: It questions the traditional separation between software design and requirements analysis. Domain-driven design uses only one model "throughout all aspects of the development effort, from code to requirements analysis". This model, at the heart of software, facilitates communication with users. It also serves as the implementation guide for programmers. Not separating concepts from their implementation provides a "ubiquitous language" for all project stakeholders, and even improves software usability, since the user's model will match the programmer's model of the system. Domain-driven design's defining characteristic is its "priority on understanding the target domain and incorporating that understanding into the software", where some of its elements closely correspond to elements in the model.

Test-Driven Development. Beck and Cunningham collaborated while inventing *pair-programming* but went in different directions when it came to testing: Whereas Beck invented the Xunit Test framework and promoted "Test-Driven Develolpment" [Beck, 2003] for developers, Cunningham focused as usual [3] on the collaborative aspect and built a tool called *FIT* [Mugridge and Cunningham, 2005b]. *FIT* (Framework for Integration Testing) allows customers (with the help of developers) to build high-level story-tests which in turn can be used by developers as concrete specifications to implement against. As soon as the story test is running, the customer can see a green light in a web-interface thus tracking the progress of the developers.

Agile projects try to align the viewpoints of customers and developers on the problem domain by minimizing the set of concepts used. That minimal set of concepts used in test-driven development which are also represented as programs consists of story tests, unit tests and code.

2.2 Constraints of Test-Driven Development

In this thesis we are focusing on the minimal set of programmable concepts necessary for test-driven development coming from a domain-driven design perspective.

[3] Ward Cunningham became famous for having invented the Wiki-Principle of collaborative Website Editing best exemplified by the Wikipedia

Customers are specifying their requirements iteratively with story tests, possibly with the help of developers. The developers then iteratively translate these story tests into unit tests and code as described by Beck [Beck, 2003]. We will not explore the role of user stories [Cohn, 2004] which are an integral part of development in eXtreme programming as they are not put into an electronic format.

Whereas customers deal with writing high level storytests[4], developers help them doing so. Developers also have to understand these storytests in order to *compose in* lower level unit tests and appropriate code to satisfy both the story- and the unit tests. The computer on the other hand should always be able to run the minimum set of *relevant* tests after a change in the code, where relevant means that the test is calling the code.

We thus end up with three different "users" of tests and code: customers, developers and the computer as seen in Figure 2.3 (p.14). We subsume the role of the customers with the role of the developers in the context of domain-driven design: In our experience it is still common that customers do not write storytests and let developers do this work, or at least help them doing so. Having unified our target audience we also unify the tests they dealing with, namely storytests and unit tests into the concept of tests.

Although we do not claim that the following goals of agile test-driven development are reachable, we however do claim that they represent the utopia test-driven developers are aiming at, and that our approach is a contributing factor to approaching this goal.

We say that the set of *agile concepts* (tests and code) are *minimal and sufficient* to describe an evolvable system implementing some external requirements if

— all possible branches of all requirements are tested by a storytest,

— there exist no redundancies within tests and code,

— all storytests either test if a requirement is met or are indirectly used by a storytest which tests if a requirement is met,

— all code is covered by at least one storytest ,

— all unit tests are used by storytests [5],

The set of agile electronically available concepts does not include any kind of classic documentation. Thus it is crucial, that the tests and the code base are as understandable as possible. We say that the set of *agile concepts* (tests and code) are *perfectly understandable*

[4] Storytests are also known under the name of acceptance tests

[5] We take the perspective that a system only has a bug, if that bug is reachable by the end-user of the system. Thus we question the value of standalone unit tests and claim that each unit test can be part of a high level storytest belonging to a user story, which specifies this situation.

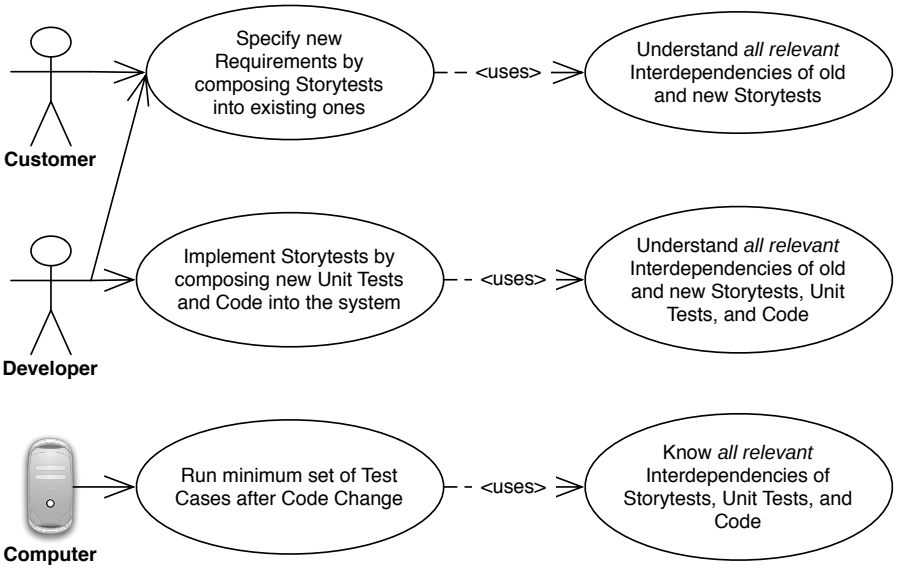

Figure 2.3: Customers and developer are composing new storytests, developers new unit tests and code into an existing system and computers should run the minimum set of relevant tests after a code change. All parties can only fully understand relevant artefacts like tests and code, by understanding the interdependencies of these artefacts.

— if the concepts are *minimal and sufficient*,

— if agile team members can be aware of all *interdependencies* between the concepts they have to deal with.

— if the computer can automatically run all *relevant* tests after a code change.

In the rest of this chapter we take a detailed look at all possible interdependencies of *agile concepts* as shown in Figure 2.4 (p.15) and the problems of comprehensibility and local testability arising from the fact, that they are implicit. We will discuss the current approaches dealing with these problems in the appropriate chapters.

2.3 Problems of Implicit Test Interdependencies

We motivate the problems arising from implicit test interdependencies by introducing the canonical example of a bank account as depicted in Figure 2.6 (p.17). Typical user stories here are *Create Bank*, *Create an account*, *Deposit money* and *Withdraw money*.

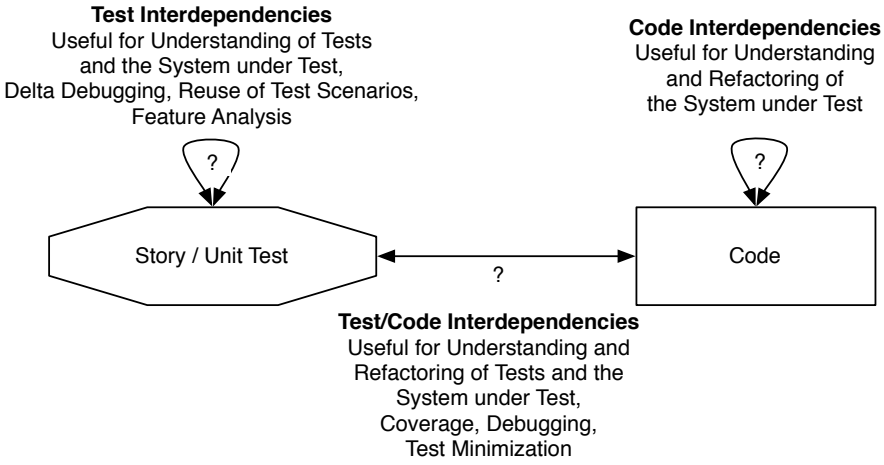

Figure 2.4: A transitive reduction of all possible interdependencies between agile concepts. None of these interdependencies is explicit, which causes problems in understandability of the concepts.

The order of the stories defines the prerequisite relationing – *e.g.*, an account must exist before money can be deposited, similarly the money is withdrawn only after the money has been deposited. To better understand this example which will follow us throughout this thesis, we show the code containing an implementation of this example in Smalltalk in Figure 2.5 (p.16).

2.3.1 Creating test scenarios is time-consuming and complex

More than 50% of the effort of writing tests is spent writing the test scenarios ([Zeller, 2005]). Whereas it is easy to develop low level unit tests, which use only scalar data types such as numbers or strings, the higher the level of the class under test, the more complex it gets to create appropriate scenarios. Creating instances of the required input parameters is more complex, as these parameters depend recursively on other objects in the object net.

Example. Consider as an example the Account » withdraw method in the bank account as seen in Figure 2.6 (p.17). To test this method it is necessary to first create a bank and an account which already has money deposited on it. Having no instances ready for use in complex tests is the main reason why writing tests of high level objects is such a time-consuming task.

```
Object subclass: #Bank.Model.Bank
 instanceVariableNames: 'accounts'
 classVariableNames:''
 poolDictionaries:  ''
 category: 'Bank.Model'

Bank.Model.Bank>>initialize
 accounts := Dictionary new.
 ^self

Bank.Model.Bank>>accounts
  ^accounts values

Bank.Model.Bank>>accountNumbered: aNumber
 ^accounts at: aNumber

Object subclass: #Bank.Model.Account
 instanceVariableNames: 'balance number'
 classVariableNames:''
 poolDictionaries:''
 category: 'Bank.Model'

Bank.Model.Account class>>numbered: aNumber
  ^self new initializeFor: aNumber

Bank.Model.Account>>balance
  ^balance

Bank.Model.Account>>withdraw: someAmount
 self assert: [self canWithdraw: someAmount].
 balance := balance - someAmount
```

```
Kernel-Classes.Behavior(Bank class)>>new
 ^self basicNew initialize

Bank.Model.Bank>>createAccount: aNumber
 lanAccountI
 self assert: [(self includesAccount: aNumber) not].
 anAccount := Account numbered: aNumber.
 accounts
  at: aNumber
  put: anAccount.
^self

Bank.Model.Bank>>includesAccount: aNumber
 ^accounts includesKey: aNumber

Kernel-Classes.Behavior(Account class)>>new
 ^self basicNew initialize

Bank.Model.Account>>initializeFor: aNumber
 number := aNumber.
 ^self

Bank.Model.Account>>deposit: someAmount
 balance := balance + someAmount

Bank.Model.Account>>number
 ^number

Bank.Model.Account>>canWithdraw: someAmount
 ^balance >= someAmount
```

```
Bank.Tests.TestCase subclass: #Bank.Tests.AccountTest
 instanceVariableNames: 'bank account'
 classVariableNames: ''
 poolDictionaries: ''
 category: 'Bank'

Bank.Tests.AccountTest>>testDeposit
 account deposit: 100.
 self assert: account balance = 100.


Bank.Tests.AccountTest>>testAccount1234Exists
 self
  should:[bank createAccount: 1234]
  raise: Exception

Bank.Tests.AccountTest>>testDepositAndWithdrawAll
 account deposit: 100.
 account withdraw: 100.
 self assert: account balance =0.
```

```
Bank.Tests.AccountTest>>setup
 bank := Bank new createAccount: 1234.

 "Developers do put assertions in the setup"
 self assert: bank accounts isEmpty.

 account := bank accountNumbered: 1234


Bank.Tests.AccountTest>>testWithdraw
 account deposit: 80.
 account withdraw: 30.
 self assert: account balance = 50.

Bank.Tests.AccountTest>>testWithdrawTooMuch
 self account deposit: 80.
 self
  should:[account withdraw: 100]
  raise: Exception
```

Figure 2.5: The canonical bank account in Smalltalk and some typical SUnit tests for it. Bank.Model.Bank denotes the class Bank within a package called Bank.Model.

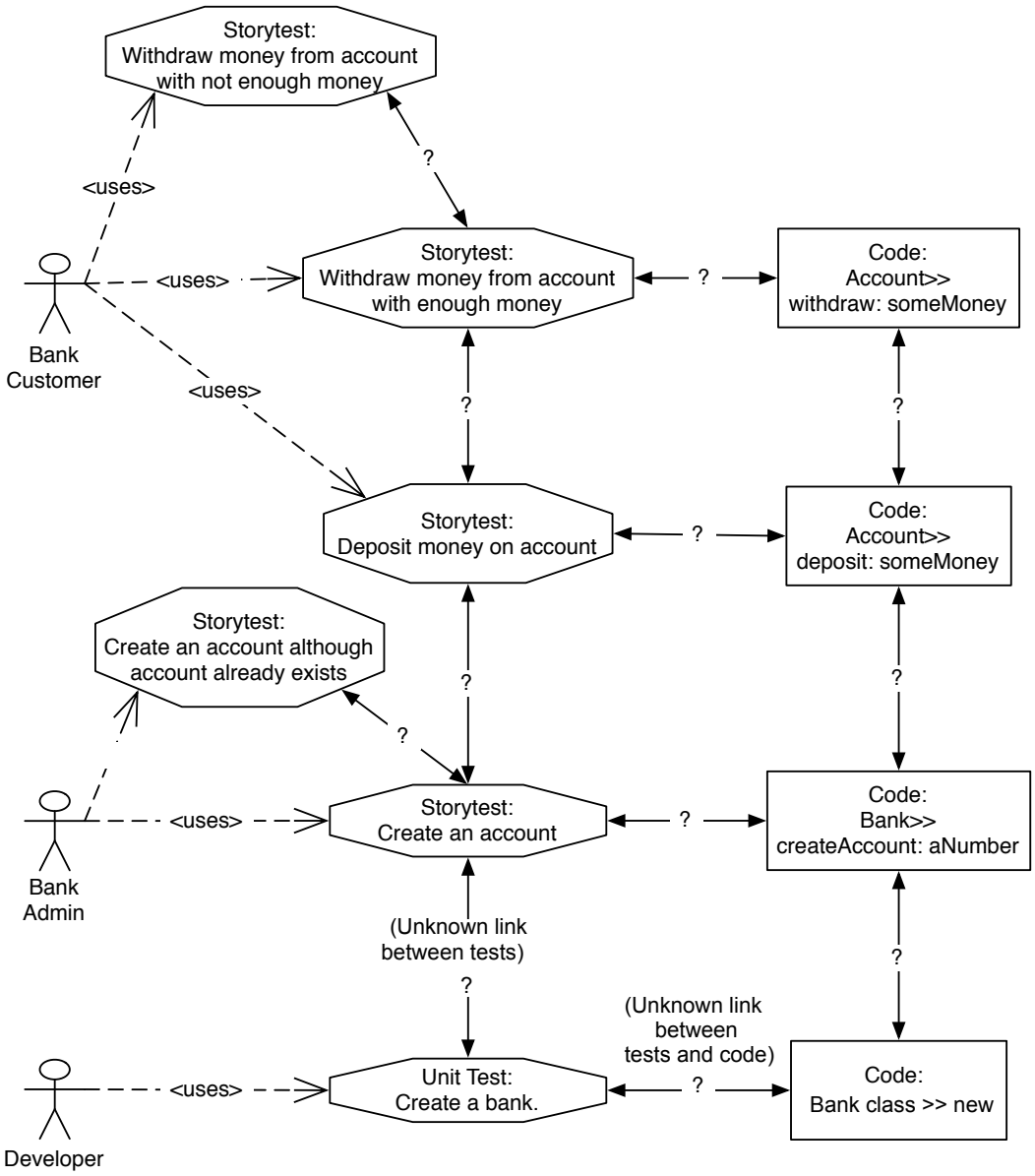

Figure 2.6: The canonical bank account: The bank customer can deposit and withdraw money, the bank admin can create an account, but for doing so a bank object must be created first. The relations between these concepts are not reflected in the system which hinders its comprehensibility and thus its evolvability.

2.3.2 Understanding the interplay of a system is hard

User stories only make sense in the context of the other user stories which either are required by or depend on them – and thus the according storytests also only make sense in the context of other tests: Dependency graphs as seen in Figure 2.7 (p.19) would help developers to understand the functionality and their according tests in context.

It is possible to build these kind of dependency graphs using a technique called gui ripping as described by Memon *et al.* [Memon *et al.*, 2003] but this technique requires user interaction with the system. To our knowledge there is no way to *automatically* infer them out of the running systems. These graphs are easy to create if the interdependencies of the test were explicit.

We take the perspective that unit tests describe low level use cases: Each functionality a developer has to write can be seen as a use case, though often a technical one. Be it connecting to a database, filtering some data or processing it somehow, none of these functionalities are treated different in the mind of a programmer to a high-level functionality like transferring money from one account to another, which just gets decomposed into smaller unit cases.

These arguments above apply also to the viewpoint of the developer writing unit tests: To understand a certain functionality developers need to put this functionality into context of all prerequisite functionalities. If they write or read functionalities, writing or reading the unit tests for the required functionalities helps them not only to create test scenarios but also to understand the context this functionality is living in.

Example. If developers of a bank system were able to have a glance at an *automatically generated* dependency graph as seen in Figure 2.7 (p.19) where the test interdependencies of the system are show together with the interdependencies of the functionality using concrete examples, they could directly infer that *e.g.,* one needs to deposit money into an account before being able to withdraw money from it.

2.3.3 Testing time is unnecessarily long

Having a set of storytests which are only implicitly calling each other tempts the developers to recreate test scenarios for each of them, although smaller storytests already provide the fixture to be used by other tests. This may lead to unnecessary long testing times.

Example. In the case of our example bank system a bank has to be created before being able to create an account on it (Figure 2.6 (p.17)). Writing two independent storytests for this clearly leads to an unnecessary increase of testing time as opposed to a cascading

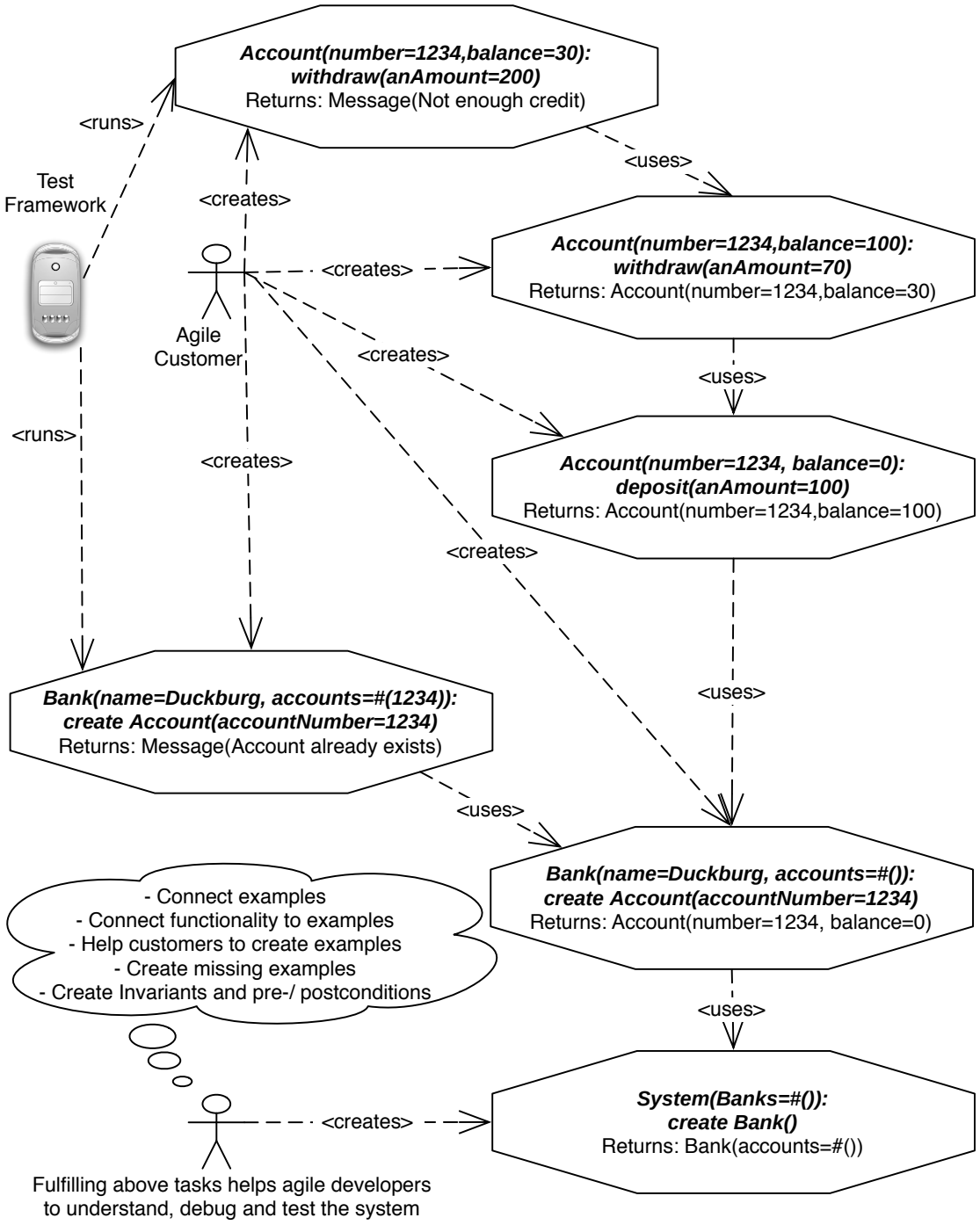

Figure 2.7: The canonical bank account described in an example notation. The additional tasks for agile developers who take this point of view are depicted. We will show in the rest of this thesis, that fulfilling these task can be easily integrated in current development styles.

style where the quality assured bank object can be reused for further test scenarios. (See Figure 2.5 (p.16)).

$$Testtime_{independent} = createBankTest_t + createBankFixture_t + createAccountTest_t > Testtime_{cascading} = createBankTest_t + createAccountTest_t \quad (2.1)$$

2.3.4 The problem of identifying relevant tests in the case of a failure

One fault can cause several unit tests to break, so developers do not know which of the broken unit tests gives them the most specific debugging context and should be examined first.

Example. Suppose the developer accidently changes addition into subtraction in the Account » deposit: someMoney method. (See Figure 2.5 (p.16)) Clearly the according test AccountTest » testDeposit will fail. But this will not be the only test failing! To withdraw money, one first has to deposit money, accordingly the correctness of the Account » deposit: someMoney method is a prerequisite for testing the correctness of the Account » withdraw: someMoney method. Otherwise the tester will assume an initial balance of the account of 80 to withdraw from, whereas is actually has a balance of -80 leading to a result of -110 – and not of 50 as expected. In our trivial example it is clear which of these failing tests should be attacked first. But facing a more complex domain, which the developers possibly have not built themselves, they might be misled and try to fix a test first, which is testing a perfectly well functioning method. Thus would lead to an unnecessary proliferation of debugging time.

2.3.5 The problem of detecting similar tests

Developers are free to write one test which tests a unit under several circumstances, or, as an alternative, to write several tests, each focusing on the same unit but in several scenarios. Developers who are reading these tests though are interested, if there are other tests which focus on the same unit, but they only can detect them by reading code and relying on brittle naming conventions. If a refactoring of code and tests becomes necessary, it is desirable to present the developers all related tests automatically.

Example. In our example the tests AccountTest » testWithdrawTooMuch and AccountTest » testWithdraw represent two tests focusing on the same unit under test: Whereas the first

is testing the boundary condition, the second one is testing the default case. It is only the brittle naming conventions and the fact that they are residing in the same test class which connects those two siblings.

2.4 Problems of Implicit Test / Code Interdependencies

Developers have written all kinds of unit tests using xUnit: some of them focus on one method, sometimes testing it in one and sometimes in several scenarios, some of them focus on several methods at the same time, some of them assure, that the method under test throws an exception if used in a wrong context, and some of them just call a method under test without checking its outcome.

As a consequence, tests are not only related implicitly to other tests as explained before, but also to their units under test. In the following we describe several problems arising out of the fact, that the relationships between tests and their unit under test are *implicit*.

2.4.1 Understanding the focus and the kind of a test is hard

Developers who are reading unit tests not only have to decrypt the focus of the test in order to understand what functionality the test is testing, but also the type of the test in order to understand to what degree it is testing the functionality. Typical questions the developers have to ask themselves include:

— Are the called methods in the test independent from each other as the test tests several independent aspects of a unit under test?

— Or is each method call required to successfully call later methods in the test?

— Is the test checking if the unit under test is working correctly after some valid input state has been established?

— Or is it ensuring, that the unit under test fails correctly, if its precondition is violated?

This is cumbersome, time consuming and could be avoided by making both the unit under test and the type of the test explicit in a meta-model.

To detect the focus of a unit test, one must understand what unit the unit test is focussing on. Is it a method, a bunch of methods, a class, or a bunch of classes? Currently the only name developers have to describe a unit test besides “developer test” is “unit test” itself, adding much to the confusion in this area. Using names like “unit”,

"thing", "entity" is a bad sign in object-oriented programs and means that the "unit" is not yet well understood.

Like design-patterns [Gamma *et al.*, 1995] allow developers to spot, communicate and reuse recurring solutions for recurring design problems, a well based nomenclature and description of the several kinds of unit tests should help developers to understand existing test suites. Having established such a terminology one could also encode the kind of unit test explicitly in the unit test itself. This is impossible without having a meta-model for unit tests.

The only hints of the xUnit framework for detecting the unit under test is the naming convention of the test classes and methods, which often implies the signature of the method under test. But this naming convention is brittle, as it is not stable to refactorings like renaming the method under test or changing the contents of a test. Van Deursen *et al.* [Deursen and Moonen, 2002] 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.

Example. Is the test AccountTest » testWithdrawTooMuch in Figure 2.5 (p.16) testing both Account » deposit: and Account » withdraw:? Though the name of the test implies the latter, developers are free to test the former one also by inserting some assertions after calling Account » deposit.

Developers have to *imply* that this test is bound to succeed if it's call to it's method under test fails. They can do so by either looking at the exception catching mechanism or by reading the name of the test. Though it is a small mental step to do so, it still takes time to read the test. Developers cannot yet rely on a *summary* which implies that this test (1) serves as a bad example by ensuring that a functionality breaks accordingly if called in a wrong situation, and (2) focuses on Account » withdraw:.

2.4.2 The problem of separating good examples from less appropriate ones

Beck points out the value of unit tests as documentation of their units under tests [Beck, 2003] but how can a unit test document a unit, if the relationship between the unit test and the unit under test is implicit? The only relationship between the unit test and a unit under test a developer using the xUnit framework can detect right now, is if the xUnit test calls a given method. The reader of the test has to bring in lots of contextual knowledge to decide, whether the test is using a given method only to setup a scenario,

if it is used for checking a desired state an object should be in, if it is used to clean up the test scenario, or if – indeed – it is the method which is in the focus of the test.

Wittgenstein [Wittgenstein, 1953] and Lakoff [Lakoff, 1990] argue, that not all examples are equal – it depends on the context, if a given example is a good example or not. It would be a bad idea to explain a child the concept of a bird using a penguin as a first example. Only after the child has some knowledge about typical bird-examples like seagulls, eagles and pigeons, one might want to introduce the example of a penguin as a typical non-flying bird.

Example. For understanding the functionality of Account » deposit: the developer should be pointed first to the test(s) focusing on this functionality, which would be AccountTest » testDeposit:. Certainly the developer should be also able to see how this functionality is used in different contexts as like in AccountTest » testWithdraw: or AccountTest » testWithdrawTooMuch: but pointing the developer first to a classic example of this functionality before pointing to some derived functionalities speeds up the process of comprehension.

2.4.3 The problem from separating tidy from untidy examples

We say that an example is *tidy* if its execution does not create any left-overs like open windows or changed global state. Currently developers do not make a distinction between tidy and untidy examples. Though it is useful to have untidy examples in order to open application in an exemplified context and thus to understand them, these untidy examples are neither composable nor can they be easily executed automatically.

Example. Imagine the following code snippet:

```
Account class≫editAccount1234
  Bank exampleAccount1234 openInWorld
```

The above would be an untidy example as it would leave open a window showing the account in action whereas Bank class≫exampleAccount1234 would be a tidy example which could be used for testing purposes.

2.4.4 The problem of keeping the tests and the code synchronized

Saff *et al.* [Saff and Ernst, 2003] suggest to continuously run all unit tests in the background, so that the developers do not have to run them explicitly. On the one hand this still leaves lots of functionality uncovered as it is not known if a method is covered by a test at all. On the other hand this brute force algorithm spends lots of unnecessary cycles to check unchanged functionality.

If methods were explicitly connected to their executing tests, one could only rerun those: Also Winger suggests to connect methods and tests via pragmas in VisualWorks Smalltalk, so that after a changed method has been saved, the according test can be run automatically [Winger, 2004].

On the other hand, test cases can be changed, too. As soon as a test is added, changed or deleted the relevant code under test should be exercised in order to assess if the code coverage stays the same and if the code complies with the current set of tests.

Test-driven development as introduced by Beck [Beck, 2003] suggests to first write a unit test and then to write the method(s) to implement this unit test. Thus the first test-driven session to implement some methods can be seen as a debugging session: The task of the developer is to enhance the method in a such a way, that the former failing unit test focusing on this method does not fail any more.

Example. Changing the functionality of Account » deposit should trigger all tests which call Account » deposit – directly or indirectly. These would be: AccountTest » testDeposit, AccountTest » testWithdraw and AccountTest » testWithdrawTooMuch.

2.4.5 The problem of seeing a method in a debugger

One classic way to debug and understand some methods is to insert a breakpoint into them and then to *hope* to know an executable entry point or command which would trigger this breakpoint during its execution. Squeak developers recently introduced a debug item into the classical do it, print it, inspect it dialog with which any command can be inspected in action in a debugger. But as long as there is no explicit knowledge about which tests execute which methods along the way, this debug menu item cannot be included into the context menu of *any* method. As a consequence developers have a hard time to see *any* method within the debugger. Not being able to see a method in the debugger requires developers to understand methods in an abstract way, *e.g.*, they often have to guess the actual callers of a possibly polymorphic method in a special use case and cannot step through its actual concrete behavior.

2.5 Problems of Implicit Code Interdependencies: Typing

Test-driven developers using a dynamically typed programming language only provide the types of the parameters *implicitly*. In theory a program could detect the types for all variables of all classes instantiated by the tests and also the types of parameters and

return values of all methods executed by the tests. But to our knowledge no tool exists until today to exploit tests in the context of dynamically typed languages to also infer the concrete types for those entities mentioned above.

Test-driven developers using a statically typed programming language have to provide the type[6] of variables and parameters twice – once explicitly when they define the class or method – and once implicitly when they provide concrete parameters and fill the instance variables in order to set up the test scenarios.

We believe that the reason for this is that there exists no meta-model for unit tests which makes it impossible to exploit unit tests for providing the type information of their units under test although all the necessary information is just lying there waiting to be picked up: If we connect a unit test with alls methods executed by it in an explicit way, not only the parameters of these methods can be exemplified and thus also concretely typed, also the type of the return value of these methods can be deduced, as at least one concrete value could be computed along the way when the test gets executed.

As test coverage in current systems is rarely 100%, the approach of using dynamic information for typing has not been followed yet – but with the rise of test-driven development one could easily imagine check in policies that would only allow methods to be stored which are actually covered by at least one repeatable command. The same reasoning applies for the storage of classes: A class-definition could only be checked into the code-repository if there is at least one way to create an instance of this class – or in the case of abstract classes one of its concrete subclasses –, where all its variables are initialized with a concrete object.

2.6 Related Work

Examples are heavily used in all fields of education such as mathematics where Mason *et al.* find that examples help students to stay focused on the problem at hand [Mason John H., 2001]. Watson *et al.* point out the effectiveness of student-generated examples as a teaching tool in [Watson Anne, 2002]. They examine the roles played by examples constructed and generated by students, illustrate and analyze the use of this tool, and develop a theory for the act of exemplification as an act of cognition.

Examples are used for learning programming: Anderson *et al.* describe in [Anderson *et al.*, 1984] an example-based model of instruction, with practice and feedback to help abstract from the examples. They suggest that students have heuristics that help them generalize from examples to idioms. Problem solutions are learned and reused when

[6] With *typing* we mean specifying the concrete types of variables, parameters and return values at development time.

like situations arise. Anderson's theory predicts that students will learn idioms and emphasizes the need for a complete repertoire of idioms. Likewise a pattern [Gamma *et al.*, 1995] only is a pattern if at least manifested three times by several developers. Summers researched an example-driven way of Lisp programming [Summers, 1977] and describes a construction methodology which consists of a series of transformations from the set of examples to a program satisfying the examples. Lieberman and Hewitt interleave programming and testing in Lisp with a system called *tinker* [Lieberman and Hewitt, 1980]. Nievergelt *et al.* emphasize the value of examples for learning and understanding examples in computer science education in [Nievergelt, 2006].

Cunningham suggests to let the customer type in names so that the developer could compose the tests if wanted. [Cunningham, 2006] Mugridge *et al.* [Mugridge and Cunningham, 2005a] deal with the problem of composing storytests but they focus on minimizing testing time. They suggest to let the computer detect the interdependencies of story tests and use a hill climbing algorithm to minimize the testing time of the resulting *cyclic* graph. But considering use cases or according storytests to be cyclic runs against the grain of conventional use case building and thus adds a level of unnecessary complexity.

Recently Mugridge has built a web based test tool called FitLibrary [7] based on the Fit Framework from Cunningham [Mugridge and Cunningham, 2005b]. FitLibrary enables customers and developers to compose story tests by composing the according tables.

Schuh *et al.* [Schuh and Punke, 2001] describe a so called mock framework called "Object Mother" to create scenarios which are difficult to set up otherwise – as other tests are not reused for creating more complex ones. Freeman *et al.* [Freeman *et al.*, 2004] emphasize the difference between mocks and stubs and explain that stubs are used to create scenarios otherwise difficult to build, whereas mocks are used for behavioral testing. Behavioral testing is concerned with checking if certain methods are called, and that the methods are called in the right sequence.

Use cases can be seen as user stories and identifying how features relate in order to generate hierarchical tests can also happen after deployment as described by Memon *et al.* in [Memon *et al.*, 2001] and [Memon *et al.*, 2003].

Greevy *et al.* [Greevy *et al.*, 2006] research the interdependencies of features and code by compacting traces into simple sets of source artefacts in order to detect the evolution of features.

Test Mentor, a commercial test framework from SilverMark, allows customers to create user interface tests and developers to create unit tests. Those tests can be combined

[7] http://fitlibrary.sourceforge.net/

by developers but there is no notion of examples. Tests are not integrated into the IDE and are also not directly composable. Instead they suggest developers and end users to store interim results into some variables of the test framework, so that testing states can be used in higher level scenarios.

Alistair Cockburn [Cockburn, 2002] [Cockburn, 2003] talks about "Dos Equis-Driven Design", a wordplay to be able to spell out XXD, meaning "eXectuable eXample-Driven design" [Cockburn, 2006]. He emphasizes the mantra of "TDD" (Test-Driven Design), that "Test-Driven Design is not about Testing" but about learning and designing programs by writing examples first.

Lieberman [Lieberman, 2001] introduced a methodology called *Programming by example* which is also called "programming by demonstration". It is is a technique for teaching the computer new behavior by demonstrating actions on concrete examples. The system records user actions and generalizes a program that can be used in new examples. Though also focusing on the power of examples, the goal of this methodology is different as it is aiming to use techniques of artificial intelligence to automatically produce code.

2.7 Summary

In this chapter we put our work into the context of agile methodologies. Agile methodologies aim at reducing the set of unnecessary artifacts by making the code as understandable and testable as possible, while on the other hand reducing the amount of extra, fast outdated and possibly redundant artifacts such as written documentation.

We observed two dichotomies in current object-oriented programming systems: (1) One dichotomy exists between documentation and tests. Documentation is bound to the building blocks of object-oriented programs via code comments but in a static and non-executable form, whereas unit tests are dynamic but currently not bound to their units under test. We identified the main reason for this missing link of unit tests to be the fuzziness of the term "unit test". A consequence of the fuzziness is that the units under test vary wildly in their granularity depending on the developer's view of what defines the "unit". (2) The second dichotomy exists between low-level and high-level tests: Whereas high-level functionality naturally reuses low level functionality, high-level unit-tests currently do not reuse low-level unit tests.

We gave a brief outlook to solve both dichotomies via the metaphor of examples. Examples always imply a specific unit they exemplify, are dynamic as they can be executed, and can be composed naturally into high-level examples.

We motivated the need for making links of the remaining code artifacts in agile projects explicit – links among unit tests, between unit tests and code, and among code. We therefore examined a number of negative consequences of missing explicit links between those artifacts and showed how these missing links hinder the understandability and testability of object-oriented programs.

Chapter 3

Partially Ordering Unit Tests

Since one fault can cause several unit tests to break, the developers do not know which of the broken unit tests gives them the most specific debugging context and should be examined first. We propose 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. We first explore the hypothesis that this order can provide developers with the focus needed during debugging phases. By exposing this order, we gain insight into the correspondence between unit tests and defects: if a number of related unit tests break, there is a good chance that they are breaking because of a common defect; on the other hand, if unrelated unit tests break, we may suspect multiple defects. The key to make the unit test suite run again is to identify the central unit tests that failed and thus caused a "failure avalanche" effect on many other tests in the suite. The results of four case studies are promising: 85% to 95% of the unit tests were comparable to other test cases by means of their* coverage sets *– they either covered other unit tests or were covered by them. Moreover, using method mutations to artificially introduce errors in a test case, we found that in the majority of cases the error propagated to all test cases covering it. By describing the results of several little case studies done within the Squeak environment, we then motivate that the technique of partially ordering broken tests can be helpful for program comprehension also.*

3.1 Implicit dependencies between unit tests

Example. Assume we have the following four unit tests for a simplified university administration system:

— PersonTest»testBecomeProfessorIn tests if some person, after having been added as a professor to a university, also has this role.

— UniversityTest»testAddPerson tests if the university knows a person after the person has been added to it.

— PersonTest»testNew tests if the roles of a person are defined.

— PersonTest»testName tests if the name of a person was assigned correctly.

For a detailed look at the run-time behavior of the test cases see Figure 3.1 (p.30) and Figure 3.2 (p.31).

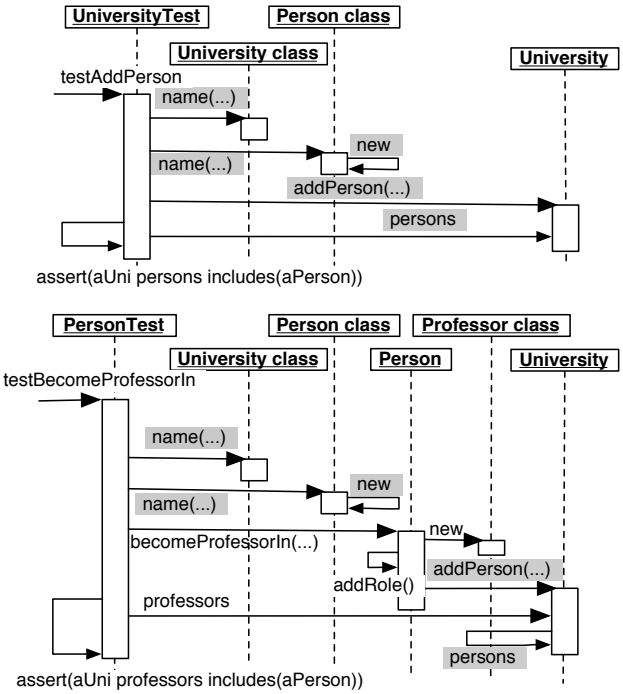

Figure 3.1: The test for #becomeProfessorIn: covers the test for #addPerson:. Intersecting signatures are displayed gray.

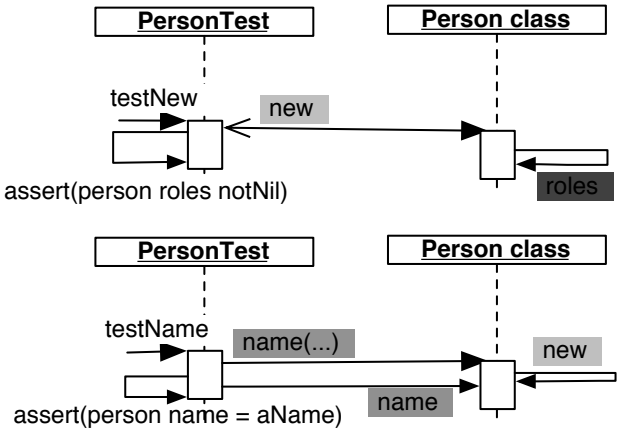

Figure 3.2: Two small unit tests, which do not cover each other.

Furthermore assume that the implementation of Person class»new is broken, so that no roles are initialized and the role variable in Person is undefined. When we run the four tests, two of them will fail:

1. The test PersonTest»testBecomeProfessorIn (see Figure 3.1 (p.30)) yields a null pointer exception: *Undefined object does not understand: add:* occurring in Person»addRole:.

2. In test PersonTest»testNew (see Figure 3.2 (p.31)) the assertion person roles notNil fails, pointing directly to the problem at hand.

As the latter failing test case provides the developer directly with the information needed to fix the error, the latter one should be presented first. We therefore order the unit tests according to their sets of covered methods. All the methods which are called in PersonTest»testNew are also called in UniversityTest»testAddPerson (see Figure 3.1 (p.30) and Figure 3.2 (p.31)). Again all methods sent by UniversityTest»testAddPerson are themselves included in the set of methods sent by PersonTest»testBecomeProfessorIn (see Figure 3.1 (p.30)). Note that PersonTest»testName is neither covered by any other test nor covering one. Consider the unit tests in Figure 3.3 (p.32). We draw an arrow from one unit test to another if the first *covers* the second. The test method PersonTest»testNew (*i.e.*, the method testNew of the class PersonTest) will invoke at run-time a set of methods of various classes. PersonTest»testBecomeProfessorIn will invoke at least those same methods, so its coverage set includes that of PersonTest»testNew. Note that we do *not* require that PersonTest»testBecomeProfessorIn invoke PersonTest»testNew, or even that it test remotely the same logical conditions; merely that at least the same methods be executed during the test run.

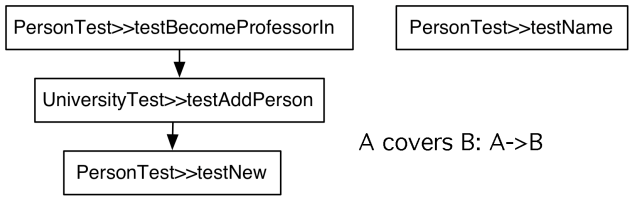

Figure 3.3: A sample test hierarchy based on coverage sets.

Unfortunately, existing unit testing tools and frameworks do not order unit tests in terms of method coverage, and do not even collect this information. In this chapter we investigate the following hypothesis: When multiple unit tests fail, the ones that cover one another fail due to the same defects. We provide initial evidence that:

— Most unit tests of a typical application are comparable by the *covers* relation, and can thus be partially ordered.

— When a unit test fails, another test that *covers* it typically fails too.

If unit tests break in the same coverage chain of our coverage hierarchy, we can infer that there is a single defect that is causing all unit tests to break. Since PersonTest»testNew is the "smallest" test (in the sense that it covers the least methods), it provides us with better focus, and helps us find the defect more quickly. In any case, the fact that these unit tests are related makes us consider them as a group in the debugging process.

3.2 Ordering broken unit tests

In this section we explain our approach of ordering in detail, and discuss an implementation in a Smalltalk environment. The problem we tackle is to infer coverage hierarchies, given a set of unit tests. We therefore need to generate traces and then order them.

3.2.1 Approach

To order the tests we used dynamic analysis because we

— have runnable test cases

— could apply it to both dynamically and statically typed languages

— and are only interested in the actual paths taken of our unit tests

The examined unit tests are all written in SUnit, the Smalltalk version of the XUnit series of unit test frameworks that exist for many languages. Our approach is structured as follows:

1. We create an instance of a *test sorter*, into which we will store the partially ordered test cases.
2. We iterate over all unit tests of a given application. We instrument all methods of the application so that we can obtain trace information on the messages being sent. The exact instrumentation mechanism to obtain the information depends on the implementation language. We used the concept of *method-wrappers* ([Brant *et al.*, 1998]), where the methods looked up in the method dictionary are replaced by wrapped versions, which can trigger some actions before or after a method is executed. Here the method wrapper simply stores if its wrapped method was executed.
3. We then
   (a) execute each unit test, in our case via the XUnit-API,
   (b) obtain the set of method signatures which were called by the test, in our case by iterating over all wrapped methods and checking if they have been executed,
   (c) check if this set is empty, which for example could be due to the fact that the test only called methods of prerequisite packages,
   (d) if the set is not empty, we create a new instance of a *covered test case*, where we store this set of method signatures together with the test,
   (e) add this *covered test case* to the *test sorter*,
   (f) reset the method wrappers, so that they are ready to store if the next unit test executes them.
4. Some of the *covered test cases* are equivalent to others as their sets of covered method signatures are equal. To obtain a partial order we have to subsume this equivalent *covered test cases* under one node, that we call an *equivalent test case*. For all equivalent *covered test cases* we create an instance of an *equivalent test case*, store the set of method signatures and the names of the equivalent test cases in it, store it in the *test sorter* and then remove the equivalent *covered test cases* out of the *test sorter*. Note that both *covered test cases* and *equivalent test cases* are *test nodes*, a superclass where we store the shared behavior of these two.
5. We then order the resulting *test nodes* stored in our *test sorter* using the following

relationship: A *test node* A is smaller than a *test node* B if the set of method signatures of A is included in the set of method signatures of B. We therefore pairwise compare the remaining *test nodes* and thus build a partial order. We store both the covering and the being covered relationship in variables of the *test node*.

6. Finally we compute the transitive reduction of this lattice, thus eliminating all redundant covering relations between the test nodes.

7. Finally we obtain an instance of a *test sorter* that we can ask which of some given tests we should attack first. Note that we did the case studies with non breaking unit tests. In the real world scenario with broken unit tests, we could either use a *test sorter*, which was initialized with the tests while they were non breaking, or reinitialize it with only the broken unit tests.

3.2.2 Implementation

In order to perform experiments to validate our claim, we implemented our approach in VisualWorks Smalltalk[1]. We chose to do the implementation in VisualWorks Smalltalk because

— tools to wrap methods and assess coverage are freely available,

— we have numerous case studies available,

— we can build on Lanza's freely available tool CodeCrawler [Lanza, 2003] to visualize the information we obtained.

We obtain the trace information by using Hirschfeld's AspectS [Hirschfeld, 2003], a flexible tool which builds upon John Brant's *MethodWrappers* [Brant *et al.*, 1998]. Though AspectS obtains the traces in the same way as method-wrappers described before, we used AspectS because it lets us obtain more detailed information about the current state of the stack, when a method is entered. In Java one would use *AspectJ* [Kiczales *et al.*, 2001].

3.3 Case studies

We performed our experiments on the following four systems, which were created by four different developers, who were unaware of our attempts to structure their tests

[1] See www.cincomsmalltalk.com for more information.

while they were writing them.

1. Wuyt's *MagicKeys*[2], an application that makes it easy to graphically view, change and export/import keyboard bindings in VisualWorks Smalltalk.
2. Gîrba's *Van* ([Gîrba *et al.*, 2004]), a version analysis tool built on top of Ducasse's *et al.* Moose Reengineering Environment [Ducasse *et al.*, 2001].
3. Renggli's *SmallWiki* ([Renggli, 2003]), a collaborative content management tool.
4. Lanza's *CodeCrawler* ([Lanza, 2003]), a language independent reverse engineering tool which combines metrics and software visualization.

3.3.1 Setup of the experiments

In a first phase, we ordered the unit tests for each case study as described in Chapter 3.2 (p.32) and measured if a relevant portion of them were comparable by our *coverage* criterium.

In a second phase, we introduced defects into the methods to validate that if a unit test breaks, its covering unit tests are likely to break as well. We therefore

1. iterated over all test cases of the case study that were covered by at least one other test case,
2. determined which methods were invoked by each of those tests, but not by any other test it is covered by,
3. mutated the methods according to some mutation strategy,
4. and, for each each mutation, executed the unit tests and all their covering unit tests and collected the results.

We used the following mutation strategies:

1. *full body deletion*, *i.e.,* we removed the complete method body.
2. code mutations of *JesTer*: JesTer, developed by Moore [Moore, 2001], is a mutation testing extension to test JUnit tests by finding code that is not covered by tests. JesTer makes some change to the code, runs the tests, and if the tests pass, JesTer reports what it changed. We applied the same mutations as JesTer, which are
   (a) change all occurrences of the number 0 to the number 1
   (b) flip true to false and vice versa

[2] http://homepages.ulb.ac.be/∼rowuyts/MagicKeys/index.html

(c) change the conditions of ifTrue statements to true and the conditions of ifFalse statements to false.

3.3.2 Results

The case studies are presented at more detail in Table 3.1 (p.36) and Table 3.2 (p.37).

System	LOC	LOC (Tests)	Coverage	#Unit Tests	Equivalent tests	Tests covered by Tests
Magic Keys	1683	224	37%	15	20%	53.3%
Van	3014	716	64%	67	9%	24.2%
CodeCrawler	4535	1071	24%	79	37.3%	40%
SmallWiki	5660	3096	64%	110	29.8%	47.4%

Table 3.1: The resulting coverage of unit tests in our case studies.

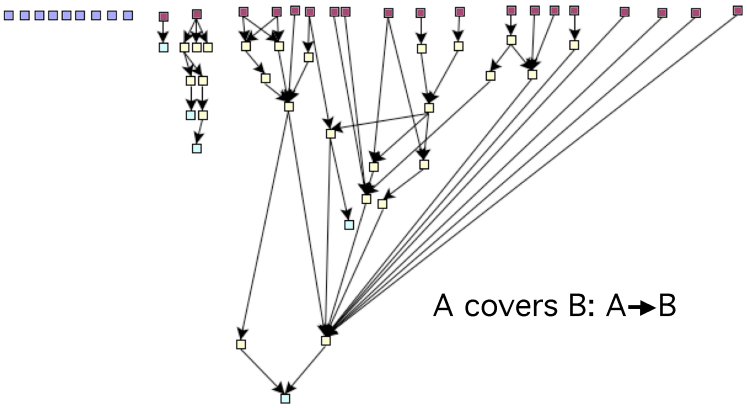

Figure 3.4: The coverage hierarchy of the Code Crawler tests visualized with Code Crawler.

As we see in Table 3.1 (p.36) our experiment was performed with applications which had 1600 to 5600 lines of code. The ratio of LOC(Tests) to LOC reached from 13% to 56%. The maximum test coverage was 64%.

In Figure 3.4 (p.36) an arrow from the top to bottom denotes that the *test node* at the top *covers* the *test node* at the bottom. 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 standalones (Figure 3.5 (p.37)).

A considerable percentage of unit tests (9% to 37%, see Table 3.1 (p.36)) called the same set of method signatures as at least one other test. 25% to 53% of the unit tests were

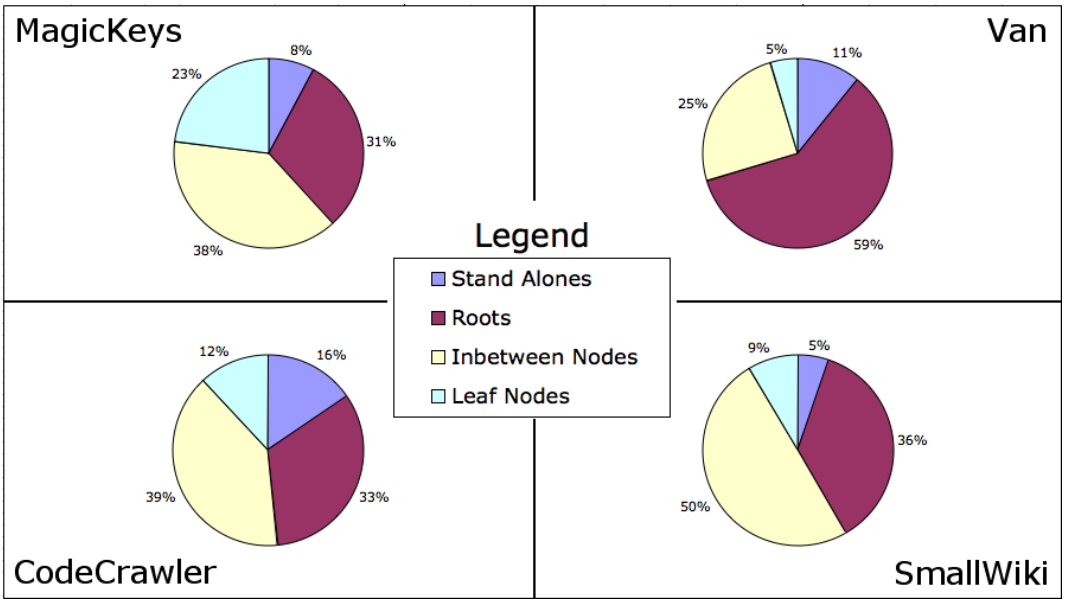

Figure 3.5: The distribution of comparable test nodes in our four case studies.

covered by at least one other unit test. This means that for roughly every third test of our case studies, the probability is high that if the test fails it will not fail alone.

System	#Methods	Strategy	#Methods mutated	Errors propagating to all covering tests
Magic Keys	277	Full Deletion	46	93.5%
		JesTer	17	58.8%
VAN		Full Deletion	357	97.8%
		JesTer	59	100%
CodeCrawler	1104	Full Deletion	41	92.7%
SmallWiki	1565	Full Deletion	2415	99.5%
		JesTer	318	100%

Table 3.2: Results of our automatic mutation experiments.

We carried out the second phase of our experiment, the automatic method mutation, in all case studies except CodeCrawler. As many mutations in CodeCrawler resulted in endless loops we did not have time to complete it. We merely did the full deletion mutation on every 10th method and omitted the JesTer mutations. The results are displayed in Table 3.2 (p.37): 92% to 99.5% of the full deletion mutations of a method broke the smallest test calling this method and all its covering tests, as did 59% to 100% of the JesTer mutations. Note that the number of mutated methods is larger than the number of methods, as the same method could be mutated in the context of different

tests.

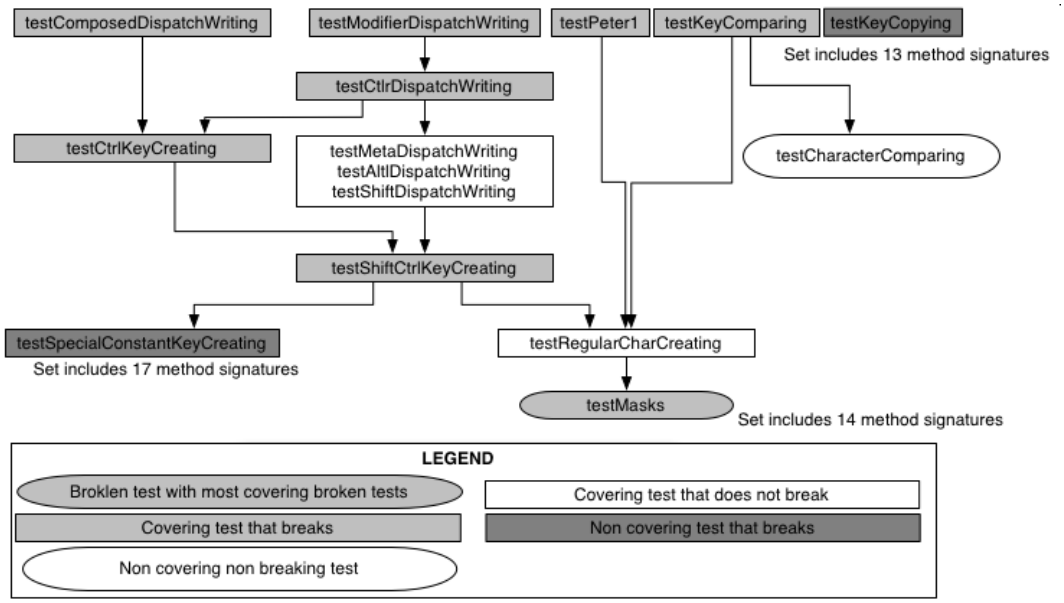

Figure 3.6: An avalanche effect in the coverage hierarchy of Magic Keys. One manually introduced bug causes 10 test cases to fail.

Let us have a detailed look at the effects of a full method deletion on the coverage hierarchy of the Magic Key tests in Figure 3.6 (p.38). We are mutating a method which is called from the test MagicKeysTest»testMasks, thus from all of its covering tests. Here we picked a rare example, where not all of the covering tests are failing. Both MagicKeysTest»testRegularCharCreating and the node including the equivalent test cases MagicKeysTest»testMetaDispatchWriting, testAltDispatchWriting and testShiftDispatchWriting do not fail because of the deleted method.

On the other hand the two tests MagicKeysTest»testSpecialConstantKeyCreating and MagicKeysTest»testKeyCopying also fail, though they do not cover the test MagicKeysTest»testMasks, they merely have a non-empty intersection set with it, including the mutated method. Also note, that MagicKeyTest»testKeyCopying, which is a standalone test, has the lowest number of method signatures called, and not MagicKeysTest»testMasks.

3.4 Discussion

The experiments we performed are rather simple, but they are also remarkable for the consistency of their results: In each case, a significant majority of the test cases was comparable to other unit tests, using the rather stringent criterion of inclusion of the sets of called methods. Furthermore, each case study consistently showed that if a defect causes a particular unit test to break, unit tests that precede it in the partial order also tend to break. The partial order over tests is therefore not accidental, but exposes implicit dependency relationships between the tests.

3.4.1 Semantic ordering of tests

In this chapter we focused on bug tracking via partial ordering of unit tests. Providing the order of unit tests could also help the developer to comprehend the structure of the unit tests and the structure of the underlying system. It can reassure the developer in his or her perceived layering of the system if the order of the test cases reflects this layering.

System	Signature of test case
Magic Keys	MagicKeysTest»testAltDispatchWriting
Magic Keys	MagicKeysTest»testMetaDispatchWriting
Magic Keys	MagicKeysTest»testShiftDispatchWriting
CodeCrawler	CCNodeTest»testRemovalOfEdgeRemovesChild
CodeCrawler	CCNodeTest»testRemovalOfEdgeRemovesParent
CodeCrawler	CCNodeTest»testRemovalOfSoleEdgeRemovesChildOrParent

Table 3.3: Examples for equivalent test cases.

The method names of the example in Table 3.3 (p.39) indicate a parallel structure of the tests, while the method names in the list below suggest a hierarchical one:

— LoaderTest»testConvertXMIToCDIF
(LoaderTest»testLoadXMI)

— SystemHistoryTest»testAddVersionNamedCollection
(SystemHistoryTest»testAddVersionNamed)

— SystemHistoryTest»
testSelectClassHistoriesWithLifeSpan
(SystemHistoryTest»testSelectClassHistories)

Another example of how partially ordering the tests can help developers to understand test suites and the system under test can be seen in Figure 3.7 (p.40) which displays the ordering relationship of a *Date and Time-package* for Squeak. The test TimeStampTest » testStoreOn covers the test TimeStampTest » testPrintOn which again covers the test TimeStampTest » testPrinting. Smalltalk developers are aware of the fact that store strings for some objects can be used to recreate these objects at a later time. Reading this hierarchy hints developers that creating store strings for time-stamps is reusing the functionality of printing time-stamps out to the user.

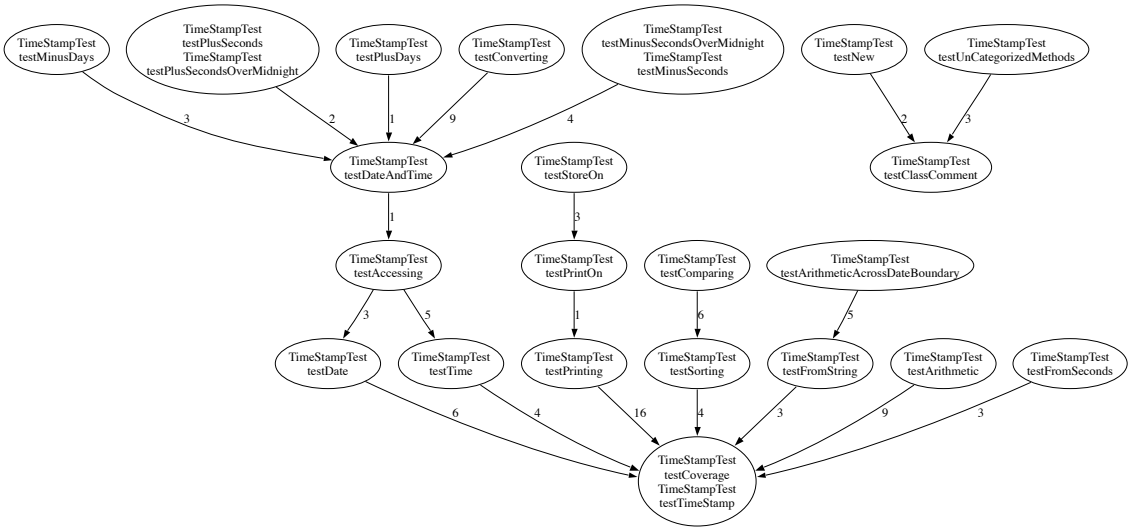

Figure 3.7: The generated partial order of a test suite concerned with a timestamp functionality in Squeak 3.7. The two equivalence classes of tests displayed in the two big ellipses on the top include similar method names indicating that they test similar behavior. Our overview also indicates, that storing and printing timestamps is similar if not interdependent behavior, like wise comparing and sorting timestamps.

3.4.2 Limitations

The lightweight nature of our approach has some drawbacks and limitations:

— One unexpected result was that if the JesTer mutations were applicable to some unit tests, in 100% of the cases a broken inner test case meant that all its covering tests were broken. Thus our first assumption that the more specific JesTer mutations would let more covering test cases survive, seems to be incorrect: The JesTer method

tweaks are even more fatal to the majority of covering tests than full body deletions. We plan to use more realistic mutations and manual introduction of errors in future experiments to overcome this problem.

— Parallel tests seem to cover each other even if they differ only by a single method signature. Sorting these basically equal unit tests does not add an advantage as an exception will probably cause all of them to fail and none of them will be more telling than the other.

— So far we have limited our case studies to Smalltalk programs. Perhaps style and conventions used in Smalltalk produce results which differ in other object-oriented languages.

— The developers of the case studies are all members of our research group thus also working in academia.

— We have not measured the implications of real bugs. How many unit tests break because of just one real bug and not because of one artificial mutation?

— We did not make any distinction between failures and errors when we were evaluating the chain of failed tests caused by one mutation.

3.5 Related Work

Memon *et al.* [Memon *et al.*, 2001] suggest the use of *planning* used in artificial intelligence in order to hierarchically sort GUI tests into partial orders.

Unit testing has become a major issue in software development during the last decade: Test-driven development (TDD) as defined by Beck [Beck, 2003] is a technique in which testing and development occur in parallel, thereby providing developers with constant feedback. The most popular unit testing framework used in TDD named XUnit and also developed by Beck [Beck and Gamma, 1998] does not currently prioritize failed unit tests.

De Pauw *et al.* [De Pauw *et al.*, 1998] suggest execution-patterns to understand and visualize traces of object-oriented programs. One pattern could be to flatten the execution trees, but they do not suggest to just put the signatures into sets and compare these sets only.

Chari *et al.* [Chari and Hevner, 2006] stress the notion of cyclically depending test cases and emphasize the schism between high level and low level tests: “we find a disconnect between test case generation approaches that deal at the low level of individual program statements and the reliability allocation models that are at the higher abstraction level

of software modules."

Parrish *et al.* [Parrish *et al.*, 2002] define a process for test-driven development that starts with fine-grained tests and proceeds to more coarse-grained tests. They state that *"Once a set of test cases is identified an attempt is made to order the test case runs in a way that maximizes early testing. This means that defects are potentially revealed in the context of as few methods as possible, making those defects easier to localize."* In their approach, tests are written beforehand with a particular order in mind, while in our approach we investigate *a posteriori* orderings of existing tests.

Rothermel *et al.* [Rothermel *et al.*, 2002] introduce the term "granularity" for software testing, but they focus on cost-effectiveness of test suites rather than on debugging processes.

Koschke's survey of software visualization has shown that graphs are the most popular means to convey information on software systems visually [Koschke, 2003].

Selective regression testing is concerned with determining an optimal set of tests to run after a software change is made [Rothermel and Harrold, 1996] [Bible *et al.*, 2001]. Although there are some similarities with the work described in this chapter, the emphasis is quite different: Instead of selecting which tests to run, we analyse the set of tests that have *failed*, and suggest which of these should be examined first.

Test case prioritization [Rothermel *et al.*, 1999] has been successfully used in the past to increase the likelihood that failures will occur early in test runs.The tests are prioritized using different criteria, the criterion which most closely matched our approach was *total function coverage* [Elbaum *et al.*, 2000]. Here a program is instrumented, and, for any test case, the number of functions in that program that were exercised by that test case is determined. The test cases are then prioritized according to the total number of functions they cover by sorting them in order of total function coverage achieved, starting with the highest.

End-users rely on the results of spreadsheets in day-to-day decision making but a study from reported errors in 38% to 77% of the spreadsheets examined. [Panko and Jr, 1996] They create cells and define formulas for them. These formulas use values contained in other cells for their calculations. But whereas spreadsheets are very concrete and thus attractive to the end-user, they lack a notion of testing. If the computations done in spreadsheets are acyclic, they can be put into a partial order, thus helping the spreadsheet developers to understand computations by visualizing them and also to debug them.

Wong *et al.* [Wong *et al.*, 1997] compare different selection strategies for regression testing and propose a hybrid approach to select a representative subset of tests combining modification based selection, minimization and prioritization. Again, they emphasize on

which tests should be run and not on how failing tests should be ordered. Modification based selection is their key to minimize the number of tests to run, thus they are relying on having prior versions of the tested program whereas our approach can in principle be used without having prior versions, as we could also order the tests using only the coverage of the failed tests.

Zeller and Cleve *et al.* [Zeller and Hildebrandt, 2002] [Cleve and Zeller, 2000] [Zeller, 2005] introduced delta debugging to simplify test case input, reducing relevant execution states and finding failure-inducing changes. We focus on reducing failing tests from a set of semantically different tests to the most concise but still failing tests. Thus the technique of Zeller *et al.* could pay off more using these smaller tests as initial input.

3.6 Summary

We have proposed a lightweight approach to partially order unit tests in terms of the sets of methods they invoke. Our experiments with four case studies reveal that this technique exposes implicit semantic ordering relationships between otherwise independent tests. We have showed that this ordering can be exploited in two ways:

Debugging failed Test Cases One fault in the software is triggering several test cases to fail, as test cases overlap with regards to the functionality they test. Our partial order enables developers to visualize that portion of tests which not only overlap but cover each other's functionality. It turned out that a high percentage of tests take part of such an ordering. As less specific unit tests tend to fail if more specific unit tests also fail, we can guide developers to the most specific failing test cases which are easier to understand and debug, as they provide the most specific context.

Understanding Test Suites We showed how exposing implicit semantic ordering relationships between test also helps developers to understand both the test suites and the system under test. As high-level functionality is using low-level functionality, high-level tests are using both low- and high-level functionality. The layering of the system is thus reflected by the layering of the tests, and visualizing the partial order helps to determine what high-level functionality is using which low-level functionality (Figure 3.7 (p.40)).

Our results imply that if more specific unit tests run, less specific unit tests are likely to run too. This makes a strong case for building high level unit tests, which is currently

difficult, as high level test scenarios cannot composed out of low level test scenarios. In Chapter 5 (p.67) we will show how the metaphor of examples allows developers to compose high level tests out of low level ones.

Chapter 4

Taxonomy of Unit Tests

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, and what to take into account while refactoring. But in our experience most unit tests focus on a single method under test. To validate this claim we have manually categorized the test base of a large existing object-oriented system in order to derive a taxonomy of unit tests which highlights the relationships of tests with their methods under test. 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 does focus on single methods. In this chapter we present our taxonomy, describe the results of our case studies, and present our approach to semi-automatic categorization.

4.1 Introduction

XUnit [Beck and Gamma, 1998] 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 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 *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 *Method command* could be displayed close to its method, and document a quality-approved usage of the method. (See Figure 4.1 (p.47)) 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 *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 *Method commands*. Thus *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.

[1] Further refactorings [Fowler *et al.*, 1999], 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.

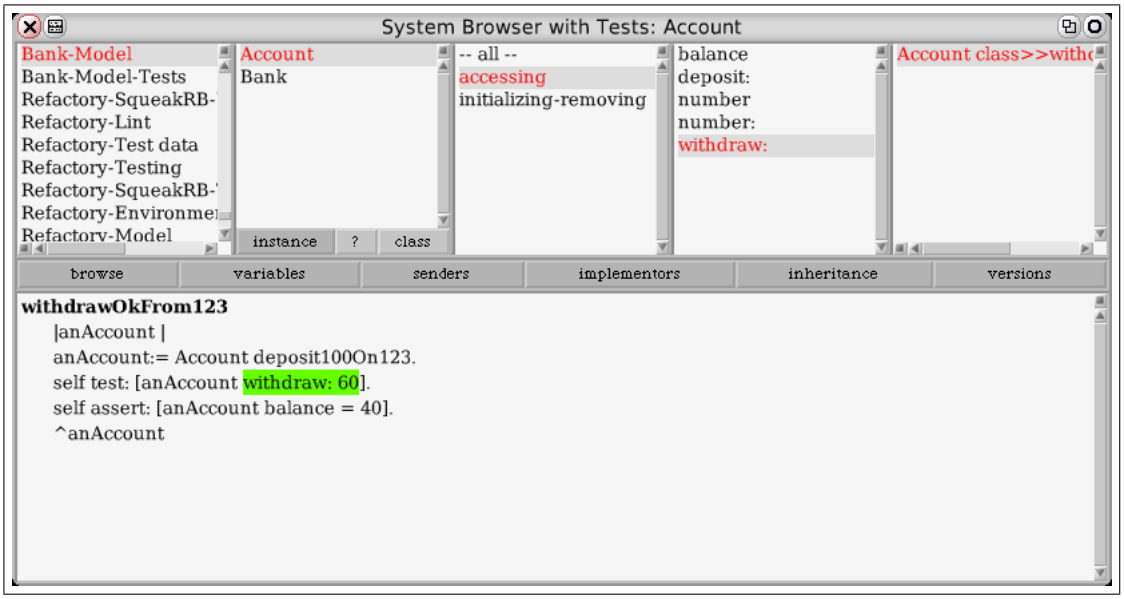

Figure 4.1: An enhanced class browser shows methods and their Method tests side by side. Note that the test returns its result, thus enabling other unit tests to reuse it. We thus store tests like other factory methods on the class side.

— 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 *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.

In Chapter 4.2 (p.48) we define some basic terms. In Chapter 4.3 (p.48) we present the taxonomy derived from our manual case study. In Chapter 4.4 (p.56) 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 Chapter 4.5 (p.60) we discuss some of the problems and difficulties encountered. Chapter 4.6 (p.63) briefly outlines related work. In Chapter 4.7 (p.65) we conclude and outline future work.

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

Abstract Command: Every XUnit Test is an *abstract command* [Gamma *et al.*, 1995], which is a parameter-free method whose receiver can be automatically created. The XUnit Test can thus be automatically executed.

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

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

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

4.3 A Taxonomy of Unit Tests

Initial case study. We derived the taxonomy by manually categorizing 982 unit tests of the Squeak [Ingalls *et al.*, 1997] base system[2]. Squeak is a feature-rich, open source implementation of the Smalltalk programming language written in itself and by many

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

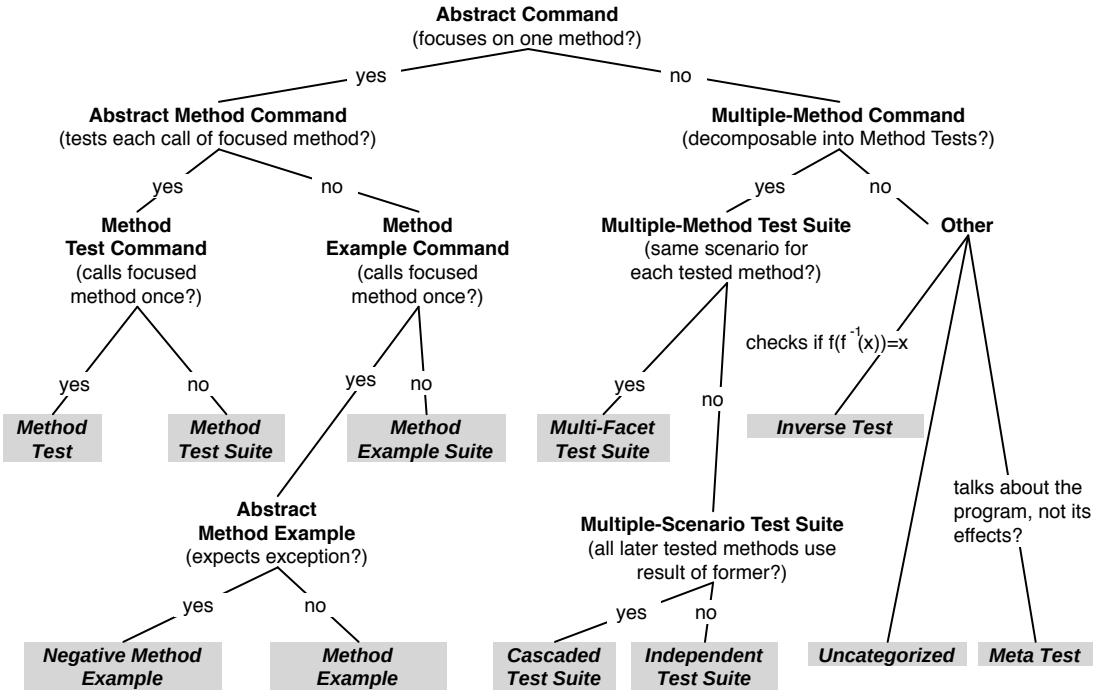

Figure 4.2: Taxonomy of unit tests. Nodes are gray and denote concrete occurrences of unit tests.

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 4.2 (p.49) by iteratively grouping tests into categories and refining the classification criteria. Our manual categorization yielded a distribution of the categories shown in Figure 4.3 (p.50).

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 Squeak unit tests[3].

We divide our taxonomy tree into two subtrees (Figure 4.2 (p.49)): (1) *Abstract method commands*, which are *abstract commands* that focus on single methods, and (2) *multiple-*

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

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.

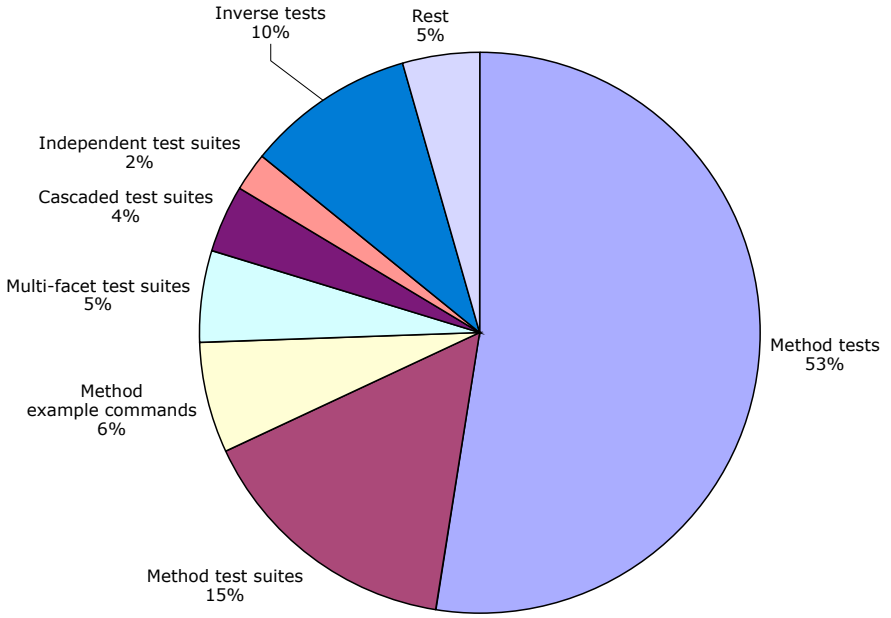

Figure 4.3: Manual classification of unit tests for the base Squeak system

4.3.1 Method test commands

A *Method test comand* is an *Abstract method command* which has assertions testing the outcome of *each call* of the method under test.

Method tests

If it tests the outcome of exactly one call of a method under test, we call it a *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.
```

Method test suites

On the other hand a *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.
```

4.3.2 Method example commands

A *Method example command* is an *Abstract 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:

Negative method example

A *negative method example* is an *abstract method example* which checks that an exception is thrown if a method is called in a way which violates a precondition. Beck [Beck, 2003] calls *negative method example "exception tests"*. Here is an example of a *negative 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 not test whether something is true or false in a given state, but merely check whether or not an exception is thrown.

Method examples

An *method example* is an *abstract method example* which expects that no exception is thrown if the method under test is called without violating some preconditions. Again, *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.
```

Method example suites

A *Method example suite* is a *Method example command* which calls the method under test more than once. It can be decomposed into several method tests or abstract method examples which call the same focused method once:

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

Above method example suite is decomposable into a negative method example and a method test.

4.3.3 Multiple-method test suite

A *multiple-method test suite* is a *multiple-method command* which is decomposable into Method tests. (See Figure 4.4 (p.53)).

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'.
```

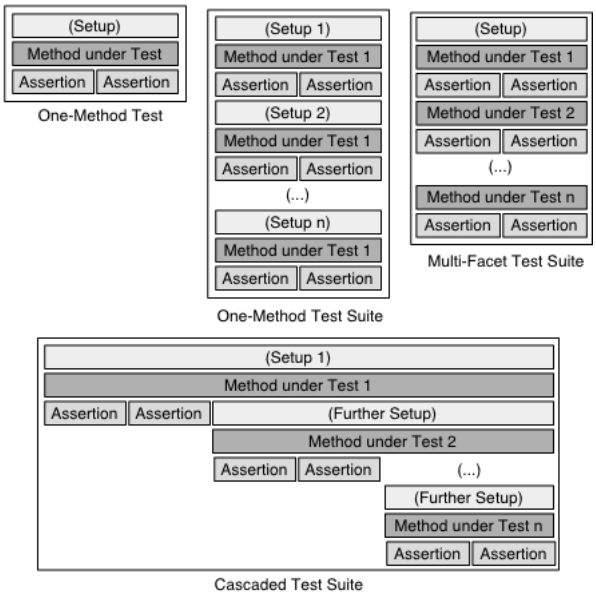

Figure 4.4: Method test suites, multi-facet test suites and cascaded test-suites are decomposable into Method tests.

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:.

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 ].


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) ].
```

4.3.4 Others

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

Inverse test

In [Gaelli *et al.*, 2005b] we called the *Inverse test Constraint test*.

An *inverse 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))
```

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

Meta test

A *meta test* is a test which talks about the implementation itself, *e.g.,* its structure, its current state, its implemented or unimplemented methods or its call graph. For example, the following squeak 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.
```

Mocks are used to state how often a certain method should be triggered in a given test. Freeman *et al.* [Freeman *et al.*, 2004] provide the following example for an access to a cache, where only the first time an object is loaded the lookup should be executed. This desired property is stated via mockLoader.expect(once()).

```
public void testCachedObjectsAreNotReloaded() {
 mockLoader.expect(once())
   .method("load").with( eq(KEY) )
   .will( returnValue(VALUE) );

 assertSame( "loaded object",
         VALUE, cache.lookup(KEY) );
 assertSame( "cached object",
         VALUE, cache.lookup(KEY) );  }
```

Ducasse *et al.* [Ducasse *et al.*, 2006] describe another approach to create *meta tests* using a logic language like Prolog for stating desired properties of coverage sets.

Uncategorized

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

4.3.5 First validation: Maven

We have detected out taxonomy using the squeak case study. To validate its generality we manually categorized 50 randomly selected JUnit tests of another case study using the Java project management and project comprehension tool called *Maven* [Massol and O'Brien, 2005].

25 of these tests merely checked some getter/setter code and were classified as inverse 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 *Method commands* as the majority of classified tests (See Figure 4.5 (p.56)).

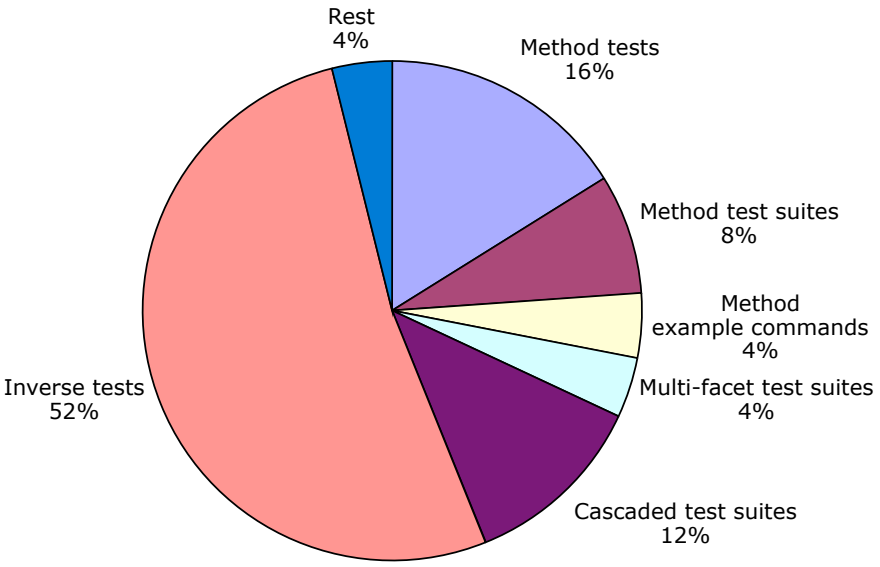

Figure 4.5: Manual classification of 50 random unit tests of Maven

4.4 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 4.2 (p.49). 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 Method commands tests, while our average precision rate was 89% (see Table 4.1 (p.58)). 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.

4.4.1 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 [Brant *et al.*, 1998], 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 and we lacked time to investigate this error further: As a consequence using the bytecode interpreter of Squeak did not work for exception handling code which is used mainly by method examples or negative method examples.

— Methods which only return a variable are inlined by the Smalltalk-compiler and thus cannot be detected[5].

4.4.2 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 4.2 (p.49). 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:.

   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.

[5] 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.

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. But due to a lack of time we could not further examine this interesting line of research.

To determine if a *Method command* is a *Method test command* or a *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 *Method tests* from *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 *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 *Method test* as *Method test command* which does not call any kind of should:raise: and shouldnt:raise:.

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

Table 4.1: Preliminary manual and automatic classifications of Method commands of the Squeak Unit Tests.

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

Table 4.2: Preliminary manual and automatic classifications of Method commands of the SmallWiki Unit Tests.

4.4.3 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 4.1 (p.58)). 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.

4.4.4 A Second Case Study: SmallWiki

After having done a manual categorization (see Figure 4.6 (p.60)) we automatically categorized the 200 unit tests of *SmallWiki* [Renggli, 2003], 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.

We only programmed the detection for three categories, namely *Method tests*, *Method test suites*, and *Method example suites*. All of them together represented already more than a third of all tests. Figure 4.6 (p.60) shows that contrary to the Squeak case study, the developers here wrote more *Method test suites* than *Method tests*. The recall and precision for *Method tests* displayed in Table 4.2 (p.59) 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.

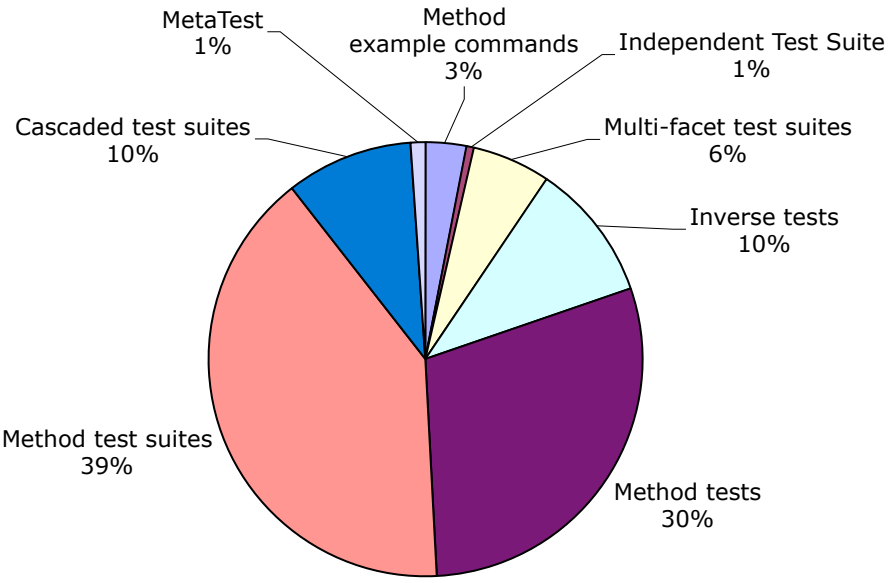

Figure 4.6: Manual classification of unit tests for the SmallWiki system

4.5 Discussion

82% of our manually classified Squeak tests implicitly focus on one method, which strongly supports our hypothesis. It shows clearly that we should give the developers means to make this link explicit[6] as we propose in [Gaelli *et al.*, 2004b].

A further advantage of our taxonomy is that it shows how tests are composable: 11% of our manually classified tests are implicitly composed of *Method commands*. We can thus give the developers means to compose and decompose tests explicitly. We will build upon this result for designing our meta-model in the following chapter.

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.* [Beck and Gamma, 1998] so

[6]For Smalltalk we suggest to use method annotations which would not add much burden to the developer and would be easy to parse

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 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 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 *inverse test*.

Test framework tests Tests of the test framework may be incorrectly categorized. The following test could be classified as a negative method example of error: but its intent is to be a 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 removeActionsWithReceiver: 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 [Mackinnon *et al.*, 2000] 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 a 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 *inverse test*.

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

Developers do not agree on the 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 *inverse tests*.

4.6 Related Work

Binder [Binder, 1999] 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 [Beck, 2003] argues that isolated tests would lead to easier debugging and to systems with high cohesion and loose coupling. *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.

Van Deursen *et al.* [Deursen *et al.*, 2001] talk explicitly about unit tests that focus on one method and start to categorize them.They also introduce bad test smells like *indirect testing*, which describe tests that we will categorize as *independent tests*. Bruntink *et al.* [Bruntink and van Deursen, 2004] 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 reuse the tests of its required classes to set up the complex test scenario, improves the testability of complex classes.

Eclipse [Holzner, 2004] 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 [Jézéquel, 1996] discusses how testing can rely on the *Design by Contract principle* [Meyer, 1992] 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 *Method commands*. The concept of *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 *Method examples* with assertion containing methods yields highly abstract *and* executable tests.

Van Geet researched the coevolution of software and tests over time [van Geet, 2006]. He used *Ant* as a case study to apply several metrics like counting methods per test or tests per method on several versions. He concludes that these heuristic metrics on dynamic test dependencies have to be combined with global code coverage information to provide a measure for overall coevolution of test code and product code. He also detects that lots of methods are executed over and over again in several test cases, whereas some methods are only executed in a few tests. He detects that these often executed methods belong to some setup code of general test scenarios.

Edwards [Edwards, 2004] 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.

4.7 Summary

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 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 *abstract commands* such as *method examples* or *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 following chapter we will exploit the fact that most unit tests focus on single methods by making this relationship explicit in a meta-model and discuss the several advantages of this explicit linking.

Chapter 5

An exemplified Meta-Model for Examples: Eg

As object-oriented developers we are used to solving a problem by introducing new objects and then describing their relationships to other objects. If we have a problem with software engineering itself, the same pattern can be applied. For example, If we find that behavior is not well factored out in current object-oriented programming languages, we introduce a new object "traits", [Schärli, 2005] or we introduce a new concept of "history" to describe the evolution of object-oriented programs ([Gîrba et al., 2005]). According to our observations made in Chapter 4 (p.45) we introduce the notion of explicit method commands *to connect methods and their exemplifying method commands. We call the meta-model based on those method commands "Eg" and describe in detail the responsibilities and collaborations of each class of our meta-model. Each responsibility is exemplified by a refactored version of our bank example, which we introduce in the beginning of the following chapter.*

We start this chapter by refactoring the example of the bank account as introduced in Chapter 2 (p.9) into explicitly linked tests serving as examples. We thus present how developers can associate tests with tests and also tests with their units under test in a seamless way in Smalltalk. We then abstract a meta-model out of this implementation centered around the concept of exemplifying method commands and describe this meta-model in detail. Then we validate our hypothesis that the single new concept of examples connecting agile concepts of tests and code is enough to solve the problems described in Chapter 3 (p.29). We finish the chapter by giving a brief outlook how this meta-model can be implemented in other object-oriented languages such as Java, Ruby and Python.

5.1 The Bank Account and its Tests refactored

As we value the principle of examples we are developing also our meta-model based on the bank example originally depicted in Figure 2.5 (p.16) in Chapter 2 (p.9). We therefore first refactor the bank account together with its tests using light-way conventions keeping the code readable but still transferable into a meta-model: Our conventions will allow developers to write tests and examples in a natural way, but will still make the missing links explicit.

As we have showed in Chapter 4.3 (p.48) most tests either focus on single methods or are decomposable into *positive method commands*. In our bank example we thus focus on the exemplification of classes and single methods. Looking back into the figure Figure 2.4 (p.15) in Chapter 2 (p.9) we can see that in the current xUnit framework two links are missing, namely the one between tests and other tests, and the one between tests and code, where code means classes and methods.

As you can see In Figure 5.1 (p.69) and Figure 5.2 (p.70) we have refactored the bank application together with its tests so that above mentioned links can be discovered in an unambiguous manner. We suggest two alternative strategies to make the meta-information persistent about what method the method command is focusing on and what kind of test it represents. In the first alternative we use code conventions using plain source code and a special parser, in the second alternative we suggest to use method properties.

5.1.1 Storing Eg Commands via Coding Conventions

We relied on this strategy for making the source code of the refactored bank tests (see Figure 5.2 (p.70) persistent. We did so by just using three conventions:

```
Object subclass: #Bank.Model.Bank
  instanceVariableNames: 'accounts'
  classVariableNames:''
  poolDictionaries:''
  category: 'Bank.Model'

Bank.Model.Bank>>initialize
  accounts := Dictionary new.
  "Postcondition"
  self assert: [self accounts isEmpty].
  ^self

Bank.Model.Bank>>accounts
  ^accounts values

Bank.Model.Bank>>accountNumbered: aNumber
 "Precondition"
  self includesAccount: aNumber.
  ^accounts at: aNumber
```

```
Kernel-Classes.Behavior(Bank class)>>new
  ^self basicNew initialize

Bank.Model.Bank>> createAccount: aNumber
   lanAccount l
  "Precondition"
  self deny: [(self includesAccount: aNumber)].
  anAccount := Account numbered: aNumber.
  accounts
    at: aNumber
    put: anAccount.
  "Postcondition"
  self assert: [self includesAccount: aNumber].
  self assert: [
    (self accountNumbered: aNumber) number =
    aNumber].
  ^self

Bank.Model.Bank>>includesAccount: aNumber
  ^accounts includesKey: aNumber
```

```
Object subclass: #Bank.Model.Account
  instanceVariableNames: 'balance number'
  classVariableNames:  ''
  poolDictionaries:  ''
  category: 'Bank.Model'

Bank.Model.Account class>>numbered: aNumber
  lanAccount l
  anAccount := self new initializeFor: aNumber.
  "Postcondition"
  self assert: [anAccount balance = 0].
  ^anAccount

Bank.Model.Account>>balance
   ^balance

Bank.Model.Account>>withdraw: someAmount
  loldBalance l
  oldBalance := balance.
  "Precondition"
  self assert: [self canWithdraw: someAmount].
  balance := balance - someAmount.
  "Postcondition"
  self assert:
    [oldBalance = (balance + someAmount)]
```

```
Kernel-Classes.Behavior(Account class)>>new
  ^self basicNew initialize

Bank.Model.Account>>initializeFor: aNumber
  number := aNumber.
  balance:= 0

Bank.Model.Account>>number
  ^number

Bank.Model.Account>>deposit: someAmount
   loldBalance l
  oldBalance := balance.
  balance := balance + someAmount.
  "Postcondition"
  self assert:[
     oldBalance = (balance - someAmount)]

Bank.Model.Account>>canWithdraw: someAmount
  ^balance >= someAmount
```

Figure 5.1: The Canonical Bank Account in Smalltalk now with Pre- and Postconditions.

Checked Method Example
(Bank class>>createAccount:)

```
Bank.Eg.Bank class>>withAccount1234
  ^self new
     createAccount: 1234;
     yourself
```

Method Example
(Bank>>accountNumbered:)

```
Bank.Eg.Account class  >>empty1234
  ^Bank withAccount1234 accountNumbered: 1234
```

Checked Method Example
(Account>>withdraw:)

```
Bank.Eg.Account class>>testWithdraw
  ^self testDeposit withdraw: 30
```

Inverse Test
(Account>>deposit: and Account>>withdraw:)

```
Bank.Eg.Account class>>testDepositAndWithdrawAll
  lanAccount anAmount oldBalancel
  anAccount := self empty1234.
  oldBalance:= anAccount balance.
  anAmount:=100.
  anAccount
    deposit: anAmount;
    withdraw: anAmount.
  self assert: [anAccount balance = oldBalance]
```

Negative Method Example
(Bank class>>createAccount:)

```
Bank.Eg.Bank class>>testAccount1234Exists
  self
    should: [self withAccount1234 createAccount: 1234]
    raise: Exception
```

Checked Method Example
(Account>>deposit:)

```
Bank.Eg.Account class>>testDeposit
  ^self empty1234 deposit: 100
```

Negative Method Example
(Account>>withdraw:)

```
Bank.Eg.Account class>>testWithdrawTooMuch
  self
    should: [self testWithdraw withdraw: 130]
    raise: Exception
```

Figure 5.2: The bank account tests refactored to our new meta-model. We managed to abstract all assertions into pre- and postconditions in the code. We implemented our meta-model in a light way: We say, that the *last* method called is the method under test. By *returning* the result we can compose the examples. As a consequence only two tests have to be called to gain the full coverage.

First convention: Tests as Factories. The tests are moved to the class sides of Bank and Account respectively. They all return according instances of these classes and thus serve as *tidy* examples of these classes. This works because tests never should leave any unclean side-effects. If there were any side-effects we would ensure [1], that these side-effects would be cleaned by allowing developers to inline any eventual TestCase » tearDown functionality into the test itself.

Note that we also allow collections of instances to be returned by positive method commands. If we *e.g.,* had a method to retrieve accounts from the bank Bank.Model class » accounts we would allow its according method test to return a collection of accounts but it still would get stored into Bank.Eg.Bank class.

Second convention: Last call of the test before assertions calls the method under test. We call the method under test always at last with two exceptions: We allow special keywords such as *yourself* to be called later (see Bank.Eg.Bank class » withAccount1234 in Figure 5.2 (p.70)), and second, in the case of method tests, we allow a list of assertions to be called later. If we need to denote a different method than the last one called, we can either use method annotations or, in the case of Smalltalk, we can also bracket the method under test with a block a la self test:(aBar foo: someParameter), so that Bar » foo is declared as the method under test. For doing so, we only have to implement Object » test: aBlock in our Eg framework and change the possibly existing SUnit framework in such a way, that this method codeInlineObject » test: aBlock is not discovered by accident to be a classic SUnit test, as it also starts with "test".

Third convention: Storage of the tests in a dedicated module. In XUnit the test methods are denoted with a naming convention: All test method names have to start with "test". In JUnit 4 this convention has been changed so that tests can also be denoted with method annotations. We chose a different convention by making clear that *all* methods which are stored in a certain module (in our example the module named "Eg.Bank") serve as examples and tests. As a consequence *no* helper methods, which are not commands, can be declared by the developers. Helper methods in tests make the tests difficult to understand and can often be refactored into the domain themselves.

Using these three conventions above is enough to construct the meta-model depicted in Figure 5.5 (p.74). The only difficult part is to retrospectively discover inverse tests. But closely looking at Bank.Eg.Account class » testDepositAndWithdraw we discover the scheme depicted in Figure 5.3 (p.72).

[1] In Smalltalk one can use a construct like BlockContext » ensure: anotherBlock. There anotherBlock gets executed in any case, no matter whether the execution of aBlock fails or not.

```
Receiver class>>testFunction1Function2
 |aReceiver oldAttribute aParameter |
  aReceiver := self createReceiver.
 oldAttribute := aReceiver getInvariantAttribute.
 aReceiver
  function1: aParameter;
  function2: aParameter.
 self assert: [aReceiver getInvariantAttribute = oldAttribute].
  ^aReceiver
```

Figure 5.3: Inverse Tests are made explicit, by storing them in this schema.

5.1.2 Storing Eg Commands via Method Properties

To avoid inherent and invisible code conventions in the source code, one can also use method properties[2]. A method storing a method test by enhancing it with a method property in VisualWorks Smalltalk would look like the following:

```
Account class » exampleDeposit
   < egClass: #'Eg.MethodTest' method: #deposit: >
    | aReceiver aResult |
    aReceiver := Account new.
    aResult := aReceiver deposit: 100.
    self assert: aResult balance = 100.
    ^aResult
```

The method property can be seen in the second line of above code. It states that this test is a method test focusing on the method Account » deposit. Note that the receiver class (Account) of the method under test can be detected by partially running the method command so that the temporary variable "aReceiver" on which the method under test is executed gets filled with an instance of the receiver class. Using this convention allows the developer to directly type in method commands as they do not have to be afraid to violate some internal and hidden code conventions which would be the case if they use our first strategy introduced above. But as our meta-model described below allows good tool support for testing, typing in method commands directly should become more and more superfluous.

After having refactored the bank tests from standalone tests as seen in Figure 2.5 (p.16) into *composed* tests as seen in Figure 5.1 (p.69) we applied our partial order mechanism from Chapter 3 (p.29) to these two sets and got the two orderings depicted in Figure 5.4 (p.73). We did not change the *semantics* of our tests as we refactored them, as a consequence the partial order of them stays the same and we can use partial orders to test if our refactorings indeed preserved the semantics.

[2]Method properties are also called Pragmas in VisualWorks Smalltalk. Since version 3.9 they are also part of the Squeak Smalltalk Environment and have been introduced to Java in SDK Version 1.5

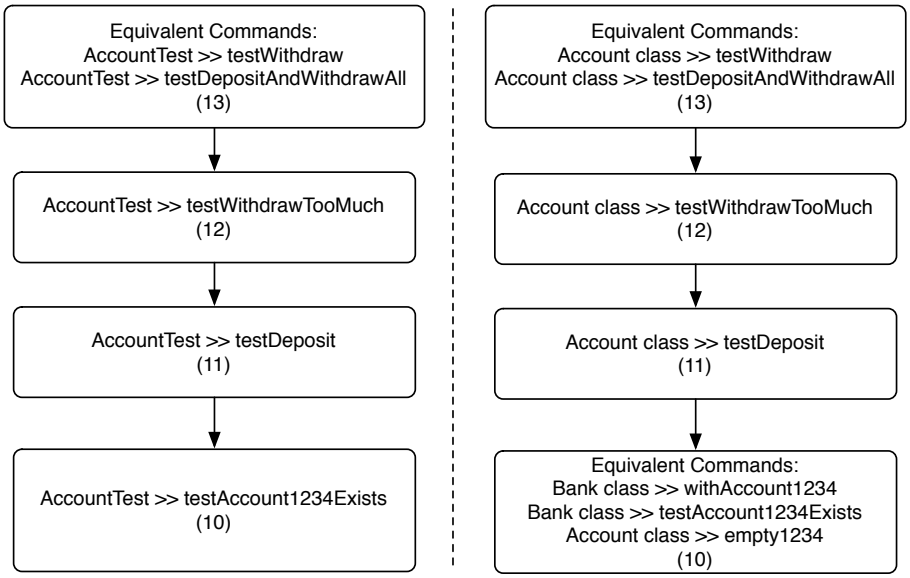

Figure 5.4: The generated partial order of the old tests (left) and the refactored tests (right) of the bank account. We only added the two tests on the bottom right making the setup explicit. The structure stays the same as we refactored the tests keeping their semantics. The partial order thus serves as a good hint how tests can be recomposed.

5.2 Exemplified Responsibilities of Eg

In Figure 5.5 (p.74) [3] you can see the hierarchy of the various explicit commands we made explicit after having detected their real world use in the taxonomy chapter Chapter 4.3 (p.48). *95% of the tests written in the Squeak case study (Figure 4.3 (p.50)) can be either directly translated or decomposed into these explicit commands.* In the following we explain the responsibilities and collaborators of each class defining our meta-model in detail following the object design approach of Wirfs-Brock *et al.* [Wirfs-Brock and McKean, 2003] extended with examples both for the classes we introduce and for the responsibilities we demand.

[3] We define that each direction of an association is navigable if it is labeled. A getter method exists for all navigable associations.

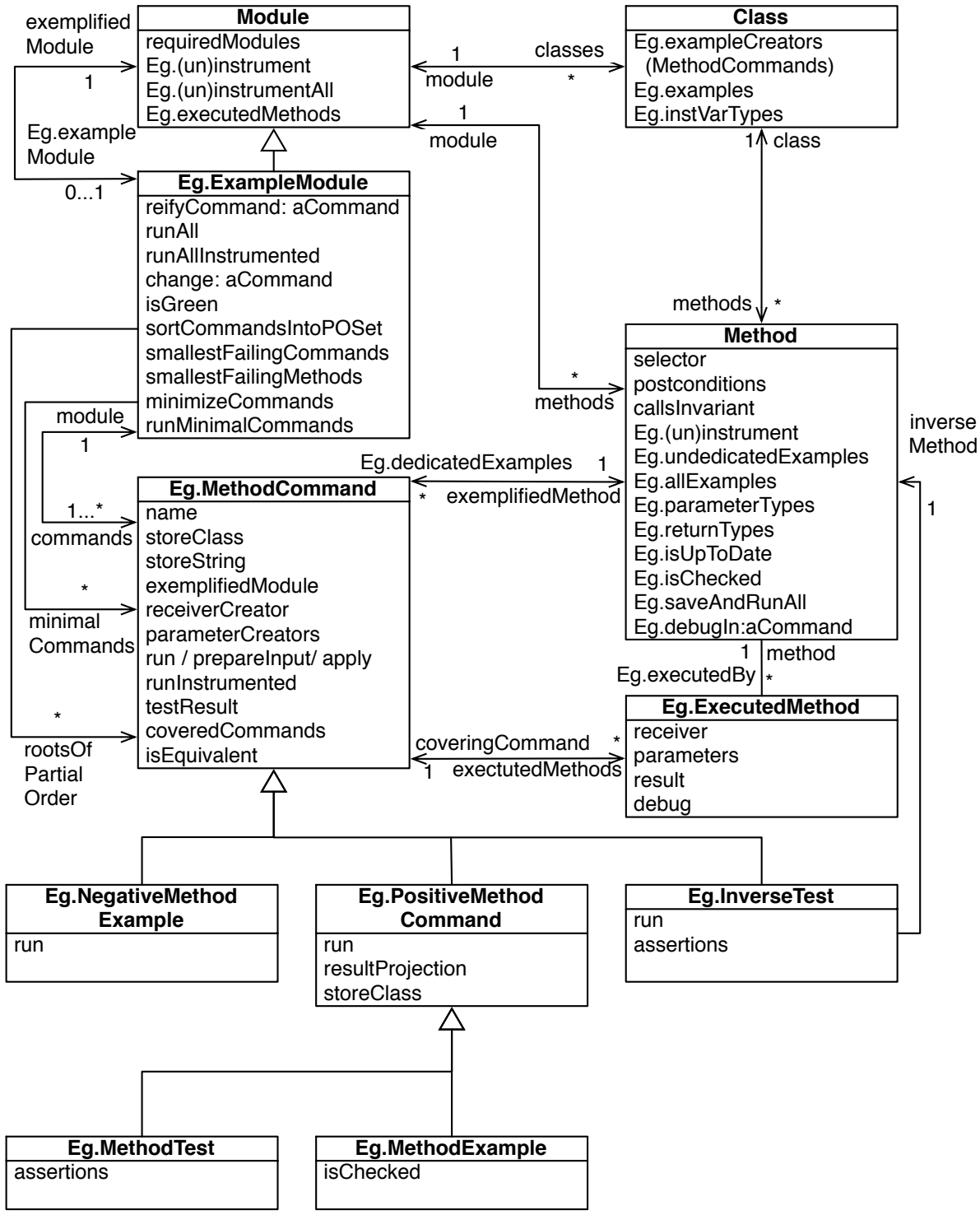

Figure 5.5: The hierarchy of commands in our meta-model. An exemplified method can collect all its exemplifying commands and display them to the developer.

5.2.1 Module

We define modules to hold a set of classes and methods. Only some languages support this point of view. In Java for example developers cannot define methods for existing classes in new modules whereas Ruby or Smalltalk allow developers to do so. Classboxes as defined by Bergel [Bergel, 2005] are a way to solve this problem for Java and other languages that do not support class extensions.

Responsibilities.

1. **Module » requiredModules** A module knows all its required modules. We define required modules to be the set of modules which are created by end-developers and not the set of modules which are already provided by the infrastructure of the underlying programming environment. *Universes* as described by Spoon [Spoon, 2006] provide this point of view: the context (a universe) is taken to be given so that a simple name of a required module is enough to load it: the newest version of this module is considered to be the best one and thus does not need to be specified.

   Scope. This functionality is expected to be provided by the module, where the class Module is defined making it a functionality of the kernel of our system.

   Example. In our bank example the required modules for Bank.Eg would contain the modules Bank.Model but not the module Eg, as first of all Eg would be a module of the underlying infrastructure, and second none of the commands defined within Bank.Eg need any functionality of the Eg-framework itself. It is the special module Eg which defines that functionality and not its contents. The advantage of keeping the examples agnostic about the Eg framework is that they can be deployed together with the code for documentation and testing issues without the framework itself – the example artifacts stay independent of the framework and the framework can be licensed freely or commercially. [4].

2. **Module » methods** A module can enumerate all the methods defined within it.

   Scope. This functionality is expected to be provided by the module, where Module is defined making it a functionality of the kernel of our system.

   Example. Printing out the source of all enumerated methods defined in Bank.Model would yield the source of all methods displayed in Figure 5.1 (p.69).

3. **Module » classes** A module can enumerate all its classes.

   Scope. This functionality is expected to be provided by the module, where Module is defined in and thus to be a kernel functionality.

[4] This example could be expressed as a method test when writing tests and examples to verify Eg.

Example. Printing out the source of all enumerated classes defined in Bank.Model would yield the source of all classes displayed in Figure 5.1 (p.69) [5].

4. **Module » exampleModule** A module knows its exemplifying module.

   Scope. This is a functionality of the Eg framework.

   Example. The example module of Bank.Model is the module called Bank.Eg.

5. **Eg.Module » (un)instrument** A module can be instrumented, so that the call of one of its method gets logged in the set of its executedMethods. All methods of the module can be wrapped by a piece of code, which registers in the module that its method was executed as soon as its method gets triggered. This functionality relies on some available wrapping technique to instrument methods with general code. For a discussion of wrapping techniques for Smalltalk see Ducasse [Ducasse, 1999]. For a modern and fast implementation using bytecode transformation see Denker *et al.* [Denker *et al.*, 2006]. Uninstrumenting the module means to bring all the methods into their original state, so that no extra code will be executed when the methods get executed.

   Scope. This is a functionality of the Eg framework whereas the used basic functionality of code wrapping is expected to be in the module, where Module is defined.

   Example. Instrumenting the module Bank.Model and then calling the command Bank.Eg.Bank class » withAccount1234 should result in a set of executed methods (ExecutedMethod) all knowing their concrete receiver, parameter and return values, and point to the methods depicted in Figure **??** (p.**??**).

Bank.Model.Bank class >> new	Bank.Model.Bank >> initialize
Bank.Model.Account class >> new	Bank.Model.Bank >> accountNumbered:
Bank.Model.Bank >> createAccount:	Bank.Model.Bank >> includesAccount:
Bank.Model.Account class >> numbered:	Bank.Model.Account >> initializeFor:
Bank.Model.Account >> balance	Bank.Model.Account >> number

Figure 5.6: The set of all methods of the Bank module directly or indirectly called from the checked method example Bank.Eg.Bank class withAccount1234.

   On the other hand un-instrumenting the module and running above command should result in an empty set of executed methods[6].

6. **Eg.Module » executedMethods** A module can be asked for all its methods which have been executed in a certain situation. It therefore iterates over all its methods

[5] This and above example could be expressed as a method tests when writing tests and examples to verify Eg. In this case a collection of methods and classes respectively would be delivered by the according method tests.

[6] This example could be expressed as a method test when writing tests and examples to verify Eg.

(Module » methods) and collects all of their executed methods (Eg.Method » executedBy:.

Scope. This functionality is part of the Eg framework.

Example. The previous example includes a call to Eg.Module » executedMethods.

7. **Module » (un)instrumentAll** A module can be instrumented together with all its required modules. This functionality simply enumerates all required modules and instruments them together with the module itself. Instrumentations can be undone for all required modules together with itself (Module » uninstrumentAll).

   Scope. This functionality is a functionality of the Eg framework.

   Example. Instrumenting Bank.Eg together with all its required modules and then running all commands of Bank.Eg would result in a combined list of all methods described in Figure 5.1 (p.69) and Figure 5.2 (p.70) as the commands – giving a 100% coverage of the module Bank.Model – would be instrumented too[7].

Collaborators.

1. A module knows all its methods and classes which are defined in it.
2. A module can enumerate all its required modules to work in.
3. A module can have one example module (Module » exampleModule).

5.2.2 Example Module

An example module (Eg.ExampleModule) is a module that holds all commands testing its exemplified module. It is responsible for running these tests, for running them in an instrumented way so that all their touched methods get logged into them and for sorting those instrumented commands into a partial order.

Scope. All further functionality of this class is part of the Eg framework.

Example. An example for an ExampleModule would be the module Bank.Eg.

Responsibilities.

1. **ExampleModule » reifyCommand: aCommand** All methods stored in an example module play the roles of genotypes of commands: An example module can reify all

[7] This example could be expressed as a method test when writing tests and examples to verify Eg.

commands from the source code of its methods into living objects (ExampleModule » commands) and thus create and hold according instances of Method Command.

Example. Reifying the source code from Bank.Eg.Bank class » withAccount1234 in Figure 5.2 (p.70) should yield an instance of a checked method example, whose storeString (Eg.Method Command » storeString) in turn gives the original source[8].

2. **ExampleModule » runAll** An example module can run all of its commands. This is the main interface for emulating the classic triggering of all unit tests. This meta-model does not provide any kind of test-suites as introduced by XUnit, but it could be easily extended to do so.

   Example. Running all commands in the module Bank.Eg the first time should result in a set of seven commands where the test result Method Command » testResult of each of these commands has passed.

3. **ExampleModule » runAllInstrumented** An example module can run all of its commands *instrumented*. It does so by calling Method Command » runInstrumented to *each* of its commands. As a consequence the underlying exemplified model gets instrumented and un-instrumented for the run of each command. For each instrumented method being executed a Eg.ExecutedMethod is created and stored both in the set of executed methods of the method itself (Method » executedBy and in the set of executed methods of the method command which directly or indirectly triggered the method (Eg.MethodCommand »

   Example. Running all commands in the module Bank.Eg instrumented the first time results in a set of seven commands where the test result Method Command » testResult of each of these commands has passed and the set of executed methods Method Command » executedMethods is not empty for each command.

4. **ExampleModule » isGreen** An example module is green if all of its commands have passed and all of the methods of its exemplified model are up to date (Method » isUpToDate). As a consequence the example module should run all commands instrumented as soon as one of its commands has changed.

   Example. Running all commands in the example module Bank.Eg instrumented the first time should result in a green example module.

5. **ExampleModule » change: aCommand** This functionality is a place-holder for adding/ removing or changing a command of the example module which is not described in detail here. But each time a command of the example module has been changed, all of its commands are run instrumented afterwards.

[8] This example could be expressed as an inverse test when writing tests and examples to verify Eg.

Example. As this functionality is a place-holder for adding, removing or deleting commands, no concrete examples are given.

6. **ExampleModule » sortCommandsIntoPOSet** An example module can sort all of its commands into a partial order. As a prerequisite for doing so, it is necessary that the example module be green as all of its commands have passed. The goal of sorting the commands into a partial order as explained in Chapter 3 (p.29) is on the one hand to ease the debugging process. On the other hand graphs of the partial order can then be displayed to the developers to help them understand the test suites at hand. This functionality first copies all commands of the module ExampleModule » commands into a collection called ExampleModule » rootsOfPartialOrder. If the algorithm detects a command being covered by another command it removes it from this collection and stores it into the set of covered commands Method Command » coveredCommands of the covering command. In a second phase the algorithm removes cyclic references by unifying all commands having the same set of executed methods Method Command » executedMethods: If several commands are equivalent with respect to these sets of executed methods, a new command node is created by fully copying one of them. In a next step this new command acts as a parent for the other ones by first removing them out of their former holders, putting them into its set of covered commands and setting its boolean Method Command » isEquivalent to true.

   Example. Sorting the commands of Bank.Model yields a partial order according to Figure 5.4 (p.73) [9].

7. **ExampleModule » smallestFailingCommands** Having the partial order of a green example module in place using ExampleModule » sortCommandsIntoPOSet allows the developer to ask the example module for its smallest failing commands. This question can be asked after some code of the underlying exemplified module has been changed and after the all the commands have been run. The set of the smallest failing commands is defined as follows: A command is the smallest and failing, if the command is failing and its set of covered commands Method Command » coveredCommands does not include any other failing command.

   Example. As a prerequisite of this example Bank.Model is green and sorted. Then running the commands of Bank.Model with a bug introduced in Account.Model » deposit: should yield a set of smallest failing commands containing only one failing command, namely the checked method example Account.Eg » testDeposit.

8. **ExampleModule » smallestFailingMethods** Knowing the set of smallest and failing commands allows *Eg* to detect the set of smallest failing methods: A method

[9] This example could be expressed as a method test yielding the example module of Bank.Eg containing sorted commands as a return value when writing tests and examples to verify Eg.

is called to be smallest and failing, if the commands which include the method in their set of executed methods Method Command » executedMethods are smallest and failing commands and if the method is not included in the set of executed methods of any of their covering non failing commands.

Example. Running the example described for ExampleModule » smallestFailingCommands yields a set of smallest failing methods containing only once instance, namely Account » deposit.

9. **Eg.ExampleModule » minimizeCommands** An example module can minimize the set of commands to be run, if commands call each other in order to create test scenarios via Method Command » receiverCreator or Method Command » parameterCreators. This way commands can be called only as often as necessary: A command, that is called by some other command, does not need to be executed in a standalone manner. A collection of directed acyclic graphs of commands calling other commands is stored in the sets of minimal commands (ExampleModule » minimalCommands).

   Example. The resulting set of minimal commands of our refactored bank example consists of three commands: Account class » testDepositAndWithdrawAll, Account class » testAccount1234Exists and Account class » testWithdrawTooMuch. All other commands are reused by these commands for creating their scenarios and thus do not have to be called explicitly (see Figure 5.8 (p.90)).

10. **Eg.ExampleModule » runMinimalCommands** An example module can run all its tests in a minimal way so that only the top nodes of the partial order created by ExampleModule » sortCommands get executed. Note that this functionality does not sort the commands before running the minimal set, so that the partial order of a set of *running* commands is preserved.

    Example. Only the three commands described in the previous example are run.

Collaborators.

1. An example module knows its exemplified module.
2. An example module stores all its reified commands.

5.2.3 Method Command

A method command (Method Command) does not need any further context to be executed. Whereas in the XUnit framework tests are stored on the instance side of a

paralleled test hierarchy we store method commands as unary methods on the class side of our domain classes. We can do this, as commands neither need a receiver nor any parameters for creating their context and only *internally* call other commands to set up the context of their method under test.

A method command of the *Eg* framework focuses on a single method. It knows how to recreate the context in which its *exemplified method* can be executed.

Scope. All functionality of this class is part of the Eg framework.

Example. Method Commands are abstract, only its subclasses have direct instances: Bank.Eg.Bank class » withAccount1234 would be an example for a *persisted* method command, namely a checked method example.

Responsibilities.

1. **Method Command » name** Method Commands have a telling name. This name is unique within the class method names of its store class and thus can be used as the method name for persisting the command. It hence should contain only characters allowed within method names.

   Example. In our bank example the name of the command (a checked method example) Bank.Eg.Bank class » withAccount1234 would be withAccount1234.

2. **Method Command » storeClass** All method commands know the class in which their source code is stored. By default they get stored in the class of the instance, which is the result of evaluating their receiver creator (Method Command » receiverCreator). In the case of method commands (MethodExample) this behavior is overwritten to return the class of the result they return.

   Example. The *negative method example* Bank.Eg.Bank class » testAccount1234Exists is stored into Bank.Eg.Bank class as the receiver creator self withAccount1234 returns an instance of the bank class Bank.Eg.Bank class.

3. **Method Command » storeString** Method Commands can store themselves as readable source code using classical source code repositories like cvs, subversion, or in the case of Smalltalk fileouts, Monticello (Squeak), Envy (VisualWorks and IBM Visual Age) or Store (VisualWorks).

   Example. Reifying the source code from Bank.Eg.Bank class » withAccount1234 in Figure 5.2 (p.70) should yield an instance of a checked method example, whose store string (Eg.Method Command » storeString) in turn gives the original source[10].

[10] This example could be expressed as an inverse test when writing tests and examples to verify Eg.

4. **Method Command » module** Method Commands know the module they are stored in.

   Example. All commands exemplifying the bank application can tell that they are stored in Bank.Eg.

5. **Method Command » exemplifiedModule** Method Commands can be asked for their exemplified module. For detecting this, they just have to ask their module for its exemplified module.

   Example. In our example the exemplified module of each command would be Bank.Model.

6. **Method Command » receiverCreator and Method Command » parameterCreators** Method Commands know how to *create* the context necessary to execute their exemplified method. The genotype of the receiver and the parameters of their exemplified method is stored in a receiver and parameter creators. These creators can either be *method commands* which would return an object of interest, or blocks (lambda expressions), literals as string, integers, symbols and the like. To treat literals and commands the same way one has to teach literals a run method to be polymorphic with method commands. This run method in literals just returns the literal itself.

   > Doing encapsulation right is a commitment not just to abstraction of state, but to eliminate state oriented metaphors from programming. – Alan Kay [Kay, 1993]

   In an early prototype of *Eg* we stored the state of the receiver and parameters in an XML format, but realized that it was cumbersome to compose these test objects into higher level scenarios. We had to side track the way the application was designed to compose higher-level objects and instead found ourselves using accessors all the time.

   Example. In the checked method example Bank.Eg.Account class » testDeposit the receiver creator code self empty1234 gets unified with the method example defined Bank.Eg.Account class » empty1234 and stored in the receiver creator of Bank.Eg.Account class » testDeposit. The "100" gets stored as the first and only element of the parameter creators of Bank.Eg.Account class » testDeposit.

7. **Method Command » run** Method Commands can be executed directly. This functionality is abstract as negative method examples have to wrap the run with an exception handler, inverse tests even have to apply two methods and method commands return a result at the end. The class Method Command itself can provide functionality to prepare the input by evaluating the receiver- and parameter

creators Method Command » prepareInput and it can apply the exemplified method to this input via Method Command » apply.

8. **Method Command » runInstrumented** Method Commands can be executed with their exemplified module together with required modules instrumented before using the following scheme:

   (a) They call its exemplified module (Method Command » exampleModule) to be instrumented together with all its required modules (Module » instrumentAll),

   (b) They run themselves (Method Command » run),

   (c) They collect the executed methods in their appropriate field (Method Command » exectutedMethods),

   (d) They finally uninstrument its exemplified module together with all its required modules (Module » uninstrumentAll).

   Example. Running the command described in Bank.Eg.Bank class » withAccount1234 instrumented, and then asking this command for its covered methods Method Command » coveredMethods should result in the same set of methods as described in the example for Eg.Module » (un)instrument.

9. **Method Command » testResult** Method Commands know if something went wrong after they have been executed. The test result is either a failure (some assertion has failed) or an error (some unexpected exception has been thrown) together with the according error message, or the test result indicates that the command has been executed without an error.

   Example. The test result of all commands of our bank example indicate that there have been no errors during their execution.

10. **Method Command » exemplifiedMethod** Method Commands can display their exemplified method. This exemplified method can be derived *statically*.

    Example. The exemplified method of Bank.Eg.Account class » testWithdraw would be the method (Method) Bank.Model.Account » withdraw.

11. **Method Command » executedMethods** Method Commands store a list of all methods they executed.

    Example. The set of executed methods (Eg.ExecutedMethod) of the checked method example Bank.Eg.Bank class » withAccount1234 would consist of the ten executed methods as explained in the list depicted in the examples for Eg.Module » (un)instrument in Figure **??** (p.**??**).

Collaborators.

1. Method Commands collaborate with one or more exemplified methods (ExemplifiedMethod). Being abstract, the cardinality and specific use is determined by the subclasses of ExemplifiedMethod.
2. Method Commands directly know the module in which they are stored, and thus indirectly also know their exemplified module.

5.2.4 Negative method example

> I am not completely useless. I could always serve as a bad example – Mark Twain

A *negative method example* is a method command that shows an invalid call of a certain method. As its method under test is supposed to fail, its method under test should not have any (side) effects. Therefore a negative method example does not return an object of interest and thus cannot be reused by other commands.

Scope. All functionality of this class is part of the Eg framework.

Example. Bank.Eg.Bank class » testAccount1234Exists would be an example for a negative method example.

Responsibilities. A negative method example inherits all responsibilities of *method commands*. It only has to implement the abstract functionality of *running* itself.

1. **Eg.NegativeMethodExample » run** A negative method example is run by wrapping the call of its method under test by an exception handling mechanism. The test result is set to green if the wrapped piece of code throws the expected exception after executing it within the context set up by the negative method example.

   Example. Running the negative method example described in Bank.Eg.Bank class » testAccount1234Exists triggers an *expected exception* as accounts of a certain name can only be created once. The test result is set to green as the method under test fails as expected.

5.2.5 Positive Method Command

A *positive method command* is a command which delivers a result. It is stored in some *storeClass* which is derived by asking its result for its class. A positive method command

is abstract but provides a common functionality for its two subclasses *Method Test* and *Method Example*, namely to run itself.

Scope. All functionality of this class is part of the Eg framework.

Example. All examples of its subclasses *Method Test* and *Method Example* serve as examples of *Positive Method Commands*.

Responsibilities.

1. **Eg.PositiveMethodCommand » resultProjection** Running a positive method command yields a result. This result is either the receiver, one of the parameters or the result of the method under test. To determine which of these objects shall be returned by the method command, the meta-model needs a slot for storing this information. A possible value for this slot is one of the following list of symbols: (receiver, parameter1, ... parameterN, result).

   Example. Running the checked method example described in Bank.Eg.Account class » testWithdraw returns the result of the method under test, which in this case is also the receiver. Thus the field resultProjection would include the symbol result.

2. **Eg. PositiveMethodCommand » run** As this method could not have been defined in the superclass command as its behavior differs for all its subclasses, we have to define it here. Running a positive method command consists of several steps:

   (a) Preparing input via Method Command »prepareInput

   (b) Applying the method under test to this prepared input via Method Command »apply

   (c) Returning either a (changed) receiver or parameter of the prepared input or the result of the application above, depending on the symbol in the resultProjection-field.

   Example. Running the checked method example described in Bank.Eg.Account class » testWithdraw would consist of the following steps:

   (a) Preparing the input: Run the checked method example Account class » testDeposit

   (b) Applying the method under test to the input: Apply the method under test Account » withdraw: to the non empty account created in the first step together with the literal of 30 and storing the resulting account to some temporary result variable.

(c) Determining and returning a result: The field resultProjection includes the symbol result, thus the returned result would be the account containing 70 as its balance.

5.2.6 Method Test

A *method test* is a positive method command which contains some assertions in the test itself.

Scope. All functionality of this class is part of the Eg framework.

Example.

```
Bank.Eg.Account class≫testWithdrawAll
  | anAccount |
  anAccount := self testDeposit.
  anAccount withdraw: 100.
  self assert: (anAccount balance = 0).
  ^anAccount
```

would be a method test as it includes assertions and returns a value.

Responsibilities.

1. **Eg.MethodTest » assertions** A method test knows all the assertions stored into it.

   Example. The set of assertions of Bank.Eg.Account class » testWithdrawAll consists of only one code block namely anAccount balance = 0.

5.2.7 Method Example

A *method example* is a positive method command which does not contain any assertions in itself. It is called to be checked, if its method under test contains a postcondition.

Scope. All functionality of this class is part of the Eg framework.

Example. Bank.Eg.Account class » empty1234 is a method example as it does not include any assertions but returns a value.

Responsibilities.

1. **Eg.MethodExample » isChecked** A method example knows if it is checked as it can ask its exemplified method if it has post-conditions.

   Example. Bank.Eg.Account class » testWithdraw is a checked method example as its exemplified method Account » withdraw contains a post-condition.

5.2.8 Class

A *class* contains variable definitions and method definitions. We require that a class knows the module where it is defined (Class » module), and that it can enumerate all its methods it defines (Class » methods).

Within the context of our meta-model *Eg* we extend classes so that they can provide examples for their instances, that they can enumerate all the method commands which provide these examples, and that they can display all concrete types of their instance variables which are deducible by their examples.

Scope. All functionalities of classes explained as follows are part of the Eg framework.

Example. Bank.Eg.Account class is a class with extended capabilities defined by *Eg*.

Responsibilities.

1. **Class » Eg.examplesCreators** If *Eg* is used, a class can enumerate all method commands which return exemplifying instances of itself. It does so by filtering out all method commands out of the example module of its module which return instances of itself.

   Example. Bank.Eg.Account class » Eg.examplesCreators collects all commands of the module Eg.Bank which is the example module of Bank where the class Bank is defined in. It returns all method commands depicted in Figure 5.2 (p.70).

2. **Class » Eg.examples** If *Eg* is used, classes can enumerate all their examples by executing all their example creators.

   Example. Bank.Eg.Account class » examples would execute all its example creators and return their results. The set of examples returned consists of five instances of accounts, as the class has five method commands returning instances of itself. Two of these accounts would be empty, one would contain a balance of 30, one a balance of 100, and one a balance of 70.

3. **Class » Eg.instVarTypes** Knowing its example instances, it is easy for a class to detect typical values and thus concrete types of its instance-variables. As a prerequisite its instances need to be able to return the values of their instance-variables which can be accomplished by exploiting reflection capabilities of the base system. In Smalltalk one could use Object » instVarAt:.

   Example. Bank.Eg.Account class » Eg.instVarTypes would return a dictionary with one key, balance, and one value, SmallInteger.

5.2.9 Method

A *method* is the atomic unit of object-oriented programs. As such it is defined within the base-system of the underlying system.

Within the context of our meta-model *Eg* we extend methods in several ways as described below. We require that methods can tell their selector (Method » selector), the class and module in which it is defined in (Method » class), (Method » module), their post-conditions (Method » postconditions), and whether they call an invariant or not (Method » invariant).

Scope. All functionalities of methods explained as follows are part of the Eg framework.

Example. All methods denoted in fig:bankaccountSUnitTestsRefactored and Figure 5.1 (p.69) serve as examples as they are reified within the system.

Responsibilities.

1. **Method » Eg.(un)instrument** Each method can be instrumented and un-instrumented. We suggest to use a default instrumentation strategy but make other strategies pluggable within the system. We require from the instrumentation strategy that it can read out the actual receiver, parameters *and* result of the method after it has been executed.

   When a method is instrumented and executed, it creates an executed method with its actual receiver- parameter and return-values, links to it and adds that executed method to the list of executed methods of the currently run command – if there is any[11].

[11] The Example Module has a helper variable "actualCommand" which is set accordingly while running over all its commands in an instrumented way.

Example. Running the method Bank.Model.Bank » accountNumbered: instrumented by running the checked method example Bank.Eg.Bank class » withAccount1234 instrumented, creates one instance of ExecutedMethod pointing to Bank.Model.Bank » accountNumbered:. Whereas a method can be executed lots of times by a single command, in our small example the method Bank.Model.Bank » accountNumbered: is called just once. You can find an example of this executed method in Figure 5.6 (p.88).

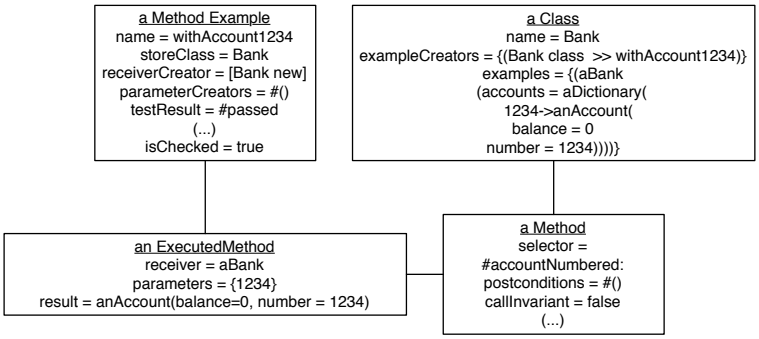

Figure 5.7: An object diagram depicting the relationship of a checked method example, an executed method and a method testing the bank application.

2. **Method » Eg.executedBy:** Each method can display a the set of executed methods which call it.

   Example. Asking the method Account » deposit: for its set of executed methods yields a set of executed methods containing four instances whose covering commands are either one of the following:

   — Bank.Eg.Account » testDeposit

   — Bank.Eg.Account » testWithdraw

   — Bank.Eg.Account » testDepositAndWithdrawAll

   — Bank.Eg.Account » testWithdrawTooMuch

3. **Method » Eg.dedicatedExamples** Each method can display a the set of dedicated method commands which focus on it.

   Example. Asking the method Account » deposit: for its dedicated examples yields a set of method commands which focus on this method containing only one instance, namely Bank.Eg.Account » testDeposit.

4. **Method » Eg.undedicatedExamples** Each method can display a the set of undedicated method commands which only cover it somewhere during their execution but do not focus on it.

   Example. Asking the method Account » deposit: yields the following set of method commands which only call this method indirectly:

   — Bank.Eg.Account » testWithdraw

   — Bank.Eg.Account » testDepositAndWithdrawAll

   — Bank.Eg.Account » testWithdrawTooMuch

5. **Method » Eg.allExamples** Each method can display a combined set of method commands which either focus on it as an exemplified method or only cover it somewhere during their execution.

   Example. Asking the method Account » deposit: for all its examples yields the following set of method commands which either focus on this method directly or indirectly:

   — Bank.Eg.Account » testDeposit

   — Bank.Eg.Account » testWithdraw

   — Bank.Eg.Account » testDepositAndWithdrawAll

   — Bank.Eg.Account » testWithdrawTooMuch

6. **Method » Eg.parameterTypes** Each method can display the concrete types of its parameters if its set of executed methods (Eg.ExecutedMethod) is non empty.

   Example. Asking the method Account » deposit: for all its concrete parameter types yields a set containing only one class, namely SmallInteger.

7. **Method » Eg.returnTypes** Each method can display the concrete types of its returned object if its set of executed methods (Eg.ExecutedMethod) is non empty.

   Example. Asking the method Account » deposit: for all its concrete return types yields a set containing only one class, namely Account, as it implicitly returns self.

8. **Method » Eg.isUpToDate** Each method knows if it "up to date", meaning if all its examples (Method » allExamples) have been executed after its latest change.

   Example. All tests as depicted in Figure 5.2 (p.70) are up to date with the code depicted in Figure 5.1 (p.69). Changing either method of the bank would make it outdated.

9. **Method » Eg.isChecked** Each method, whose set of post conditions is not empty or which calls an invariant, *and* which has at least one covering command, is called to be "checked". Furthermore we require that the covering command is not a negative method example focusing on this method as this would only check if the precondition of the method fails in a certain circumstance.

   Example. All methods depicted in Figure 5.1 (p.69) which come with a post-condition are checked, as all of them have at least one method command which is executing them.

10. **Method » Eg.saveAndRunAll** Each method not only can save itself, but also can save itself and run all its examples (Method » Eg.allExamples) just afterwards –making it up to date again (Method » isUpToDate).

    Example. Saving the changed method Account » deposit: and running all its examples would trigger the following method commands:

    — Bank.Eg.Account » testDeposit

    — Bank.Eg.Account » testWithdraw

    — Bank.Eg.Account » testDepositAndWithdrawAll

    — Bank.Eg.Account » testWithdrawTooMuch

    Method » Eg.debugIn: aCommand Each method can be debugged within one of its exemplifying commands (Method » Eg.allExamples). A breakpoint is inserted into the method and the appropriate command gets executed, so that the method can be seen in the debugger within a running context.

    Example. Debugging the method Account » deposit: becomes easy by running its dedicated example Bank.Eg.Account class » testDeposit.

5.2.10 Eg.ExecutedMethod

An *executed method* is a persisted message send retrieved by instrumenting the method.

Scope. All functionalities of executed methods explained as follows are part of the Eg framework.

Example. An executed method for the method Eg.Bank » accountNumbered: is depicted in Figure 5.6 (p.88).

Responsibilities. An executed method can tell the concrete receiver, parameters and result of the message send, knows its covering command, and the method it instantiates.

11. **Eg.ExecutedMethod » receiver/ parameters/ result** Being a message send an executed method not only knows its concrete receiver and parameter objects but also can tell its result.

    Example. Values for the receiver, parameters and result of an executed method are depicted in Figure 5.6 (p.88).

5.3 Validation

In this section we describe how the meta-model solves the problems of implicit test interdependencies as described in Chapter 2 (p.9).

5.3.1 Creating Test Scenarios is easy

Objects to create scenarios can be easily found for reusing them as they are bound as factory methods to their classes. Using *Eg* developers can explicitly ask a class for all its *tidy* examples. *Eg* just collects all positive method commands of that class and presents them to the developer. Creating a scenario thus boils down to filling the receiver and parameters of new methods under test by either selecting existing method commands or by creating new literals by just typing them in. As an advantage of our approach compared to xUnit, scenarios are not bound in the setup and can be used by other tests: In our example application of the bank account depicted in Figure 5.2 (p.70) we "freed" the setup code into the methods Bank.Eg.Bank class » withAccount1234 and Bank.Eg.Account class » empty1234 respectively.

5.3.2 Our sorting techniques help to understand the interplay of the system

We have presented two sorting techniques of tests whose created order reflects the interplay of the system under test. The first sorting technique we have presented in Chapter 3 (p.29) is to partially order the sets of commands of a given example module via ExampleModule » sortCommandsIntoPOSet. The second sorting technique we have presented is to minimize a set of method commands who call each other in order to reuse created test

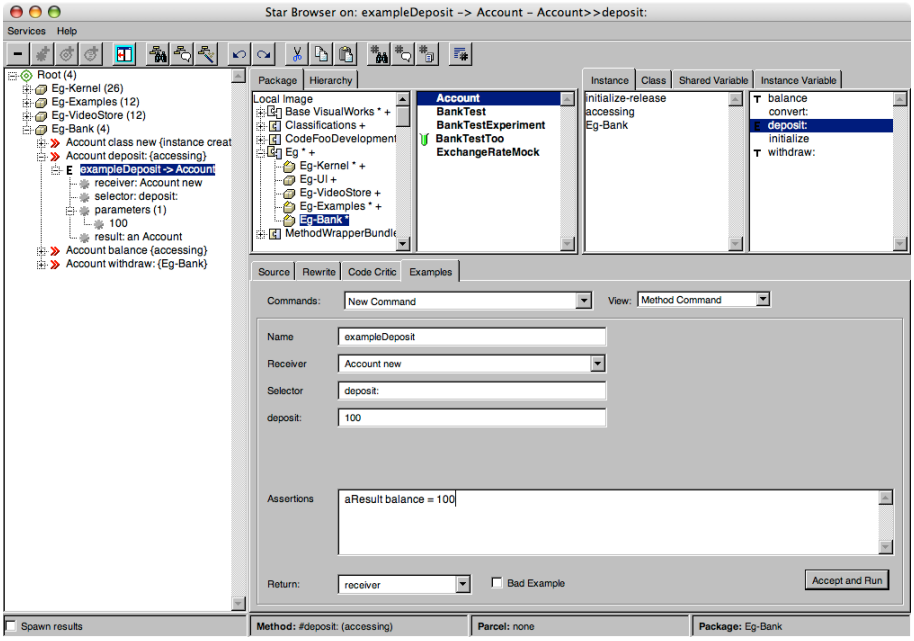

Figure 5.8: Creating an example for depositing money with Eg, the tool. More complex objects than numbers represented by method commands can be dragged and dropped out of the left pane into the parameters fields.

scenarios. (ExampleModule » minimizeCommands). This sorting directly mirrors the structure of the system under test with the tests themselves. In Figure 5.4 (p.73) you can see the partial order of our bank account example before and after refactoring it, already implying the layering of the system and the according tests, whereas the sorting of the refactored tests according to their reuse mirrors directly the dependency-layering of the system under test (Figure 5.8 (p.90)).

5.3.3 Minimizing Testing Time

We have shown how the testing time can be minimized by first allowing the developers to reuse created test scenarios via method commands and then minimizing these sets of method commands in order to only call the high level commands which in turn call the rest of the commands (ExampleModule » minimizeCommands.

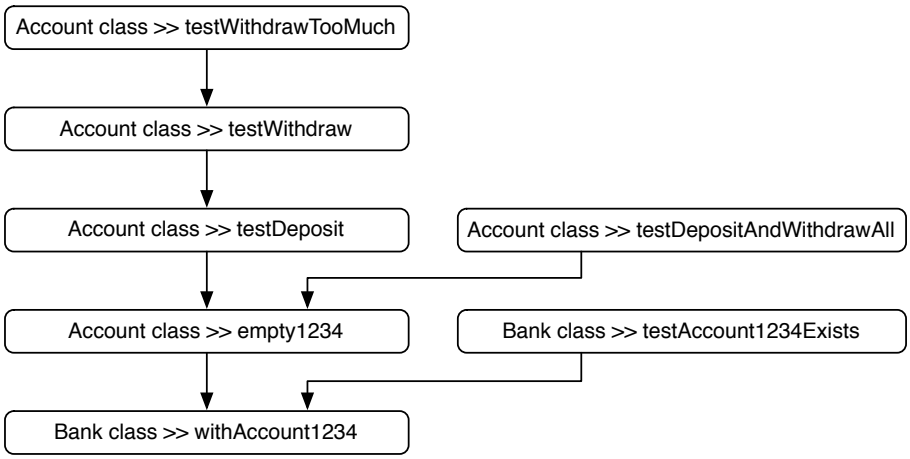

Figure 5.9: The sorted refactored bank tests according to what test reuses what other tests. This structure mirrors the structure of the system under test: A bank has to be created before an account can be added to the bank, some money has to be deposited on the account, before it can be withdrawn.

5.3.4 Identifying relevant failed tests in the case of a failure is easy

Having a partial order of tests of a running green test suite stored in an example module allows developers to spot both the set of smallest failing commands *and* the set of smallest failing methods (ExampleModule » smallestFailingCommands and ExampleModule » smallestFailingMethods).

Example. Consider the partial order of unit tests of the Squeak-package "Aconagua" which deals with converting scalar units in Figure 5.9 (p.91). All tests are running there. We planted an error in ArithmeticObjectInterval » reverseDo and reran the tests *uninstrumented without* sorting them again. Updating the color of the tests in the graph according to the state of the tests but keeping the topology of the running partial order graph results in Figure 5.10 (p.92): The two failing tests ArithmeticObjectIntervalTest » testReverseDo and ArithmeticObjectIntervalTest » testReverse are colored red now. Our meta-model allows us to present the developers both the smallest failing command, which in this case would be ArithmeticObjectIntervalTest » testReverseDo. But it even allows us to guide the developer directly to the failing method itself (ArithmeticObjectInterval » reverseDo) and opening a browser on the failed method.

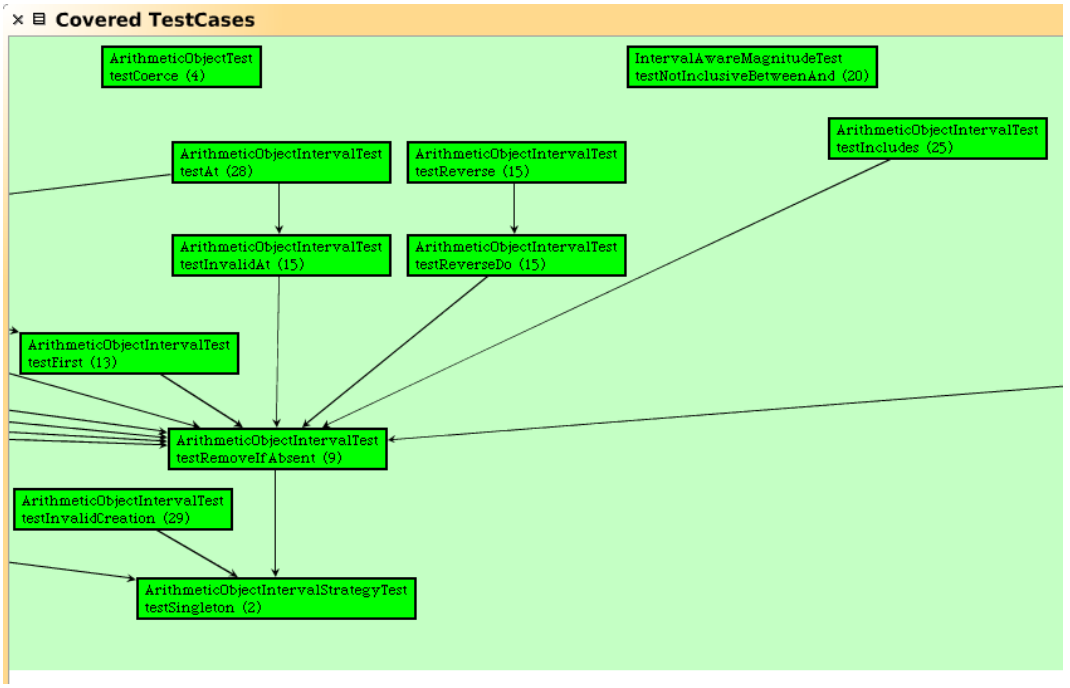

Figure 5.10: One can see the partial order of tests for the Squeak package Aconagua. All tests are green.

5.3.5 We can detect similar tests

Our partial order also allows us to detect similar tests. Tests which cover the same set of executed methods are stored in equivalence classes represented by commands which are tagged as equivalent and contain the set of equivalent commands in their covered commands (Method Command » coveredCommands).

5.3.6 We know exactly the scope and the kind of a test

When we make our commands persistent, we store also its kind in a method property. As a consequence we can reify commands into method examples, negative method examples, inverse tests etc. Thus we can always display the type of the test to the developer. Also the scope of all tests is made clear, as they in general only focus on one method, with the exception of inverse tests focusing on two methods at the same time.

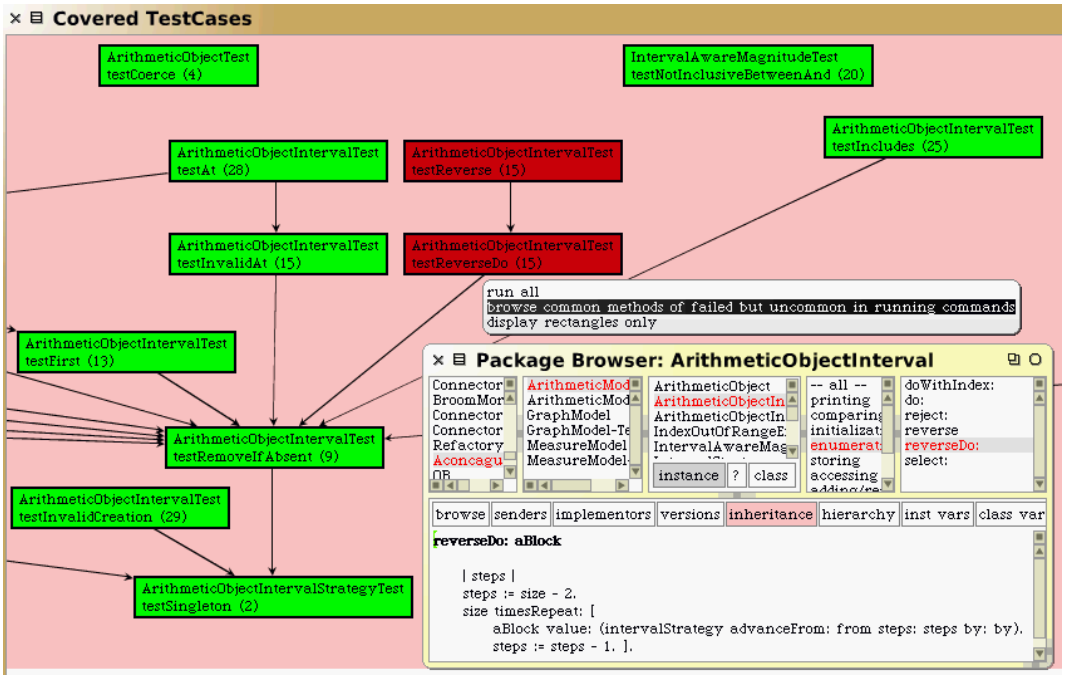

Figure 5.11: We planted an error in reverseDo:, reran the tests but display them using the original order of running tests.

5.3.7 We can highlight the best examples for methods

Each command is focusing on exactly one method Method Command » exemplifiedMethod and thus serves as one of the "best examples" for it. The methods know their dedicated examples (Method » dedicatedExamples) and undedicated examples (Method » undedicatedExamples). Hence browsers can display the dedicated examples for a method next to the method in a five pane browser as suggested in Figure 4.1 (p.47) in Chapter 4 (p.45) and tag them as best examples with a special color. Method Commands which only include a certain method only in the set of executed but not as an exemplified method are the undedicated examples and can be displayed next to the method, just colored differently.

5.3.8 We can separate tidy from untidy examples

Only tidy example creators should be stored as method commands into example modules. Though we provide this context for storing tidy examples we cannot enforce developers to follow this convention but by a rigorous tool approach. But having blurred the difference between tests and examples allows developers to use module examples as their main test suites, and thus they would soon clean up all commands which would create any left-overs.

5.3.9 We can synchronize tests with code with a minimal overhead

Each method knows if it is up to date, by checking if all tests which are concerned with it have been executed after its latest change. If there are no commands testing it, the method is not up to date, but untested. Using our meta-model can easily implement a function which would not only save a method but also exercise all commands concerned with that method. (Method » undedicatedExamples)

5.3.10 All exemplified methods can be seen in a debugger

Developers just have to select a command of the set of all examples (Method » allExamples) and *Eg* can guide them directly into a context where this method is executed. It does so by simply inserting a temporal breakpoint into the method and exercising the command.

5.3.11 Typing

All methods exercised by any command(Method Command » allExamples know the concrete types of their parameters and their returned object (Figure 5.11 (p.94)). All classes exemplified by example creators can be asked for the concrete types of their instance variables.

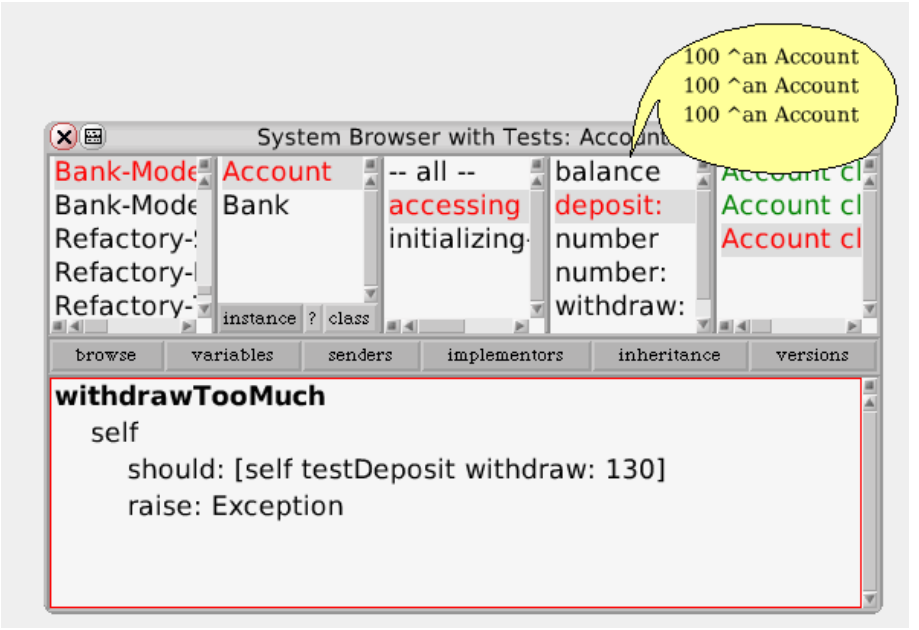

Figure 5.12: A prototype of a five pane Eg browser displays the concrete receivers and parameters of the exemplified method after hovering over the method signature using a tooltip. In this case the method deposit has been called via three tests, all of them feeding the account with an amount of 100. The method deposit: always returns an account. Inferring the concrete types just means to ask the concrete values for its classes. The test we display here includes depost: as an executed method, but serves as a dedicated example (as a negative method example) for withdrawing too much from an account.

5.4 Converting existing tests into Eg-Tests: A Small Case Study

We converted 40 Unit Tests from Mondrian, a visualization framework developed by Meyer *et al.* [**?**], into the new meta-model in a pair-wise programming scenario.

5.4.1 Results

We used the heuristic that the method called last before calling any assertions was the method under test. All tests could be decomposed into positive method commands, as we neither encountered any negative method examples, nor any inverse tests or

multiple method commands. Many of the tests only checked some internal behavior of the framework which had a high distance from the method under test. As a consequence we only succeeded in refactoring three tests into checked method examples which are our preferred tests as they give rise to the most reusability of tests. We managed to refactor 34 of the remaining 37 of the tests into method tests. We had problems to refactor three tests as they asserted some state on a temporary variable which was neither a receiver, parameter or result of our candidate methods under test. We would categorize those tests as "independent tests" according to our taxonomy introduce in Chapter 4 (p.45).

5.4.2 Discussion

The most interesting result of this case study was our inability to refactor around 10% of the unit tests according to our meta-model as we could not understand them. Our ignorance about Mondrian forbid us to refactor the code under test together with the unit tests to make them conforming with our meta-model. XUnit allows developers to write all kinds of tests – and they will make use of this facility until a more rigid testing methodology like the one we introduced with *Eg* suggests developers to make the links between tests and code explicit – and as soon as developers understand the benefits of these explicit links.

Chapter 6

Conclusions

In the introduction to this dissertation (Chapter 4.1 (p.46)), we claimed the following:

— Tests are currently neither composable nor explicitly bound to the concepts of current object-oriented languages.

— Detecting above implicit links and making them explicit using the guiding metaphor of examples helps developers to document, type, debug and test object-oriented programs.

In this final chapter, we first summarize how the contributions presented in this thesis support this statement (Chapter 6.1 (p.97)).

Then, we we discuss directions for future work (Chapter 6.3 (p.99)) and conclude by presenting the lessons we have learned (Chapter **??** (p.**??**)).

6.1 Contributions

In Chapter 2 (p.9) of this dissertation, we have given a detailed description of problems that arise with missing explicit links between tests and code in object oriented programming languages. These problems include difficulties in test creation, program understanding and debugging, test synchronization, test time reduction, and duplicated typing.

To overcome these problems we first analyzed existing test suites. This analysis consisted of two parts: First we researched hidden interdependencies among unit tests by

partially ordering their coverage sets. (Chapter 3 (p.29)). Our experiments with four case studies revealed that this technique exposes implicit ordering relationships between otherwise independent tests. This ordering strengthened our initial hypothesis that unit tests are inherently but only implicitly intertwined with each other. Furthermore, our experiments show that the partial order corresponds to a semantic relationship in which less specific unit tests tend to fail if more specific unit tests also fail. We showed how we can exploit this order to present the developers the most specific failing test case first.

In a second step (Chapter 4 (p.45)) we validated our experience that most unit tests focus on single methods by analyzing a case study consisting of more than 1000 unit tests. In this analysis we developed a taxonomy which categorizes the relations between unit tests and methods under test and between unit tests and other unit tests. We gave 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 method commands focusing on a single method. We also 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.

Synthesizing our findings of above two chapters and we continued by developing the taxonomy of Chapter 4 (p.45) into a meta-model called *Eg* orbiting around *method commands* which are exemplifying single methods (Chapter 5 (p.67)). We described in detail the responsibilities and collaborations of each class of our meta-model. Each responsibility was exemplified by a refactored version of the bank example. We validated our meta-model by explaining how it solves the problems described in Chapter 2 (p.9) together with a small case-study refactoring 40 unit tests of the graphics framework *Mondrian*.

6.2 Future Work

6.2.1 Integration of Traits

In Chapter 5 (p.67) we described our meta-model by describing the responsibilities for each class in a sequential order and gave example situations where these responsibilities would apply. But for explaining responsibilities we had to refer to responsibilities belonging to classes only later described in the meta-model. We believe that the reason for this is that classes are too coarse-grained to explain the responsibilities of a model in a logical, linear fashion, where each responsibility can only be explained after its required responsibilities have been explained before, and thus where each example and

command for a responsibility can build upon already existing examples and commands. Traits [Schärli, 2005] on the other hand shatter classes into smaller sets of coherent responsibilities. One the one hand there still exists no meta-model for integrating tests and examples with traits – which should be fruitful in itself as soon *traits* get more adapted by developers. On the other hand scenarios like the one above should be much easier sortable and describable using fine grained traits. It would be especially interesting to research the roles of required and provided methods in the context of providing and reusing exemplified methods.

6.2.2 Implementing Eg in other languages

The main constraint to implement the meta-model of Eg into other programming languages is that the links and the kind of test (*e.g.*, a *method example*, a *method test*, an *inverse test*) should be represented in parsable source-code. The links we have identified are

— the link between the *method command* and its *exemplified method*

— and the link between the result of a *positive method command* and the class of the result.

It is desirable to store *positive method commands* close to the classes of their returned object in order to provide developers directly with *tidy examples* for the classes when browsing them. In above meta-model we therefore made heavy use of class extensions. As a consequence we did not have to store positive method commands in a parallel test hierarchy but rather could treat them as *factory methods*. Note that class extensions work without executing the test cases first, so browsing examples for classes does not need any tool support as the developer can easily navigate to a class and discover the tests which would provide exemplifying instances of that class.

But extending the classes under test with class methods representing these positive method commands is not feasible in all object-oriented languages. The reason is that only some dynamic languages such as Smalltalk or Ruby know the concept of class extensions. As a consequence languages not providing the concept of class extensions can provide this link only (1) by dynamically executing all positive method commands and storing/providing the result in some cache or (2) by requiring the developer to provide the concrete type of the returned object within the method property/pragma annotating the test case.

Let us have a look on three wide-spread object-oriented languages in detail:

Ruby To our knowledge Ruby does not come yet with method properties. As a consequence one would have to implement the meta-model of *Eg* using coding conventions as introduced in Chapter 5 (p.67).

Connecting methods and method commands. Parsing the method to identify the last method called before the assertions is as well possible, also we can store a list of words which should not be treated as candidates for methods under command. If we needed to denote another method as the last one as the method under command, we can use the block/lambda expression of Ruby Proc»call as we use the block statement block in Smalltalk.

Connecting classes and exemplified instances. In Ruby on can extend any class with a method within a new module. A module can include methods and classes are kind of modules. A class can include several modules in a sorted manner. Traversing this sequence of included modules in the reverse order and asking if this method is defined in some of these modules for that class and if not repeating this procedure starting with the superclass delivers the module where the method is defined in. We can make use of the interface Module»method_defined?(id). Thus we can always identify if a method is a dedicated method command by detecting if its module is a dedicated example module. This means that we can denote and retrieve method commands in the same way as we denote and retrieve them in Squeak, namely by putting them into some dedicated module.

Parsing the kind of test. In order to recreate if the test was a method example, a method test or an inverse test, we can parse the source-code just as we did in the Smalltalk solution introduced in Chapter 5 (p.67): *e.g.,* if there are no assertions called after the method under test is called, it is a method example.

Java

Connecting methods and method commands. Java 1.5 introduced the notion of method properties. JUnit 4 is already making use of these method properties. We can use them to annotate the method under test ("mut") with the method as explained by Marschall in [Marschall, 2005].

```
@Testscape(mut =
"package.under.test.Class2#method2(ArgumentTypeOfMethod2)")
public void testXYZ()
....
```

Connecting classes and exemplified instances. In order to detect the type of the returned object in case of positive method commands, it would be necessary to be able to change the signature of the return type of the test from "void" to the according

signature of the returned object. Whereas it was possible in earlier versions of JUnit to return any kind of object, with the introduction of JUnit 4, tests "are void" again.

Parsing the kind of test. The kind of the test can be annotated with method annotations in an analogue manner to annotating the method under test.

Python Python does not know method or class extensions nor pragmas or method properties so we could not come up with a scheme to build the missing links within Python though we do not think it would be impossible to do so.

6.3 Lessons Learned

> Until real software engineering is developed, the next best practice is to develop with a dynamic system that has extreme late binding in all aspects. (Alan Kay)

During our work on better integrating unit tests into the developer's workflow we have learned that the principle of late binding cannot be overestimated. All techniques we relied on to enhance and play with the base system are relying on this principle.

6.3.1 Reifying and extending base classes and browsers helps

We have developed *Eg* within Smalltalk, a system architected with the principle of late-binding in mind. Relying on such a system can be a curse and a blessing. As Smalltalk stems back from times where computing power was precious, we still find essential code in current Smalltalk systems which is optimized for computers of the 1970's. As a consequence we have problems within Smalltalk at places where the design-principle of Smalltalk was given up in favor of performance. It was only last year that Denker *et al.* [**?**] introduced methods as first-class objects to make Squeak more evolvable. The good news is that systems with a clean and clear architectural metaphor as "Everything is an object" [**?**] have this kind of self-heeling capabilities. If it is not (yet) an object, we *can* make it one, especially in systems like Squeak whose source is open on all levels. We *can* extend the class Method to directly point to its exemplifying method commands or to a sampled set of executed methods. Likewise we easily can play with and extend the developer's interface of Smalltalk like its class browsers [**?**].

Another advantage of late-bound systems like Smalltalk is the *extensibility* of classes. We have heavily made use of this feature. Smalltalk and Ruby store methods in an

extensible dictionary. This allows us to add positive method commands as class extensions to the classes of whom these positive method commands provide exemplified instances of – within the scope of the example module and not within the scope of the exemplified module. So positive method commands can be displayed to developers as class factory methods which we find to be a natural role tests should and can provide. We thus strengthened our believe that class extensions are a powerful mechanism making systems like Smalltalk or Ruby superior to Java or Python in this aspect.

6.3.2 Coverage information is essential for testing

Having had already good success in industry by introducing coverage information to testing we learned during our work on this thesis how essential coverage information is for testing. A decent test- and developing environment cannot and should not live without a mechanism which provides coverage information. Coverage information (1) tells the developer what really happened, it lowers the value of static typing – which often hinders late binding – by providing concrete types (2), and it allows us to develop new views on tests (3) like our partial order and – as a consequence – new debugging facilities (4) like delta-debugging using partial orders as introduced in Chapter 5 (p.67).

6.3.3 Checked method examples are the atoms of unit testing

We believe that the problems we have with unit tests – *e.g.*, their missing navigability and composability – can be solved like most problems within the context of object-oriented programs: Make them first-class objects. We therefore analyzed and decomposed existing unit-tests suites down to smallest units we could find, namely into *checked method examples*. Those checked method examples not only focus on methods, the atoms of object-oriented programs, but also separate the scenario-building part from the verification part. Here the verification takes place *within the method under test* and not within the test. This separation of concerns allows developers to compose high-level examples without having to bother about writing concrete assertions, whereas they can focus on writing decent post-conditions and invariants within their domain models. Whereas most software-engineering books suggest to write post-conditions and invariants **[?]** **[?]** they are far from being common sense: In Squeak version 3.7 we could detect more than 1000 unit tests but only three(!) post-conditions. We believe the reason for this rareness of "design by contract" is that developers prefer automatically running tests over static contracts. Contracts do not come with a concept of how to run them, on the other hand current unit tests are not composable as they only provide concrete assertions which only apply in specific contexts. We therefore gave evidence that we can close

the gap between static code coming with contracts and runnable and composable examples by zipping those two valuable concepts via checked method examples Figure **??** (p.**??**).

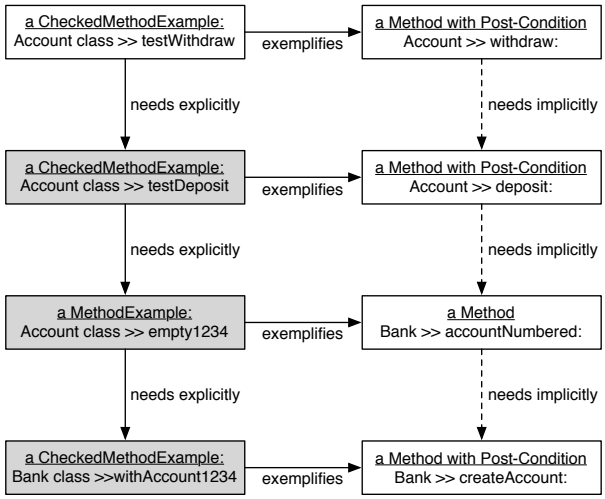

Figure 6.1: Bridging the gap between static code which comes with post-conditions and executable examples by zipping the methods under tests with checked method examples. As long as the checked method example on the top left calls the three methods at the bottom right to set up its scenario, we could even get rid of the method examples displayed in gray: All previously checked methods on the right would stay checked, and they still could provide undedicated examples.

Appendix A

First Validations of the Eg-Browser

In the following I present the results of the Master Thesis of Rafael Wampfler, with whom we implemented *Eg* and conducted the research presented in this appendix [**?**].

We built the meta-model for unit tests and an editor for it. But is the editor easier to use than the XUnit framework? Are developers faster in writing unit tests with our tool than with the classic XUnit? This chapter gives some initial answers.
The three main measurements of usability evaluations according to [Alan Dix, 2004] are:

— Assess accessibility of the system's functions

— Assess user's expirience of interaction

— Identify any problems

It is difficult to find good parameters for a user interface metric [Ivory and Hearst, 2001]. Counting the number of clicks or keyboard inputs is a simple metric, but it does not state something about the usability of the system. Stopping the time a user needs to fulfill a certain process can benchmark the usability, but this depends of how fast the user can interact with the system.

In this chapter we use two approaches to measure the usability of *Eg*. We first use a metric called GOMS (Goals, Operators, Methods, Selection) which can be applied without real users by manually counting the necessary steps and estimating their time to accomplish a given task. We apply this metric both to creating unit tests with XUnit

and with Eg. We then did a "speak aloud" experiment with five pairs of our research group and compared their experience of using XUnit with Eg under laboratory conditions.

A.1 GOMS keystroke-level model

GOMS keystroke-level model is a metric for user interfaces [John and Kieras, 1994] [Alan Dix, 2004]. GOMS computes the time spent for the needed actions like clicking the mouse, moving the mouse or entering a word on the keyboard. GOMS is a model-based evaluation and does not need a user to participate.
The GOMS model uses the values from table A.1. The times are mean values. The time

Table A.1: GOMS model

Operator	Time	Description
K	0.2 sec	Keying: Perform a keystroke or mouse click
P	1.1 sec	Pointing: Position the mouse pointer
H	0.4 sec	Homing: Move hands from keyboard to mouse
M	1.3 sec	Mental: Prepare for the next step
R	?	Responding: Computer responds to the user inputs

for performing a keystroke can vary from 0.12 (good typist) to 1.2 seconds (non-typist). The time needed for pointing on an object is dependent on the objects size and the distance to current mouse position. Fitt's law says: $t = 0.1 * log_2(Distance/Size + 0.5)$.
To calculate the time spent for an action the times of its subtasks is summed up. Mental preparation is needed before accomplishing different tasks and has be to considered too.

A.1.1 Validation of the EgBrowser

The EgBrowser is compared to SUnit with Refactoring Browser Extensions support. The GOMS model from section A.1 is used as metric.
Different tasks are measured with the tools.

A.1.2 Creating a test for an existing method

The initial situation of both is the same: a browser on the selector we want to test. The test uses the bank account example. The test should deposit an amount of money on

an new account and assure that the balance is greater than zero. GOMS measure the time spent to implement and run the test.
Summary: $134*K+11*P+12*H+21*M = 134*0.2+11*1.1+12*0.4+21*1.3 = 71$ seconds. The process took 178 steps. A lot of mental work and typing is needed with SUnit. A person would have longer to create this test because the mental work assume the person knows exactly what to. But coding is rarely straight forward, often the formulation changes or is try and error.
The class and the setup method can be shared by different tests. If they are already built for another test, the time to create a new SUnit test reduces to 43 seconds in 122 steps.
Summary: $32 * K + 6 * P + 4 * H + 8 * M = 32 * 0.2 + 6 * 1.1 + 4 * 0.4 + 8 * 1.3 = 25$ seconds. The process took 50 steps.

A.1.3 Creating a test for a new method

The same scenario in section A.1.2, except that the method withdraw: is not yet implemented. The browser has the bank class selected. The test class and setup method of SUnit can be reused from the first comparison.
Summary: $125 * K + 10 * P + 4 * H + 17 * M = 125 * 0.2 + 10 * 1.1 + 4 * 0.4 + 17 * 1.3 = 60$ seconds. The process took 156 steps. A lot of steps are reused from section A.1.2 as a new test class and setup method is not needed.
Summary: $91 * K + 11 * P + 10 * H + 15 * M = 91 * 0.2 + 11 * 1.1 + 10 * 0.4 + 15 * 1.3 = 54$ seconds. The process took 127 steps. The steps used in the debugger for defining the method are the same as with SUnit, but the debugger is opened automatically when a test runs the first time.

A.2 Usability Experiment

The best usability tests are done with real users. We built a tool for unit testing, so developers are our users. Developers have a base knowledge in using new applications, they get used to it faster than other users.
The experiment is done with real user participation under laboratory conditions. Laboratory conditions mean that the user tests the tool in a prepared setup and not in daily work (field studies). This conditions and the experiment in general can affect the results.

A.2.1 Test Setup

Participants The test users are developers from our research group. They all have at least a basic knowledge about unit testing.

Hypothesis

— The EgBrowser easier to use than SUnit.

— It is faster to accomplish the tasks.

— The EgBrowser is easy to use without previous knowledge.

Technics We did the experiment with pair programming. Pair programming is a common way to develop software. Two developers are drown by lot. We used think aloud as observation technic [Alan Dix, 2004]. Think aloud is easier to arrange with pair programming [Holzinger, 2005]. The users should describe and explain their thoughts while interacting with the system.
To analyze the protocol later, the process is filmed with a video camera. The camera records the conversation from the think aloud, but also the screen where the interaction with the system happened. The screen is also recorded with a screen capture application.
The user answers a short questionnaire after the experiment.

A.2.2 Tasks

The team has to write some tests with SUnit and the same with EgBrowser. The tool to start is chosen randomly to not bias the results.

1. Write an account class.
2. Write a test to create an empty account. Assure that the balance is zero.
3. Write a test to deposit an amount of money on the account. Assure that the account has the right balance.
4. Write a test to withdraw an amount of money from a not empty account. Assure that the balance is greater than zero.
5. Write a test to withdraw a too big amount of money from an account. The method should fail.
6. Browse between the withdraw test and the withdraw method.

A.2.3 Questionnaire

This questions are answered by the participants after doing the experiment. Each questions is responded for SUnit and the EgBrowser on a scale from 1 (disagree) to 5 (agree). The participants are the 10 developers from the usability experiment.

1. The system is easy to use
2. It is easy to learn
3. It supports the developer's workflow
4. It tells me what to do at every point
5. It is easy to recover from mistakes
6. It is easy to get help when needed
7. I always know what the system is doing

A.2.4 Questionnaire Analysis

The answers of the questionnaire are shown on figure A.1. The results are biased because most participants are SUnit experts and use SUnit every day. Another caveat is that SUnit is a well proven tools used for 30 years. The EgBrowser is in early testing stage and therefore has some bugs.

1. *The system is easy to use*
   ∅ SUnit: 4, ∅ EgBrowser: 3.1
   Because most participants are SUnit user, it is unambiguous they prefer SUnit and believe it is easier to use. Some users like the idea of tests and examples as reusable commands. The limiting factor was mainly the user interface.
2. *It is easy to learn*
   ∅ SUnit: 4.1, ∅ EgBrowser: 3.6
   Again higher votes for SUnit, but some participants cannot remember how hard it was to learn SUnit.
3. *It supports the developer's workflow*
   ∅ SUnit: 3.3, ∅ EgBrowser: 3.9
   The participants believe that the EgBrowser supports the workflow. Some developers are really fast in writing SUnit tests, other needs to browse the SUnit source code first.

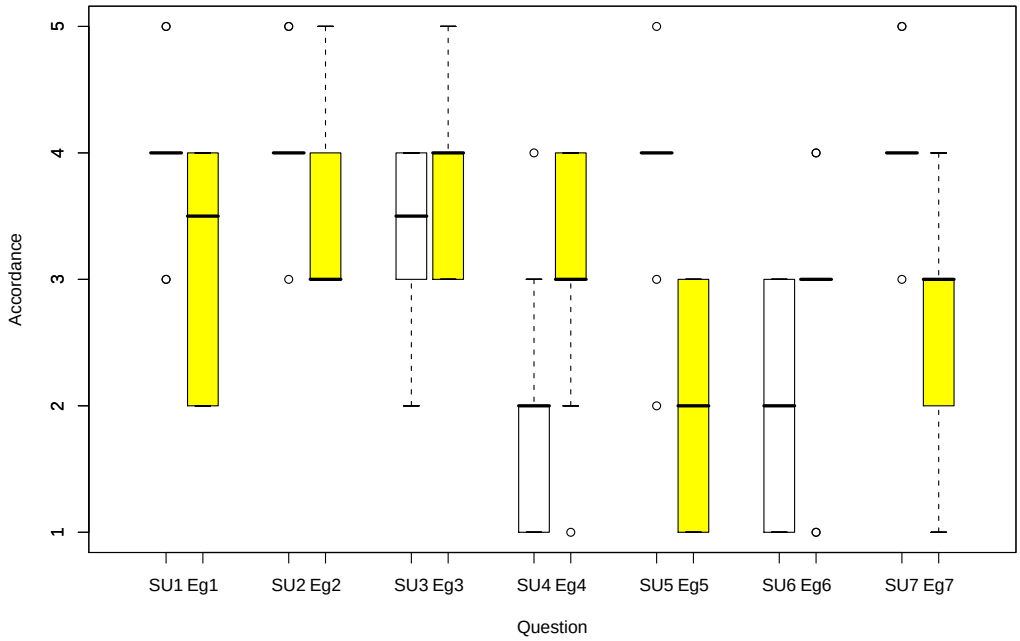

Figure A.1: Results of the questionnaire displayed as box plots

4. *It tells me what to do at every point*
   ∅ SUnit: 2, ∅ EgBrowser: 3.1
   EgBrowser has a defined number of input fields. If all fields are filled, the test should work. SUnit does not provide any steps, the user may be lost in an empty editor.

5. *It is easy to recover from mistakes*
   ∅ SUnit: 3.8, ∅ EgBrowser: 2
   Support for recovering from mistakes was limited in the used test version of EgBrowser, therefore again good results for SUnit where the debugger opens on the right context.

6. *It is easy to get help when needed*
   ∅ SUnit: 2, ∅ EgBrowser: 2.8
   Surprisingly the users think they get more help from EgBrowser than SUnit, but both tools does not provide any help system. In SUnit a lot of help can be obtained

by reading the source code. The EgBrowser leads the user and there are fewer situation where a user needs help. The EgBrowser has small syntax helps when hovering over an input field.

7. *I always know what the system is doing*
   ∅ SUnit: 4.1, ∅ EgBrowser: 2.7
   Another problem with the EgBrowser was the missing feedback to the user through the user interface. Whereas the participants knew what SUnit is doing in the background the way Eg works in the background was not immediately understood.

A.2.5 Video Analysis

The same tasks were performed with two different tools, *SUnit* and *EgBrowser*. The *EgBrowser* is a prototype able to handle the underlying meta model. The version of the EgBrowser used for the experiment had the following bugs:

— The interface was not always updated correctly. The EgBrowser needed a manual reload to create a new command or modifying existing commands.

— The EgBrowser did not warn the user if the example is compiled to another class than the receiver of the command. The default return value was the result and not the receiver.

— The commands were running in an anonymous context, so the debugger stack could not be used to fix mistakes.

— If a command failed and the EgBrowser was closed the command was not saves and the user had to rewrite it.

— Bad examples did not report a success after running and were displayed as failure in the interface.

Therefore the situation cannot be compared directly. The user had more problems with the EgBrowser interface bugs than expected.
To compare the EgBrowser with SUnit, the time spent with the bugs is subtracted from the EgBrowser time. The estimation of the bug time is ambiguous and in same cases too optimistic. Sometimes it is the time spent to implement the command for the third time. The developers already knew what to enter and were accordingly fast. This corrected time is displayed in the diagrams as *EgBrowser real*.

Task 1: Write an account class

The EgBrowser did not yet supported generating classes. The process of creating the class was the same as with SUnit. Because the tasks was done twice, it was faster implemented with the second tool.
Team 1, 3 and 5 started with Eg, team 2 and 4 with SUnit.
Team 1 wanted to implement the test first and the class after until they realized it is not possible.
Team 3 was very fast with SUnit because they first implemented the test case and defined the class in the debugger.
Team 4 had problems defining a class because both team member were not familiar with the VisualWorks environment.

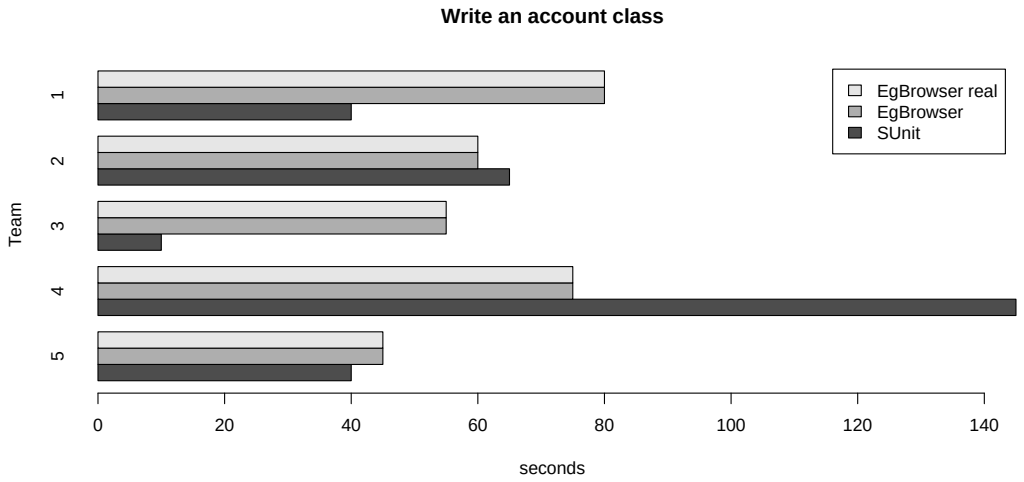

Figure A.2: Analysis of task 1

Task 2: Write a test to create an empty account

Team 2 and 3 implemented the test with Eg as fast as in SUnit.
The other Teams had more problems. Because the EgBrowser crash course was a bit short team 1 did not remember how to open the right browser and noticed it after creating the tests.
Team 4 and 5 had difficulties formulating the demanded test as one method command.

They compiled the command to a wrong receiver class because the receiver was the default return value. So they had to delete the wrong command and the compiled methods and re-implemented the command with the right return value.

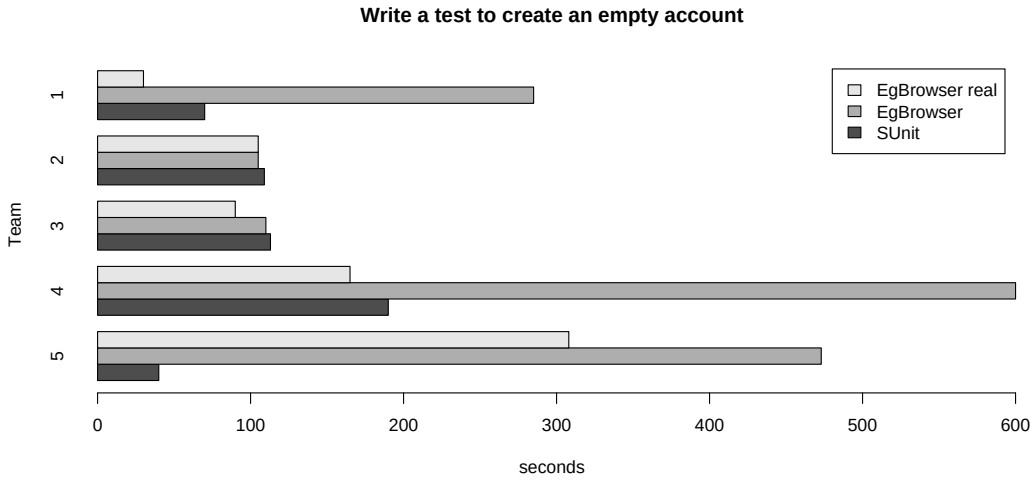

Figure A.3: Analysis of task 2

Task 3: Write a test to deposit an amount of money on the account

Overall this task was done faster than the previous because most teams comprehended the function of the EgBrowser.
Team 1, 3 and 4 had the first problems with refreshing the interface.
Team 5 wanted to reuse the command of task 2 in a strange way. Because the task 2 returned the wrong value, it did not work at all. Finally the model was out of sync and the team was very confused about the function of Eg.

Task 4: Write a test to withdraw an amount of money from a not empty account

The main goal of this task was to reuse the command from the previous task. The difference between SUnit and Eg is smaller than in the tasks before. Except for team 3, they had serious problems and had to implement the command multiple times.

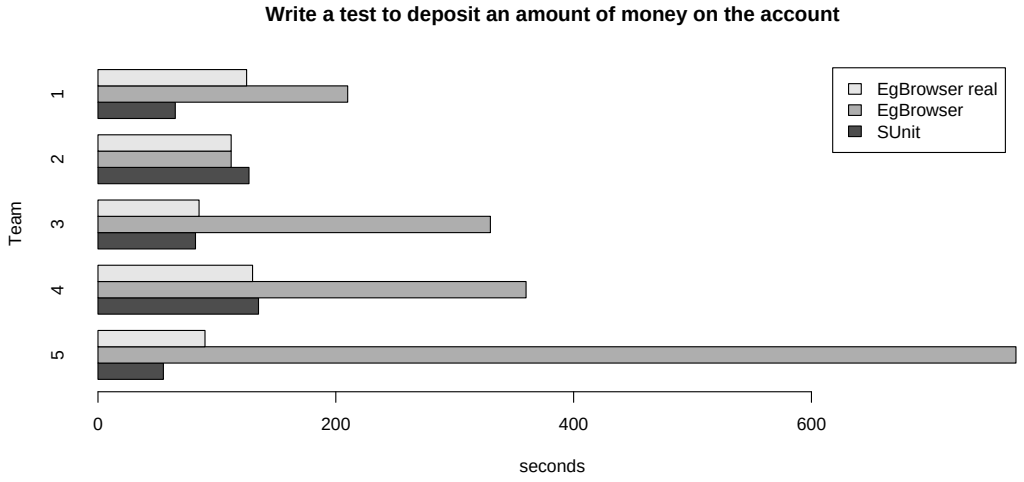

Figure A.4: Analysis of task 3

Task 5: Write a test to withdraw a too big amount of money from an account

Most teams did not know by heard how to raise an exception and how to catch it with a SUnit test. Some teams needed to browse the SUnit class to look up the syntax for the failed test.
Team 3 used most time to implement a working withdraw method and did not waste all the time to create an Eg command.

Task 5: Browse between the withdraw test and the withdraw method

This was not a real task. It was to demonstrate the features of Eg where you can browse between test and implementation with the meta model. SUnit does not have this feature and is not comparable.

A.2.6 Conclusion

Most teams chose a test driven approach: they implemented the first test before the class and the method implementation in the debugger. If the duration for the tasks of SUnit and the EgBrowser is compared, they are more or less equal. Most participants

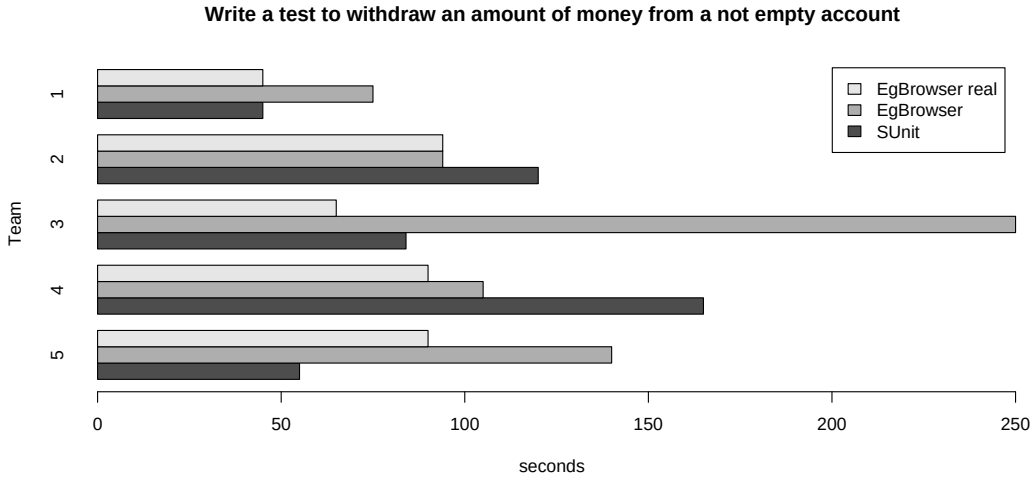

Figure A.5: Analysis of task 4

are fast with SUnit. They use SUnit every day and know how to create a test without mistakes.
The participants learned the usage of the EgBrowser fast. With the EgBrowser the time to create a test is reduced, so the inexperienced user had approximately as long to create a test as with SUnit.
Most teams had problems to create a command with Eg. The interface did not provide enough feedback. They were not sure if the model is comiling the right thing in the background.

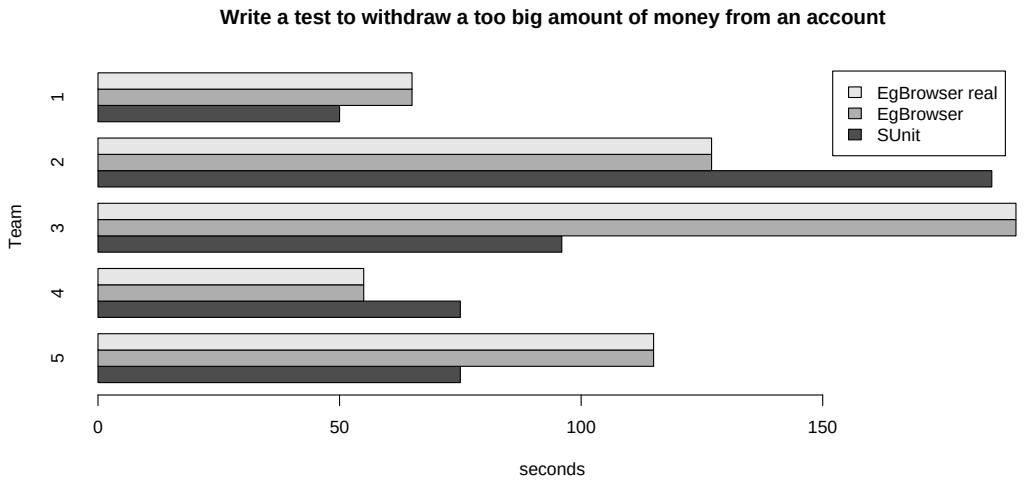

Figure A.6: Analysis of task 5

Bibliography

[Alan Dix, 2004] Gregory D. Abowd Alan Dix, Janet E. Finlay. *Human-Computer Interaction (3rd Edition)*. Prentice Hall, 2004.

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

[Anderson *et al.*, 1984] John R. Anderson, Robert Farrell, and Ron Sauers. Learning to program in Lisp. *Cognitive Science*, 8(2):87–129, 1984.

[ANS, 1983] ANSI/IEEE Standard 729-1983, New York. *IEEE Standard Glossary of Software Engineering Terminology*, 1983.

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

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

[Bergel, 2005] Alexandre Bergel. *Classboxes — Controlling Visibility of Class Extensions*. PhD thesis, University of Berne, November 2005.

[Bible *et al.*, 2001] John Bible, Gregg Rothermel, and David Rosenblum. A comparative study of coarse- and fine-grained safe regression test selection. *ACM TOSEM*, 10(2):149–183, April 2001.

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

[Brant *et al.*, 1998] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP 1998)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[Brooks, 1987] Frederick P. Brooks. No silver bullet. *IEEE Computer*, 20(4):10–19, April 1987.

[Bruner *et al.*, 1956] Jerome S. Bruner, Jacqueline J. Goodnow, and George A. Austin. *A Study of Thinking*. John Wiley & Sons, New York, NY, 1956.

[Bruntink and van Deursen, 2004] Magiel Bruntink and Arie 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, September 2004.

[Chari and Hevner, 2006] Kaushal Chari and Alan Hevner. System test planning of software: An optimization approach. *IEEE Transactions on Software Enginering*, 32(07):503–5099, 2006.

[Cleve and Zeller, 2000] Holger Cleve and Andreas Zeller. Finding failure causes through automated testing. In *Proceedings of the Fourth International Workshop on Automated Debugging*, August 2000.

[Cockburn, 2002] Alistair Cockburn. *Agile Software Development*. Addison Wesley, 2002.

[Cockburn, 2003] Alistair Cockburn. *Writting Effective Use Cases*. Addison Wesley, 2003.

[Cockburn, 2006] Alistair Cockburn. Dos equis driven design, 2006, http://alistair.cockburn.us/index.php/Dos_equis_driven_design. Retrieved August 25th 2006.

[Cohn, 2004] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[Cunningham, 2006] Ward Cunningham. Private communication, 2006.

[De Pauw *et al.*, 1998] Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 219–234. USENIX, 1998.

[Denker *et al.*, 2006] Marcus Denker, Stéphane Ducasse, and Éric Tanter. Runtime bytecode transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures*, 32(2-3):125–139, July 2006.

[Deursen and Moonen, 2002] Arie van Deursen and Leon 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.

[Deursen *et al.*, 2001] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In M. Marchesi, editor, *Proceedings of the 2nd International*

Conference on Extreme Programming and Flexible Processes (XP2001), pages 92–95. University of Cagliari, 2001.

[Ducasse *et al.*, 2001] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. The Moose Reengineering Environment. *Smalltalk Chronicles*, August 2001.

[Ducasse *et al.*, 2006] Stéphane Ducasse, Tudor Gîrba, and Roel Wuyts. Object-oriented legacy system trace-based logic testing. In *Proceedings 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 35–44. IEEE Computer Society Press, 2006.

[Ducasse, 1999] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

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

[Elbaum *et al.*, 2000] Sebastian G. Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *International Symposium on Software Testing and Analysis*, pages 102–112. ACM Press, 2000.

[Evans, 2003] Eric Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Fowler and Highsmith, 2001] Martin Fowler and Jim Highsmith. The Agile Manifesto. *Software Development Magazine*, 9(8), August 2001. http://agilemanifesto.org.

[Fowler *et al.*, 1999] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[Freeman *et al.*, 2004] Steve Freeman, Nat Pryce, Tim Mackinnon, and Joe Walnes. Mock roles, not objects. In *Companion of OOPSLA '04, ACM SIGPLAN Notices*, pages 236–246, New York, NY, USA, 2004. ACM Press.

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

[Gaelli *et al.*, 2004b] Markus Gaelli, Oscar Nierstrasz, and Stéphane Ducasse. One-method commands: Linking methods and their tests. In *OOPSLA Workshop on Revival of Dynamic Languages*, October 2004.

[Gaelli *et al.*, 2005a] Markus Gaelli, Orla Greevy, and Oscar Nierstrasz. Composing unit tests. In *Proceedings of SPLiT 2006 (2nd International Workshop on Software Product Line Testing)*, September 2005.

[Gaelli *et al.*, 2005b] Markus Gaelli, Michele Lanza, and Oscar Nierstrasz. Towards a taxonomy of SUnit tests. In *Proceedings of ESUG 2005 (13th International Smalltalk Conference)*, September 2005.

[Gaelli, 2004] Markus Gaelli. PhD-symposium: Correlating unit tests and methods under test. In *5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2004)*, volume 3092 of *LNCS*, page 317, June 2004.

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

[Gîrba *et al.*, 2004] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 40–49, Los Alamitos CA, September 2004. IEEE Computer Society Press.

[Gîrba *et al.*, 2005] Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

[Greevy *et al.*, 2006] Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice*, 2006. To appear.

[Hirschfeld, 2003] Robert Hirschfeld. AspectS – aspect-oriented programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, number 2591 in LNCS, pages 216–232. Springer, 2003.

[Holzinger, 2005] Andreas Holzinger. Usability engineering methods for software developers. *Commun. ACM*, 48(1):71–74, 2005.

[Holzner, 2004] Steve Holzner. *Eclipse*. O'Reilly, May 2004.

[Ingalls *et al.*, 1997] 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, ACM SIGPLAN Notices*, pages 318–326. ACM Press, November 1997.

[Ivory and Hearst, 2001] Melody Y. Ivory and Marti A. Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Comput. Surv.*, 33(4):470–516, December 2001.

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

[John and Kieras, 1994] Bonnie E. John and David E. Kieras. The GOMS Family of Analysis Techniques: Tools for Design and Evaluation. Technical Report CMU-CS-94-181, Carnegie Mellon University School of Computer Science, August 1994.

[Kay, 1993] Alan C. Kay. The early history of Smalltalk. In *ACM SIGPLAN Notices*, volume 28, pages 69–95. ACM Press, March 1993.

[Kiczales *et al.*, 2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceeding ECOOP 2001*, number 2072 in LNCS, pages 327–353. Springer Verlag, 2001.

[Koschke, 2003] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109, 2003.

[Lakoff, 1990] George Lakoff. *Woman, Fire, And Dangerous Things*. University Of Chicago Press, 1990.

[Lanza, 2003] Michele Lanza. Codecrawler — lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, pages 409–418. IEEE Press, 2003.

[Lieberman and Hewitt, 1980] Henry Lieberman and Carl Hewitt. A session with Tinker: Interleaving program testing with program writing. In *LISP Conference*, pages 80–99, 1980.

[Lieberman, 2001] Henry Lieberman. *Your Wish Is My Command — Programming by Example*. Morgan Kaufmann, 2001.

[Mackinnon *et al.*, 2000] T. Mackinnon, S. Freeman, and P. Craig. Endotesting: Unit testing with mock objects, 2000, http://www.mockobjects.com.

[Marschall, 2005] Philippe Marschall. Detecting the methods under test in Java. Informatikprojekt, University of Bern, April 2005.

[Mason John H., 2001] Watson Anne Mason John H. Getting students to create boundary examples. *MSOR Connections*, 1(1):9–11, 2001.

[Massol and O'Brien, 2005] Vincent Massol and Timothy M. O'Brien. *Maven: A developer's Notebook*. O'Reilly, 2005.

[Memon *et al.*, 2001] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Trans. Softw. Eng.*, 27(2):144–155, 2001.

[Memon *et al.*, 2003] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings IEEE Working Conference on Reverse Engineering (WCRE 2003)*, pages 260–269, Los Alamitos CA, November 2003. IEEE Computer Society Press.

[Meyer, 1992] Bertrand Meyer. Applying design by contract. *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, October 1992.

[Moore, 2001] I. Moore. Jester – a JUnit test tester. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*. University of Cagliari, 2001.

[Mugridge and Cunningham, 2005a] Rick Mugridge and Ward Cunningham. Agile test composition. In Hubert Baumeister, Michele Marchesi, and Mike Holcombe, editors, *Extreme Programming and Agile Processes in Software Engineering, 6th International Conference, XP 2005*, volume 3556 of *Lecture Notes in Computer Science*, pages 137–144. Springer, 2005.

[Mugridge and Cunningham, 2005b] Rick Mugridge and Ward Cunningham. *Fit for Developing Software: Framework for Integrated Tests (Robert C. Martin)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[Nierstrasz, 2002] Oscar Nierstrasz. Software evolution as the key to productivity. In *Proceedings Radical Innovations of Software and Systems Engineering in the Future*, Venice, Italy, October 2002. preprint.

[Nievergelt, 2006] Jürg Nievergelt. Die Aussagekraft von Beispielen. *Informatik Spektrum*, 29(4):281–286, 2006.

[Panko and Jr, 1996] Raymond R. Panko and Richard P. Halverson Jr. Spreadsheets on trial: A survey of research on spreadsheet risks. In *HICSS '96: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS) Volume 2: Decision Support and Knowledge-Based Systems*, page 326, Washington, DC, USA, 1996. IEEE Computer Society.

[Parrish *et al.*, 2002] Allen Parrish, Joel Jones, and Brandon Dixon. Extreme unit testing: Ordering test cases to maximize early testing. In Michele Marchesi, Giancarlo Succi, Don Wells, and Laurie Williams, editors, *Extreme Programming Perspectives*, pages 123–140. Addison-Wesley, 2002.

[Pawson and Matthews, 2002] Richard Pawson and Robert Matthews. *Naked Objects*. Wiley and Sons, 2002.

[Renggli, 2003] Lukas Renggli. SmallWiki: Collaborative content management. Informatikprojekt, University of Bern, 2003. http://smallwiki.unibe.ch/smallwiki.

[Rothermel and Harrold, 1996] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[Rothermel *et al.*, 1999] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proceedings ICSM 1999*, pages 179–188, September 1999.

[Rothermel *et al.*, 2001] Gregg Rothermel, Margaret Burnett, Lixin Li, Christopher Dupuis, and Andrei Sheretov. A methodology for testing spreadsheets. *ACM Trans. Softw. Eng. Methodol.*, 10(1):110–147, 2001.

[Rothermel *et al.*, 2002] Gregg Rothermel, Sebastian Elbaum, Alexey Malishevsky, Praveen Kallakuri, and Brian Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *Proceedings ICSE-24*, pages 230–240, May 2002.

[Saff and Ernst, 2003] David Saff and Michael D. Ernst. Can continuous testing speed software development? In *Fourteenth International Symposium on Software Reliability Engineering ISSRE 2003*. IEEE, November 2003.

[Schärli, 2005] Nathanael Schärli. *Traits — Composing Classes from Behavioral Building Blocks*. PhD thesis, University of Berne, February 2005.

[Schuh and Punke, 2001] Peter Schuh and Stephanie Punke. ObjectMother, easing test object creation in XP, 2001, http://www.xpuniverse.com/2001/pdfs/Testing03.pdf. http://www.xpuniverse.com/2001/pdfs/Testing03.pdf.

[Soanes, 2001] Catherine Soanes, editor. *Oxford Dictionary of Current English*. Oxford University Press, July 2001.

[Spoon, 2006] S. Alexander Spoon. Package universes: Which components are real candidates? Technical Report LAMP-REPORT-2006-002, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

[Summers, 1977] Phillip D. Summers. A methodology for Lisp program construction from examples. *J. ACM*, 24(1):161–175, 1977.

[van Geet, 2006] Joris van Geet. Coevolution of software and tests: An initial assessment. Diploma Thesis, University of Antwerpen, July 2006.

[Watson Anne, 2002] Mason John H. Watson Anne. Student-generated examples in the learning of mathematics. *Canadian Journal of Science, Mathematics and Technology Education*, 2(2):237–249, 2002.

[Wikipedia, 2006] Wikipedia. David Wheeler, 2006, http://en.wikipedia.org/wiki/David_Wheeler. Retrieved August 10th 2006.

[Winger, 2004] Eric Winger. Pragmas: Running tests on method change, 2004, http://www.cincomsmalltalk.com/userblogs/eric/blogView?entry=3265627283. Retrieved August 10th 2006.

[Wirfs-Brock and McKean, 2003] Rebecca Wirfs-Brock and Alan McKean. *Object Design — Roles, Responsibilities and Collaborations*. Addison-Wesley, 2003.

[Wittgenstein, 1953] Ludwig Wittgenstein. *Philosophische Untersuchungen*. Blackwell, Oxford, 1953.

[Wong *et al.*, 1997] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 230–238, November 1997.

[Zeller and Hildebrandt, 2002] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, SE-28(2):183–200, February 2002.

[Zeller, 2005] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, oct 2005.