Composing Tests from Examples

Markus Gaelli
Rafael Wampfler
Oscar Nierstrasz
Software Composition Group, University of Bern, Switzerland

Understanding and maintaining complex software systems is a difficult task. In principle, tests can be a good source of information about how the system works. Unfortunately, tests are frequently unstructured and disconnected from each other and from their units under test. We propose a new approach to organizing unit tests in which tests produce examples of their units under tests which also can be reused for composing higher-level tests. The approach is based on the EG meta-model, which classifies tests according to their granularity and their goals. We have developed the EGBROWSER, an experimental tool for specifying tests that conform to the EG meta-model while keeping track of the connection between tests themselves and their units under test. Initial usability studies suggest that the approach is both easy to learn and more efficient than the programmatic approach to developing tests.

1 INTRODUCTION

Good documentation often makes use of concrete examples, since they are prescriptive and thus typically easier to understand than abstract descriptions. Code examples are useful because they illustrate how the code is instantiated and used, and because they are executable, hence open to experimentation. Furthermore code examples can be composed to exemplify higher level code constructs.

Unfortunately code that generates didactic examples is generally missing from complex software systems.

Tests are often held up as an indirect form of documentation since they typically encode a concrete scenario of code usage. Tests typically build up and exercise examples from the code base, and therefore provide information about how the code can be instantiated and used. Unfortunately the examples created in the tests are normally not accessible outside the test. So tests do not offer a means to interact and experiment with instances nor do they allow developers to compose higher level test scenarios out of lower level ones.

We identify two main problems with the way that unit tests are conventionally programmed that impedes their full exploitation as "live" documentation of the tested system:

- Tests are not explicitly linked to their units under tests. This can make it hard to determine what functionality is being tested. Heuristics can help to
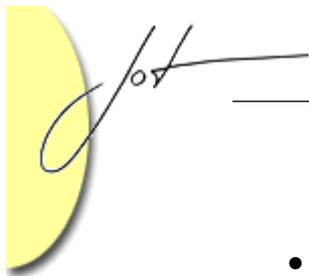
recover this information, but only with limited reliability [vG06][GLN05].

- Tests typically form a flat hierarchy. It is rare for tests to be related either by inheritance or composition. This makes it harder to understand the natural relationships between tests (such as coverage), and may also lead to duplicated boilerplate code in tests that cover different scenarios for the same units. As a further consequence one error can break several (overlapping) tests. But developers using flat tests cannot be pointed to the failing test with the smallest footprint providing the narrowest debugging context without using additional techniques [GLNW04].

We propose a new approach to unit testing in which tests are explicitly composed from examples that are generated by other tests. EG is a meta-model for unit testing that classifies tests according to their granularity and purpose. The abbreviation *e.g.* means "for example":

> It is short for the Latin exempli gratia, "for the sake of example." A list of examples may be preceded by e.g.: "She loved exotic fruit, e.g., mangoes, passion fruit, and papayas."[EDH02].

A key feature of EG is that most tests are expected to return a result, which usually is a changed example of the unit under test. Tests can therefore be composed from examples produced by other tests.

We claim that EG promotes the exploitation of tests as live documentation in several ways. First of all, EG makes the link between tests and the unit under test explicit. This makes it easier to navigate between code that we wish to understand and the relevant tests. Next, tests produce live examples, thus not only documenting how instances are produced, but also providing entities that one may interact with and explore. Complex tests can be composed from simpler tests, making explicit the relationship between them. In particular, these relationships aid navigation by suggesting which tests are more general or specific.

To validate this claim, we have developed EGBROWSER, an interactive editor for developing tests that conform to the EG meta-model. With EGBROWSER, tests can be composed from examples produced by other tests. The actual test source code is generated by EGBROWSER and maintained together with the units under test. As a consequence the link between tests and units under test is explicit and can therefore be exploited for various purposes, such as documentation, reverse engineering and simple coverage analysis.

We have carried out initial usability studies reflecting a number of typical tasks related to test development, and have gathered evidence that even our experimental prototype offers significant advantages over the conventional, programmatic approach to developing tests. In particular, many tasks can be performed much more efficiently with dedicated tool support supporting the developer's workflow. Furthermore, a generative approach is easy to learn, without significantly restricting the kinds of tests that can be written.

In Section 2 we establish various problems that arise with currently popular approaches to unit testing. Section 3 provides an overview of the EG testing meta-model. In Section 4 we present features of the EGBROWSER test editor. We discuss the validation of EG and EGBROWSER in Section 5. First we present a GOMS keystroke model which demonstrates that EGBROWSER is more efficient for various test development tasks than SUnit. Next we present a small usability study with five teams of developers carrying out a series of typical testing tasks. Finally we consider a case study in which an existing, non-trivial test suite is migrated to the EG meta-model. Section 6 presents our concluding remarks.

2 PROBLEMS WITH CONVENTIONAL UNIT TESTING

Unit testing frameworks for various programming languages exist today, mostly based on "XUnit", where "X" stands for the language in question. A prime example is SUnit [Bec]. SUnit tests produce no results and cannot be composed or reused. There is no explicit link between an SUnit test and its unit under test. As a consequence it is tedious to navigate between a test and its corresponding unit under test. Furthermore, there is no automatic way to determine from the code of a test which unit is actually under test. Lightweight heuristics can help [GLN05], but offer no guarantee that the correct unit under test has been identified.

An SUnit test operates on a "fixture" consisting of one or more objects (*i.e.*, examples) that perform in the tested scenario. The fixture may be created in the setUp method, but this is not required. It is common to forgo the setUp method entirely and create the required instances in the test method itself. In any case, the example instances used by an SUnit test are not typically created by a dedicated method, and are consequently not available outside of the context of the test itself. As a consequence, even though an SUnit test will normally create and manipulate examples of interest, these examples cannot be used, examined or manipulated without refactoring the test code.

JUnit[1] is a rewrite of SUnit in Java. The supported meta-model differs only slightly from that of SUnit. JUnit is the de facto standard for unit testing in Java.

JUnit 4 offers a few new features. Tests are no longer denoted by means of a naming convention but with annotations. Annotations are supported in Java since J2SE 5.0 (Java 1.5)[2]. With JUnit 4 any method can act as a test and the test class does not need to extend a test case class. Nevertheless, the key problems with XUnit remain: there is no explicit link between tests and units under test, test do not return results, tests cannot be composed, and examples are not accessible outside of tests.

A further problem with XUnit is that test creation is ad hoc and unstructured: Developers obtain no guidance in how to organize tests. Since tests cannot be reused

---

[1] www.junit.org
[2] java.sun.com/j2se/1.5.0/
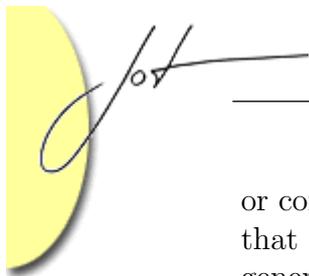
or composed, there is a tendency to make use of boilerplate code, especially in tests that cover sets of related scenarios. GUIs are available to run tests, but not in general to develop them [Wam06].

Dakar Testing [CK06] is an experimental extension to VisualWorks Smalltalk based on SUnit that maintains the link between tests and units under test, but it also does not provide means to compose tests based on examples.

3   THE EG META-MODEL

The EG meta-model was developed empirically by analyzing and categorizing a large number of existing SUnit tests. The result of this study was a taxonomy of unit tests which distinguishes tests according to their granularity and purpose. As it turned out, the bulk of the SUnit tests studied focussed on a single method, and only few tests bundle together a suite of tests [GLN05].

We have refined this taxonomy into a new testing meta-model, called EG, in which *method commands* return a result, which is an example of the unit under test [Gae06]. As a consequence, tests can be composed: A test checking the correct functionality of withdrawing money from an account can be composed out of a test checking the correct functionality of depositing money on an account simply by reusing the account object that results from the deposit test.

Furthermore tests are explicitly linked to their exemplified units [GGN05] allowing developers to see methods with their exemplifying usages side by side. This meta-model has been implemented in Smalltalk. Tests in EG are presented to the developer abstractly, *i.e.*, as objects, not as source code. But the meta-model supports the generation of source code from tests to provide persistence as well as a textual format for viewing tests. The meta-model encodes which method is under focus for a given test [GND04]. EG also provides means to gather information about tests as they are run.

We will first provide an overview of the test classes provided by EG, and then we will briefly discuss source code generation and coverage information.

Eg Test Classes

Figure 1 presents the key class of the EG meta-model. In EG, all tests are *commands*, that is they either inherit from or implement the interface of the (abstract) Command class. A command may be run, it returns a result, and it also produces a test result. A command belongs to a package and it has a name. There are two broad categories of commands: a MethodCommand focuses on a single method, and a MultipleMethodCommand focuses on multiple methods.

A *method example* is the simplest form of method command. It represents an example of sending a single message to a particular example object. The method
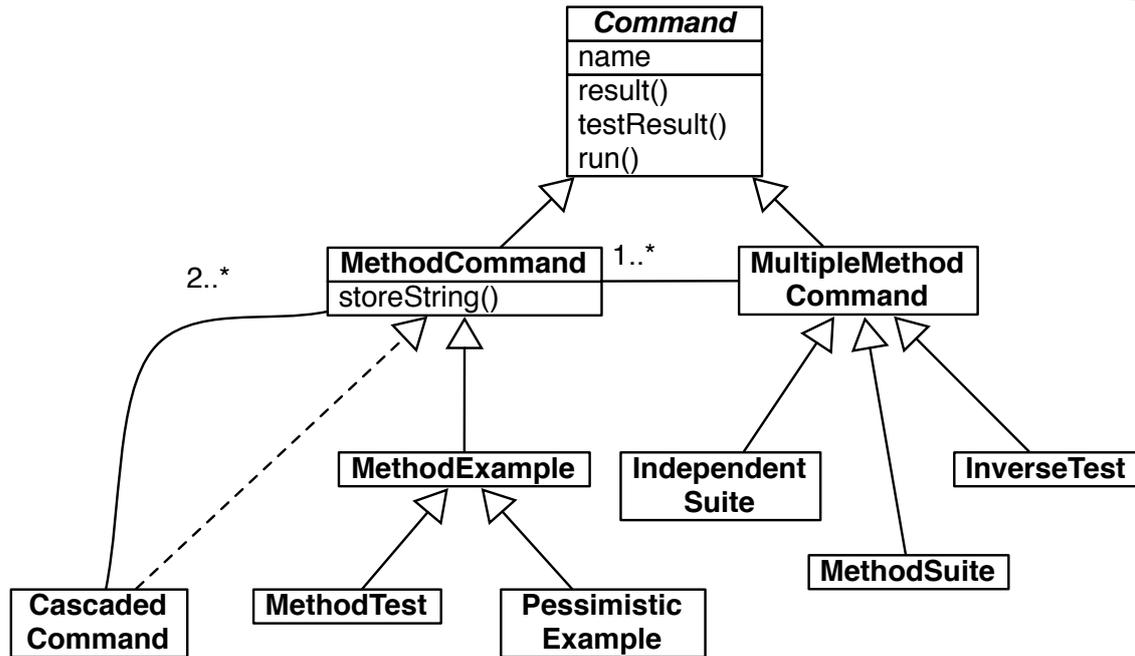
Figure 1: Eg meta-model

under test should be a "command" in the Eiffel sense of the "command-query separation" [Mey97] thus performing an action and changing or creating an object within the context of the method. The method example then returns exactly this changed or created object as a return value. A method example can function as a unit test in the case that the method under focus has an explicit postcondition. The test contains no assertion of its own, since the postcondition belongs to the method under focus. Such a test is called a *checked method example*.[3]

In Figure 2 we see an abstract representation of a method example for an Account class. The associated package is "Eg-Bank", the receiver is a new instance of the Account class (*i.e.*, the result of evaluating the Smalltalk block closure [Account new]), the message sent is "deposit:" (*i.e.*, in Smalltalk, the literal symbol #deposit:) and the argument sent with the message is the number 100. Running the example (*i.e.*, by sending the run message to this instance) will cause the receiver to be instantiated, the message deposit: 100 to be sent, and the result of this message send (*i.e.*, the modified Account instance) to be returned. Having a test which returns an *interesting result*, namely a filled account, we can reuse it in a test checking the correct functionality of an according withdraw method. The developer thus can compose higher level tests aided by EG as we show in Figure 4.

Note that there is no source code associated with a method example. The method example is created interactively by using factory methods of the meta-model classes.

---

[3]Developers can ensure the automatic execution of "query methods" by calling them in side-effect free postconditions.

```
exampleDeposit:MethodExample
package = 'Eg-Bank'
receiver = [Account new]
selector = #deposit:
parameters[*] = { 100 }
```

```
Account class >> exampleDeposit
  <egClass: #'Eg.MethodExample' method: #deposit: >
  | aReceiver aResult |
  aReceiver := Account new.
  aResult := aReceiver deposit: 100.
  ^aResult
```
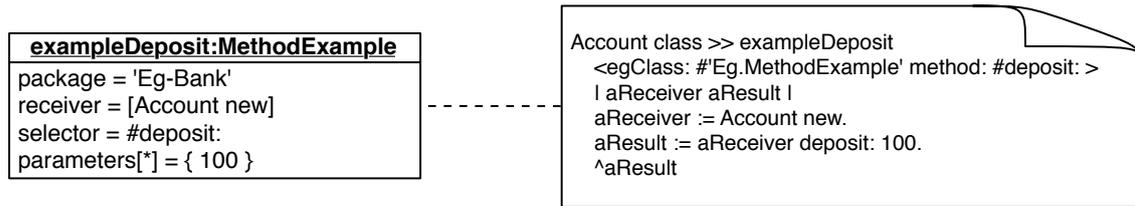
Figure 2: A method example and its generated code.

Nevertheless, source code may be generated. Source code representation of tests is convenient for: (i) persistence; (ii) browsing; (iii) execution; and (iv) explicitly invocation and composition. Tests can be saved as source code rather than as objects, they can be viewed in source form, they can be compiled and directly executed rather than interpreted, and tests stored as methods can be explicitly invoked, also be other tests. In this case an exampleDeposit method would be generated on the class side of the Account class.[4]

A *method test* extends a method example with one or more explicit assertions. In Figure 3 we augment the method example of Figure 2 with the assertion aResult balance = 100. In the Smalltalk implementation, the assertions consist of an ordered collection of boolean expressions (*i.e.*, Smalltalk blocks). Assertions may reference the receiver (*i.e.*, the example object), parameters to the message send, and the result of sending the message.

```
exampleDeposit:MethodTest
package = 'Eg-Bank'
receiver = [Account new]
selector = #deposit:
parameters[*] = { 100 }
assertions[*] = { [aResult balance = 100] }
```

```
exampleWithdraw:MethodTest
package = 'Eg-Bank'
receiver = [Account exampleDeposit]
selector = #withdraw:
parameters[*] = { 20 }
assertions[*] = { [aResult balance = 80] }
```
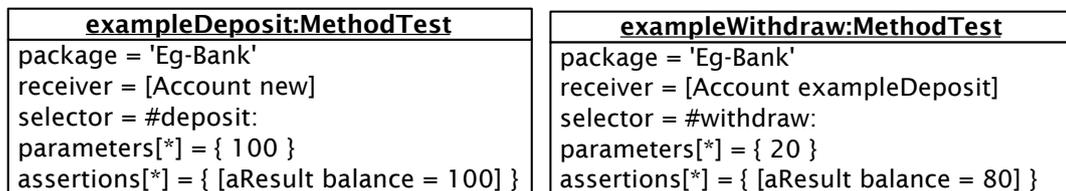
Figure 3: A method test and a cascaded command

A *pessimistic example* is simply a method example that is expected to raise an exception.[5]Pessimistic examples do not return reusable objects. A *cascaded command* is any method command whose receiver is the result of another method command. In Figure 3 the receiver of the exampleWithdraw cascaded command is the result of evaluating the exampleDeposit method test.

Multiple method commands represent suites of commands. An *independent suite* is a collection of independent commands. It is analogous to an XUnit test suite.

A *method suite* consists of a suite of commands that focus on the same selector.

---

[4]Since knowledge of Smalltalk is not critical to this paper, we show this code without further comment.

[5]These kinds of tests are also known as "exception test cases".

This kind of command is automatically generated by the IDE.

An *inverse test* is similar to a method test, except that a second message is sent to undo the effect of the first. The test then checks whether the result is equivalent to the original receiver.

Eg Coverage

During the execution of a command, additional information can be gathered. A method command calls one method, so the method command serves as an example for this method. The receiver and parameters are exemplified values for the method. A *coverage* on the method calls displays all executed messages with the command. These methods are not identifiable in the command source.

4 EGBROWSER

EGBROWSER is a prototype tool for developing tests that conform to the EG meta-model. In this section we present a brief overview of EGBROWSER. For more details, please consult Wampfler's Masters thesis [Wam06].

EGBROWSER has been designed to support the developer in writing tests. The process of developing a test has been simplified in various ways. In particular, EG has been designed to address the following requirements:

- The user can navigate directly between units under tests and their tests.

- Typing is minimized — tests are generated with the help of forms.

- Generated tests conform to the EG meta-model.

- Tests can be reused to compose new tests — composition is supported by drag-and-drop.

- Test runner is integrated — graphical feedback indicates test status.

- Selected views are automatically generated, such as method suites focussing on the same method.

EGBROWSER is implemented as an extension of the StarBrowser[6], a modern Smalltalk class browser that provides the ability to classify software artifacts while browsing [WD04].

The EGBROWSER tool provides a graphical interface driven from the units under test. The interface allows tests to be created that conform to the meta-model, and automatically generates the test source code from this interface. The tool therefore

---

[6]homepages.ulb.ac.be/~rowuyts/StarBrowser/
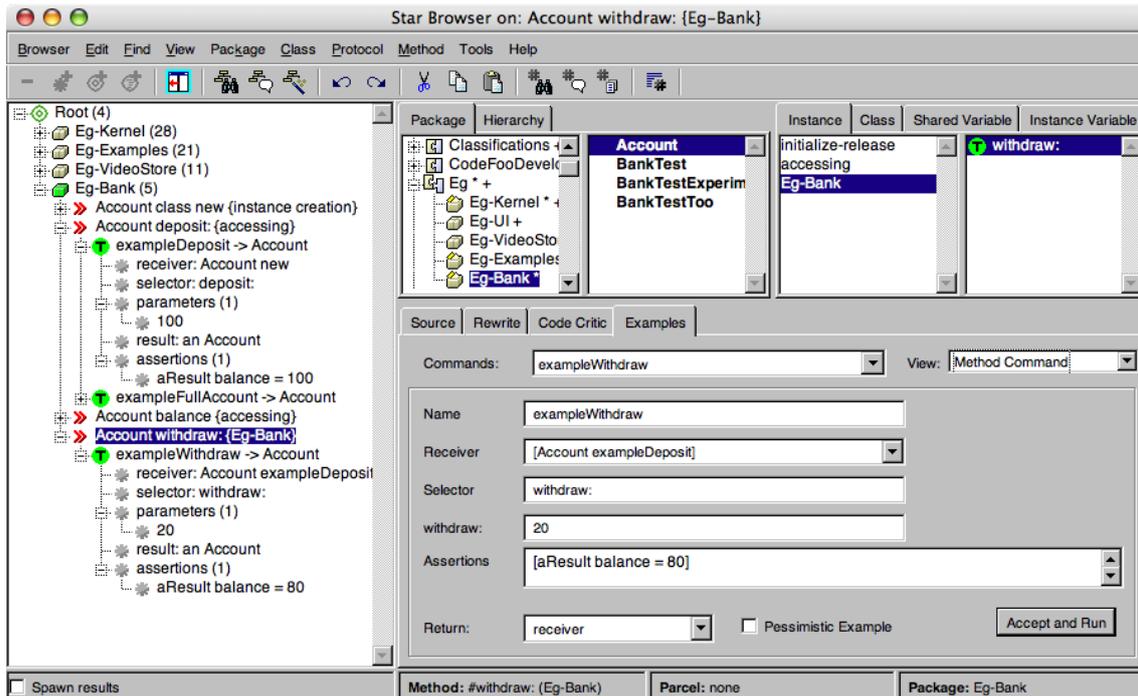
Figure 4: Defining a Method Command

maintains the link between tests and the classes and methods that they test. A key element is that most of the tests can return an example object that can be reused to compose other tests. This not only encourages a better structuring of tests, but it also supports program understanding since one may instantiate and interact with examples that are produced by running tests.

In Figure 4 we see a snapshot of EGBROWSER. In the leftmost panel we see a StarBrowser classification tree consisting of packages of interest and Method Commands grouped by methods under test. Commands are assigned an icon indicating their role within the EG meta-model: **E** for a Method Example, **T** for a Method Test, **I** for an Inverse Test and **S** for any kind of suite of method commands. The icon colour is green if the command runs without any errors or failures, otherwise it is red.

The next four panes at the top provide the usual views of Smalltalk packages, classes, method categories and methods. At the bottom, to the right of the classification tree, is a tabbed view where one may, amongst other things, edit source code or create EG tests.

In Figure 4 we see the EGBROWSER positioned on the method withdraw: within the Eg-Bank method category of the Account class inside the Eg-Bank package.

We are also positioned on the Examples tab which allows us to use a forms-based interface to view and edit EG examples. Here we have opened a Method Command view which allows us to edit an individual method command. This is a cascaded

command, since the receiver is the example returned from another command (Account exampleDeposit). This command will send withdraw: 20 to the example and test that [aResult balance = 80]. If no assertion is given, then a method example is built rather than a method test. In this case we decide that we should return the receiver, not the result, since the result (a boolean) is not useful as an example for any further tests. If we expect the test to fail, we may flag it as a *pessimistic example*.

Both receivers and parameters can be specified by dragging and dropping them from the classification tree.

Pressing the *Accept and Run* button will cause the source code for this command to be generated and compiled on the class side of the Account class, as the method exampleWithdraw. The test will then be evaluated, *i.e.*, by evaluating Account exampleWithdraw run.

By changing the view to Method Suite (menu at top right of the Examples tab), one will see instead a list of all command that focus on the currently selected method. The Coverage view, on the other hand, lists all the methods called by the currently selected method command. This information is gathered by temporarily installing *method wrappers* [BFJR98, Duc99] on all selectors of the package containing the command. Finally, the Command Suite view allows one to build an independent suite by bundling a subset of the available commands.

5   VALIDATION

We explore the effectiveness of the EG meta-model and the experimental EGBROWSER test editor from several perspectives. First, we develop a keystroke model to evaluate how efficiently certain tasks can be accomplished with EGBROWSER as opposed to with the standard SUnit testing framework. We then carry out a usability experiment with five teams of programmers carrying out a series of tasks with both SUnit and EGBROWSER. Finally, an experiment is carried out to manually convert an existing test suite for a real application from SUnit to the EG meta-model. In each case, convincing evidence has been gathered to support the claim that the EG meta-model offers benefits over the traditional approach to testing.

GOMS evaluation

We have developed a Keystroke-Level GOMS (Goals, Operators, Methods, and Selection) model [CNM83] of EGBROWSER, which measures the number of keystrokes that are required to carry out typical test development tasks. This model is then used to demonstrate that EGBROWSER is more efficient in many tasks than the manual approach.

We use the GOMS execution time prediction as a metric; less time means better
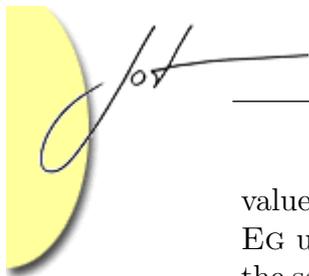
values. We select three typical tasks of the EGBROWSER compare them to SUnit. EG uses a meta-model whereas SUnit uses inheritance. The tools are not built for the same tasks. Therefore SUnit will have problems with tasks that EG is specialized for. Where a feature is missing in SUnit we use the simplest way to get a similar result.

The results of the comparison are shown in Table 1, where K measures the number of keystrokes or mouse clicks, P measures number of positioning the mouse pointer, H means homing – the number of times needed to move the hands from the keyboard to the mouse, and M denotes the number of mental steps to prepare the next action. *Time* sums up all numbers multiplied by their according average time. The different steps and average timings are in detailed in the appendix of Wampfler's MSc thesis [Wam06].

Table 1: KLM-GOMS results

Task	Tool	K	P	H	M	Time
Creating a test (existing method)	SUnit	134	11	12	21	71 sec
	EG	32	6	4	8	25 sec
Creating a test (new method)	SUnit	125	10	4	17	60 sec
	EG	91	11	10	15	54 sec
Browsing between test and implementation	SUnit	3	2	0	3	7 sec
	EG	3	3	0	2	7 sec

Creating a test for an existing method. The initial situation of EG and SUnit is the same: a browser opened on the selector of the new test. The test should deposit an amount of money on a new account and assure that the balance is greater than zero. GOMS is used to measure the time spent to implement and run the test.

SUnit used 178 steps in 71 seconds, assuming that the developer knows exactly what to do. A great deal of mental work and typing is needed with SUnit for the individual steps. In reality, even more time would be needed, since time to formulate the solution is not taken into account. The class and the setup method can be shared by different SUnit tests. If they have already been built for another test, the time to create a new SUnit test reduces from 71 seconds in 178 steps to 43 seconds in 122 steps.

EG used 50 steps in 25 seconds. The main advantage of EG over SUnit is that more code is generated and the user needs to type less. Because the EGBROWSER guides the developer in writing tests it entails less mental work than in a free text field like SUnit.

Creating a test for a new method. This scenario is similar to the previous one, except that the method #withdraw: is not yet implemented. The bank test class is selected in the browser. The test class and setup method of SUnit can be reused from the first comparison.

SUnit used 156 steps in 60 seconds. Numerous steps can be reused from the previous scenario since a new test class and setup method are not needed.

EG used 127 steps in 54 seconds. 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.

Browsing between test and implementation. The starting point is a newly created test similar to the end position of the first scenario.

EG can use the meta-model to determine which method is tested. The model information can be accessed through the command tree view.

SUnit does not have a meta-model and therefore the method under test is not denoted in the test code. The user has to find the method manually by reading the code. If he finds the method under test he can browse the implementors to see the method source.

Usability study

A usability study was carried out in which five teams carried out a series of tasks with both EGBROWSER and plain SUnit. The results were videotaped and analyzed with the help of questionnaires. Although the usability study was quite small, and the EGBROWSER prototype was not as polished as it could have been, still the study showed that EGBROWSER was highly effective, even in a prototype state.

The experiment was done with real user participation under *laboratory conditions*, *i.e.*, in a prepared setup and not in daily work (field study). The test users were developers from our research group. They all have experience with unit testing.

The study was intended to test the following hypotheses: (i) The EGBROWSER is easier to use than SUnit. (ii) It is faster to accomplish the tasks. (iii) The EGBROWSER is easy to use without previous knowledge.

The study consisted of a series of tasks solved by pair programming. Two developers were drawn by lot, resulting in five pairs. To analyze the record later, the process was filmed with a video camera. Each team wrote some tests with SUnit and the same with EGBROWSER. The tool to start is chosen randomly to not bias the results.

The tasks:

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.
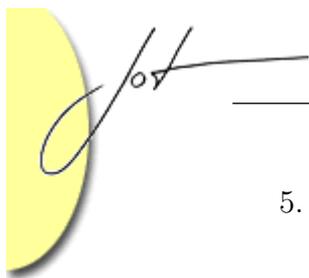
5. Write a test to withdraw a too big amount of money from an account. The method should fail.

Participants were familiar with SUnit, but had no previous experience with EgBrowser. The version of the EgBrowser used for the experiment suffered from the following defects:

- 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 was compiled to another class than the receiver of the command. The default return value was the result and not the receiver.

- Commands ran 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 saved and the user had to rewrite it.

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

Since users experienced more problems with the EgBrowser prototype than expected, the two situations could not be directly compared. In order to compensate for these problems, the time spent dealing with these bugs is subtracted from the EgBrowser time.

Task 1 was not supported by EgBrowser, so the results are identical for SUnit and Eg. Tasks 4 and 5 are similar. We therefore summarize the results for tasks 2, 3 and 4.

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, teams 1 and 4 did not remember how to open the right browser and noticed it after creating the test. Both teams returned the result instead of the receiver, therefore the command was compiled to the integer class. This command was not reusable for other commands. They deleted the wrong command and needed to re-implement the test with the right return value.

Team 5 had difficulties formulating the required test as a method command. They did not realize that a message can be sent to an assertion variable, so they wrote an example and another test to check if the example was right. This did not work because the meta-model compiled the example to the wrong class.

Task 3: Write a test to deposit an amount of money on the account. Overall this task was done faster than the previous one because most teams now understood the functionality of the EgBrowser.

Team 3, 4 and 5 had the first problems with refreshing the interface. Team 3 and 4 compiled the command to the integer class. They needed to re-implement the tests. Team 5 wanted to reuse the command of task 2 in a unsupported 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 got confused about the functionality 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.

Team 4 again had to redo the command because the interface did not update correctly.

Most teams chose a test-driven approach: they implemented the first test before the class and the method body in the debugger. If the duration for the tasks of SUnit and the EGBROWSER is compared, they are more or less equal. Most participants are fast with SUnit. They use SUnit every day and know how to create a test without mistakes.

The participants learned how to use EGBROWSER quickly. With the EGBROWSER the time to create a test is reduced, so the inexperienced user took approximately as long as with SUnit.

Most teams were initially confused when creating a command with EG. The interface did not provide enough feedback. They were not sure if the model is compiling the right thing in the background. A solution could be to explain first how to write a test manually as source code.

Converting SUnit Tests to EG

A final validation concerned the migration of a number of existing, manually written tests to the EG framework. The case study suggested that, in most cases, traditional tests can be easily transformed to the more structured meta-model.

We chose Mondrian as application for the case study. Mondrian is a scriptable, dynamic visualization framework developed by Meyer *et al.* [MGL06]. Mondrian can handle many sorts of data and supports different charts as output. Currently Mondrian has about 190 SUnitToo tests. We randomly selected a sample of 35 tests to categorize using the meta-model.

A majority of 18 test cases can be converted to method tests. One problem encountered is that the method under test is sometimes ambiguous. Often the method under test is the #open method that opens the graphics window. In this method the points and bounds of the figures are calculated, and the assertion checks these values. Another 9 test cases we categorized into two-method test suites. These tests can be clearly separated into two tests because there are two different setup procedures with corresponding assertions. They are bundled together because they test the same selector with different data. We had problems with 5 test cases. The
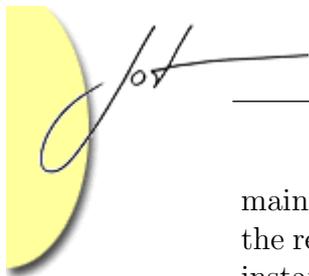
main problem was that the assertions use a variable that is neither the receiver nor the result nor a parameter. The meta-model cannot handle assertions with unknown instances. A solution could be to rewrite the implementation of the method under test so that the receiver has a reference to the missing variable. Then the receiver could be used in the assertions. Three test cases are simple. The test logic can be formulated as postcondition in the method. Therefore we converted the test cases to method examples with a checked method, which means the method has a postcondition.

6 CONCLUDING REMARKS

EGBROWSER is a proof-of-concept tool for systematically developing tests while maintaining their correspondance to the units under test. EGBROWSER is based on the EG testing meta-model which groups tests into categories that arise commonly in practice. A key feature of the meta-model is that tests are structured as *commands* which should yield an example instance of the class being tested. This feature not only allows tests to be composed by reusing examples generated by other tests, but it also encourages example-driven testing, and makes examples explicitly available for debugging and documentation.

Although EGBROWSER is just a prototype, an initial usability study suggests that several benefits arise from this approach. Many improvements still are needed to arrive at a truly usable tool. EGBROWSER should be extended to support the complete EG metamodel [Gae06]. For example, code generation for method suites is not yet supported. Coverage information is not generated, though it is foreseen by the EG metamodel. Whenever units under test are modified, the status of their tests should be changed. Invalidated tests could then be run automatically.

A tight integration of coverage information into the development environment as proposed by Reichhart *et al.* [RGD07] fits well with our approach: On the one hand commands can be used to create code coverage, on the other hand appropriate commands could be derived to exemplify a method at hand, which has not been implemented yet into the EGBROWSER.

Acknowledgments

REFERENCES

[Bec]   Kent Beck. Simple Smalltalk testing: With patterns. www.
           xprogramming.com/testfram.htm.

[BFJR98]   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.

[CK06]   Damien Cassou and Karsten Kuche. Dakar testing. European Smalltalk
           User Group Innovation Technology Award, September 2006. www.esug.
           org/data/ESUG2006/esug06_inno_aw_dakartesting.pdf.

[CNM83]   Stuart K. Card, Allen Newell, and Thomas P. Moran. *The Psychology
           of Human-Computer Interaction.* Lawrence Erlbaum Associates, Inc.,
           Mahwah, NJ, USA, 1983.

[Duc99]   Stéphane Ducasse. Evaluating message passing control techniques in
           Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–
           44, June 1999.

[EDH02]   James Trefil E. D. Hirsch, Joseph F. Kett. *The New Dictionary of
           Cultural Literacy.* Houghton Mifflin, 2002.

[Gae06]   Markus Gaelli. *Modeling Examples to Test and Understand Software.*
           PhD thesis, University of Berne, November 2006.

[GGN05]   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.

[GLN05]   Markus Gaelli, Michele Lanza, and Oscar Nierstrasz. Towards a tax-
           onomy of SUnit tests. In *Proceedings of 13th International Smalltalk
           Conference (ISC'03)*, September 2005.

[GLNW04]   Markus Gaelli, Michele Lanza, Oscar Nierstrasz, and Roel Wuyts. Or-
           dering broken unit tests for focused debugging. In *20th International
           Conference on Software Maintenance (ICSM 2004)*, pages 114–123,
           2004.

[GND04]   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.

[Mey97]   Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall,
           second edition, 1997.
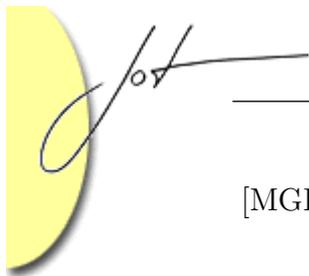
[MGL06]   Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile
           visualization framework. In *ACM Symposium on Software Visualization
           (SoftVis 2006)*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[RGD07]   Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. Rule-based as-
           sessment of test quality. In *Proceedings of TOOLS Europe 2007*, 2007.

[vG06]   Joris van Geet. Coevolution of software and tests: An initial assessment.
           Diploma Thesis, University of Antwerpen, July 2006.

[Wam06]   Rafael Wampfler. Eg – a meta-model and editor for unit tests. Master's
           thesis, University of Bern, November 2006.

[WD04]   Roel Wuyts and Stéphane Ducasse. Unanticipated integration of de-
           velopment tools using the classification model. *Journal of Computer
           Languages, Systems and Structures*, 30(1-2):63–77, 2004.

ABOUT THE AUTHORS

Markus Gaelli holds a PhD from the Software Composition Group
in Bern, is President of the German Squeak Association Squeak e.V.,
and works for SUISA in Zurich as a Software Engineer. See also
https://www.xing.com/profile/Markus_Gaelli.

Rafael Wampfler completed a Masters degree in Computer Science
at the University of Bern and is working as a Java developer at Glue
Software Engineering AG in Bern.

Oscar Nierstrasz is a Full Professor of Computer Science at
the University of Bern, Switzerland, where he has led the Soft-
ware Composition Group since its founding in 1994. See also
scg.iam.unibe.ch.