Eg – a Meta-Model and Editor for Unit Tests

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Rafael Wampfler

2006

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz

Institut für Informatik und angewandte Mathematik

The address of the author:

Rafael Wampfler
Im Holz 3
3309 Kernenried
Switzerland
wampfler@gmail.com

Abstract

In our life we often use examples to explain difficult topics. Examples help us to comprehend the problem. An example is easier to understand than an abstract description of the problem.

In software design the problems are complex and abstract. But examples are rarely used to explain a complicated situation.

We are using examples to document and explain software. Examples demonstrate the creation and behavior of an instance. They can be reused to compose new examples. Examples can be extended with assertions and become unit tests.

Because the link between test and method under test is often missing, we created a meta-model for tests. Our meta-model stores the objects, methods and parameters used for the tests. It can generate the source code of its tests which is human readable. First studies shows that most unit tests are method tests concerning only a single method call. The other tests can be refactored to method tests.

We built an editor for the meta-model to create examples and tests. The editor is integrated in the environment and lets the developer create new tests with a minimal effort.

Acknowledgments

First of all I want to thank Prof. Dr. Oscar Nierstrasz, head of the Software Composition Group, for giving me the possibility to do my master's thesis in his group. It was an interesting year and I learned a lot during this time.

Thanks also to Markus Gälli for supervising my work. We had many exciting discussions and he had plenty of good ideas for my thesis.

Then I thank all members of the SCG group, specially those who had time to participate in our usability experiment. And Michael Meyer for the encouragement during the work in the SCG student pool.

Many thanks also to my friends chrigu, höf, mike, möiri, plo, sam, ste and all I have forgotten for playing games, doing sportive activities or going on holidays.

Last I want to thank my family. They always supported me in my study. I know it was not an easy time, but this era will end now.

Rafael Wampfler,
November 2006

Contents

1. Introduction

Today's software projects are big with many developers working on them. It is nearly impossible for a single developer to understand a whole project. Therefore documenting code is an important part in object-oriented software design. The documentation helps other developers to maintain and extend the code. But it also helps developers to understand their own code better.

The different aspects of code need different *documentation*. We focus on object-oriented languages, where the source code offers comments and is divided in classes and methods. The following documentation techniques are available in most languages.

1.1. Documentation

Comments *Methods* are often documented with comments. The comments are written in the source code with a special comment syntax. Comments are written for humans and are ignored by the compiler. There is no improvement of the binary code by adding comments, but comments improve the readability of the code.

Classes are written in source code and can be documented like methods. The instance variables of the class should also be commented. The comment of a variable documents its role in the class. This is important for dynamic languages like Smalltalk.

Most developers are lazy. They write bad comments or none at all. Some developers believe that good code does not need any documentation.

But comments are an important part of software documentation. Some algorithms are hard to understand and it is difficult to follow the developer's thoughts without any comment.

Only a few tools encourage the developers in documenting their code. Different Java development environments can generate the skeleton for Javadoc comments with predefined parameter statements.

Examples *Instances* of classes are objects living in memory while executing an application. Objects can serve as an example instance of a class. An example object shows the *instantiation* of a class. Classes can have different ways to instantiate an object, but not all ways are meaningful in a given context. A living example object holds a set of

valid instance variables. This is interesting in a dynamic language where the type of a variable is not defined in the code but rather while executing it. Inspecting an example shows the type and value of the variables. Classes depends on other classes and are not self-contained. An example scenario is an example with multiple classes involved. An example scenario also shows the context in which the different classes and objects can interact.

Examples are rare. Some examples are written in the comments of classes and methods and can be executed from there. A developer can only detect these examples by browsing the source code of a class.

Some classes have methods that can be used as examples. For example executing Date today in Smalltalk returns an example instance of a date. In fact all class methods without parameters returning an instance of the class return an example for this class.

1.2. Understanding

Reading code If no documentation is available for the code the only way to understand it is by reading the source code. But reading source code of an object-oriented program can be difficult because the code is scattered into classes, extensions and packages. It is easier to understand a problem by examining an example of it.

Software visualization It is difficult to obtain an overview of a big software project in a limited time. Even if the developer reads all source code he does not necessarily understand the design of the system.

In this situation visualizations help. A picture is easier to understand than hundred pages of source code. In software visualization the information extracted from the code is bundled and displayed in different graphics. CodeCrawer [Lan] is a tool that can visualize software in various diagrams.

1.3. Testing

Unit testing Another way of documenting code is to write unit tests. Unit tests are small, automated software tests that can be run at any time. Tests assure that the code is working without problems in any stage of the development. Unit tests make use of examples. They can document how the application works, because the tests are running on real objects.

Because user testing is too time-consuming the developers are using a testing framework for automated unit testing. Some methods are tested with a testing framework. But most software is not 100% covered by tests. Writing unit tests is an exhausting work. With

test-driven design developers should first write the tests and later the implementation. In reality most developers test their code after implementing it.

Navigation between test and implementation Most tests are written as source code. The method under test can be found in the test, but is not denoted as such. Often the interesting method is not the method that is directly called, but rather called somewhere in the call tree. There is no control if executing a test calls the methods that should be tested. To link the test and the method under test dynamic analysis is needed. Dynamic analysis is more time expensive because the execution of the code is modified. With both ways it is difficult to determine which method is tested rather than used to set up a test scenario.

Creating tests A unit test is created by writing down its source code. The syntax of the testing frameworks are loose and permit the developer to assemble different unit tests. But the developer is not supported by the tools. He can write any test for a method. But not all tests have useful assertions checking the needed functionality.

Test reuse The tests are not reused. The tests cannot be reused because they do not have a return value. But most methods return a meaningful object. This object can serve as an example instance or can be reused for another test.

User interface Good user interfaces for unit tests are rare. There are interfaces for running and debugging tests, but not to create them! The interfaces cannot display the connection between the method and their tests. Having no good tools limits the developer's workflow [Csi91].

User interface testing Testing user interfaces is a hard task. Unit testing is not possible because the interface needs user interaction with a mouse and keyboard. There are tools that can simulate the mouse motion and clicks of a user. Abbot[1] is an open source testing framework for Java user interfaces.

In reality user interfaces are tested by clicking through them. A developer knows how to use the interface and where he can click. But an unfamiliar user clicks everywhere he wants and that can be wrong in some contexts. Hence the best tests are done with real users who do not understand the application. This means user interface testing is a time-consuming task if it is done professionally.

[1] `http://abbot.sourceforge.net`

1.4. Our Approach

Our goal is to test and understand software. *Examples* are our metaphor. We want running examples that are changeable. Tests need to be flexible because requirements can change late in the development process. We use examples to demonstrate and document source code. An example of a class is a parameterless class method returning an example instance of the class. Examples return a value and can be reused to compose other examples.

Benefits of examples include:

- An example shows the way to initiate its class.
- It is an example instance of its class.
- It is a real object we can modify and play with.
- It can run at any time.
- It returns a reusable example object. The return value can be reused to compose another example.
- It can be extended by assertions becoming a unit test.

Because test and method under test are not linked, we implemented a meta-model for examples and tests. The model knows the connection between a method and its test. The meta-model can run the examples and supports return values. Therefore the examples can be reused to compose new examples. The meta-model has different types and can cover a large section of unit testing. The model can export the examples as human readable source code.

Examples do not substitute other documentations like comments, examples are just an additional way to document code. Examples are more valuable than comments because they also assure that the code is running.

1.5. Structure of the Thesis

Chapter 2 **(p.7)** gives an overview of unit testing frameworks available today. All frameworks have some limitations that are discussed here. We try to solve the problems stated in this chapter with our meta-model.

Chapter 3 **(p.17)** describes the first part of our solution for documenting and testing code. We implemented a meta-model for commands in Smalltalk. The chapter explains the different types of commands and how they work together. This chapter includes technical details about the implementation of the model.

Chapter 4 (p.29) is about the second part of our solution, the test editor. The editor is built for the meta-model from chapter 3 (p.17). An example application of a video store demonstrates the usage of the editor.

Chapter 5 (p.43) compares our solution with some tools from chapter 2 (p.7). The tools are compared with a metric for measuring user interfaces. We also arranged a small usability study where we compared the time to accomplish different testing tasks and the learning process with the editor.

Chapter 6 (p.59) discusses the conclusion of our solution. Finally we list the future work and some ideas to improve the meta-model and test editor.

2. State of the Art

This chapter describes the most often used frameworks and techniques for testing object-oriented software. We focus on tools implemented in Smalltalk and Java.

There are many approaches for unit testing, but none solves all problems from chapter 1 (p.1). All frameworks have some limitations. We focus on unit testing, thus we evaluate XUnit, a common testing framework.

2.1. XUnit

Unit tests are very popular in software design. XUnit is a widely used testing framework. The XUnit framework has been adopted to many programming languages. The tests can be re-run at any time without user interaction. The tests are repeatable.

XUnit tests are built with conventions. XUnit tests need to be in a test class inheriting from a test superclass. The test method name starts with `test`. This convention is used by the test runner for gathering all tests.

Modern implementations of a testing framework use annotations instead of naming conventions. Thus the framework permits more flexibility for testing. The class does not need to extend a test superclass and a real test hierarchy can be realized.

2.1.1. SUnit

SUnit[1] is the "mother of unit test frameworks" [Bec]. SUnit is suitable for all kind of applications. But SUnit can be slow on big test suites because every test case needs to create a test scenario to run. There is no reuse of the test objects, because a test does not return a result value.

The syntax of a test is rather free. It is not always clear what the purpose of the test is and which methods should be tested. Often the methods are called in the assertions. Multiple assertions can test more than one method, so a test does not focus on a single method. Therefore the test case is rather a test suite for different methods.

[1] `http://sunit.sourceforge.net`

Figure 2.1 (p.8) shows the test runner of SUnit in VisualWorks[2]. The interface is limited, the test runner can only run and debug tests. The interface cannot display the source of the test or the methods accessed by the test.

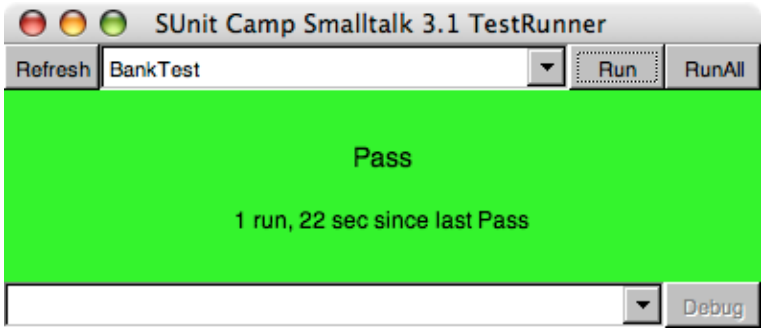

Figure 2.1.: Test runner of SUnit

SUnit example

The following code written in Smalltalk defines a test class using SUnit. The test class extends TestCase and has an instance variable named account.

```
Smalltalk defineClass: #BankTest
    superclass: #{XProgramming.SUnit.TestCase}
    indexedType: #none
    private: false
    instanceVariableNames: 'account '
    classInstanceVariableNames: ''
    imports: ''
    category: 'Eg-Bank'
```

A setup method can be used to create a test scenario. The setup method is used for every test method and is called before each test case run. SUnit also supports a tear down method.

In this example the setup defines the instance variable of the class. Because we use the variable only in one test, we could initialize the variable there and do not necessarily need a setup method.

```
BankTest ≫ setUp
    account := Account new
```

[2] http://smalltalk.cincom.com

The fundamental test is in the test case. Any name starting with test can be used as method name. To highlight the selector #deposit:, we choose testDeposit as the test's name.

The test is adding an amount of money to the empty account. The assertion ensures that the balance has increased. Each test case usually has at least one assertion.

```
BankTest ≫ testDeposit
    |balanceBefore|
    balanceBefore := account balance.
    account deposit: 100.
    self assert: account balance = (balanceBefore + 100)
```

A tear down method is not needed in this example, the account gets garbage collected. A tear down method is executed after the tests, for example to close a database connection.

A test runner collects all test methods and checks the assertions by running them. A test passes if all assertions hold. If at least one assertion is wrong the test fails. It can also raise an error if a syntax error happens.

Refactoring Browser SUnit Extensions

Refactoring Browser SUnit Extensions[3] add a small test runner on the bottom of the browser when browsing a subclass of TestCase. The test runner looks similar to figure 2.2 (p.10), except the button order is different. The test runner can run SUnit tests without opening a test runner and selecting the test. It simplifies running a test while browsing, instead of opening an extra test runner window. The connection between test and implementation is still not visible.

SUnitToo

SUnitToo[4] started as a patch for SUnit because of a memory problem with a large test. Today SUnitToo is a forked version of SUnit. It has a simpler design and performance improvements over SUnit. SUnitToo has its own model, but uses the same syntax and interfaces like SUnit.

The test runner of SUnitToo is an adaption of the Refactoring Browser SUnit Extensions and is integrated in the browser. Figure 2.2 (p.10) shows the different visual feedbacks of the test result: The big status bar on the bottom left turned green, the test tube before the class is filled green and the test method is ticked green. The test runner also supports profiling and debugging.

[3] http://www.cincomsmalltalk.com/publicRepository/RBSUnitExtensions.html
[4] http://www.cincomsmalltalk.com/userblogs/travis/blogView?entry=3278236086

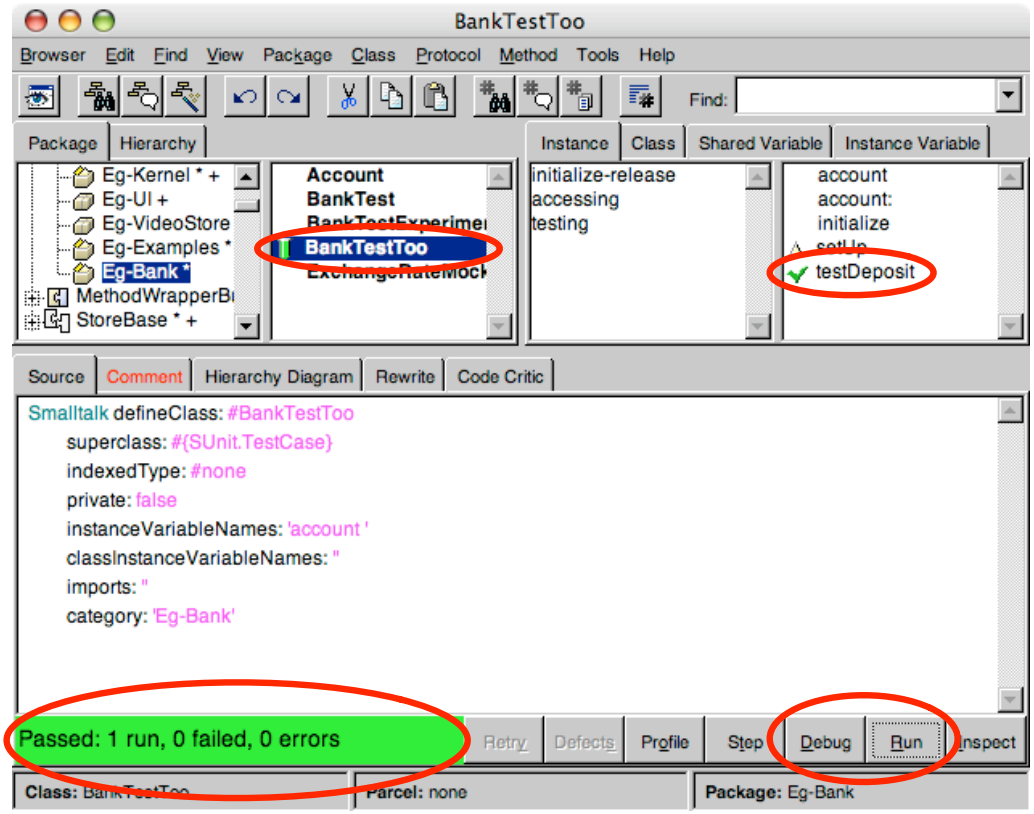

Figure 2.2.: Test runner of SUnitToo

BrowseUnit

BrowseUnit by Romain Robbes [Rob04] provides integration for SUnit and SLint in the various browsers, including the Star Browser and the Refactoring Browser. It is implemented in Squeak[5].

BrowseUnit can generate the test class, the method under test, the setup and tear down methods and the test method for SUnit. BrowseUnit allows you to browse between the implementing method and the test. To browse the tests of a method, BrowseUnit examines all compiled tests and collects the calls of the method. Likewise to browse all methods of a test, BrowseUnit collects the method sends by the test.

Figure 2.3 (p.11) shows the menu of BrowseUnit.

[5] `http://squeak.org`

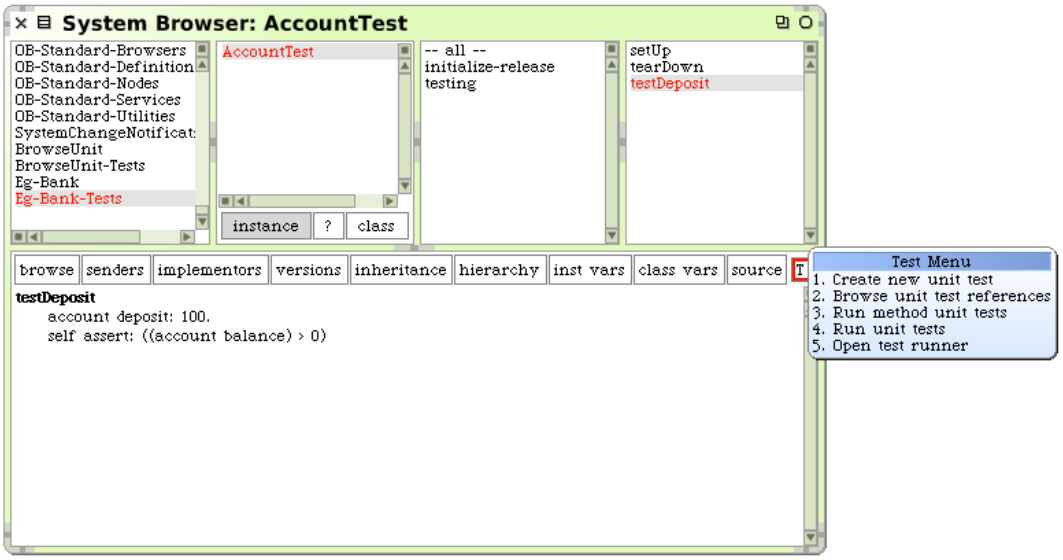

Figure 2.3.: BrowseUnit

2.1.2. JUnit 4

JUnit[6] is a rewrite of SUnit in Java by Erich Gamma and Kent Beck, the inventor of SUnit. It is the de facto standard for unit testing in Java.

JUnit 4 has some new features. The tests are no longer denoted by a naming convention, but are denoted with annotations. Annotations are supported in Java since J2SE 5.0 (Java 1.5)[7].

The same bank example implemented in SUnit above has the following form in JUnit 4:

```
package egBank;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import org.junit.Before;
import org.junit.Test;

public class BankTestJUnit4 {
    Account account;

    @Before
    public void setUp() throws Exception {
```

[6] http://www.junit.org
[7] http://java.sun.com/j2se/1.5.0/

```
        account = new Account();
    }

    @Test
    public void testDeposit() {
        int balanceBefore = account.getBalance();
        account.deposit(100);
        assertEquals(account.getBalance(), balanceBefore + 100);
    }

    @Test(expected = NotEnoughMoneyException.class)
    public void testWithdraw() {
        account.deposit(100);
        account.withdraw(150);
        assertTrue(account.getBalance() > 0);
    }
}
```

The static import is needed because Java does not allow you to extend the class Object with new methods. With the static import, assertTrue can be used as if it had been included in Object. The test method is denoted with the @Test annotation. Any name can be chosen as test method name. The setup method is denoted by the annotation @Before and not by its name. An @After method defines a tear down method [Wes05].

With JUnit 4 any method can act as a test and the test class does not need to extend a test case class. But a test method still cannot return a value. Exceptions can be passed as parameters to the annotation: @Test(expected = Error.class). A time consuming test can have a time limit with @Test(timeout = 100). A test suite is no longer required. The test runner is text based only. The developer environments are responsible for graphical test runner. JUnit 4 is well integrated in Eclipse[8].

2.1.3. TestNG

TestNG[9] is another unit testing framework for Java. TestNG uses annotations, but was developed before JUnit 4. It can also denote the tests with Javadoc instead of annotations for Java 1.4 compatibility.

JUnit tries to isolate the tests so they can be run individually. TestNG has the approach of dependency testing. A test can depend on a method or a group of other tests. If a required method fails, the test is skipped and is not marked as failure. TestNG can re-run the skipped and failed tests only if desired.

[8] http://www.eclipse.org/jdt/
[9] http://testng.org

Another feature missing in JUnit is parametric testing. The test method can take parameters. The same test method can be reused with different data. The parameters are taken from an XML file or a data provider class. In JUnit we would need to write the tests for each data set. TestNG can group tests in categories.

TestNG is more flexible and suitable for large projects. It uses Ant[10] with an XML configuration file. But there is also a nice Eclipse plugin with a graphical test runner.

The following test is implemented in TestNG. It has the same test cases as the SUnit and JUnit implementation. The code looks similar to the JUnit code. The `@Test` annotation takes different parameters than the JUnit version.

```
package egBank;

import static org.testng.Assert.assertEquals;
import static org.testng.Assert.assertTrue;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;

public class BankTestTestNG {

    Account account;

    @BeforeClass
    public void setUp() {
        account = new Account();
    }

    @Test(groups = {"Bank"})
    public void testDeposit() {
        int balanceBefore = account.getBalance();
        account.deposit(100);
        assertEquals(account.getBalance(), balanceBefore + 100);
    }

    @Test(groups = {"Bank"}, expectedExceptions = NotEnoughMoneyException.class)
    public void testWithdraw() {
        account.deposit(100);
        account.withdraw(150);
        assertTrue(account.getBalance() > 0);
    }
}
```

[10] `http://ant.apache.org`

2.1.4. JTiger

JTiger[11] is yet another testing framework for Java. It uses annotations similar to JUnit 4. Like TestNG it is suitable for large test scenarios and uses Ant as its test runner. JTiger does not have a user interface to run the tests. JTiger can report the test results in plain text, HTML or XML. A test method can have six different exit states:

- Success
- Ignored (Annotated)
- Ignored (Cannot Invoke)
- Failure (Set Up)
- Failure
- Failure (Tear Down)

JTiger is the only framework that supports return values in tests. Because JTiger tests can have a return value, the withdraw test is implemented twice. The first is done like the other frameworks without reusing a test. The second implementation reuses the result from the deposit test.

```
package egBank;

import static org.jtiger.assertion.Basic.assertTrue;
import static org.jtiger.assertion.Basic.assertEqual;
import org.jtiger.framework.Category;
import org.jtiger.framework.ExpectException;
import org.jtiger.framework.SetUp;
import org.jtiger.framework.Test;

public class BankTestJTiger {
    Account account;

    @SetUp
    public void setUp() {
        account = new Account();
    }

    @Test
    @Category("Bank")
    public Account testDeposit() {
        int balanceBefore = account.getBalance();
        account.deposit(100);
```

[11] http://jtiger.org

```
        assertEqual(account.getBalance(), balanceBefore + 100);
        return account;
    }

    @Test
    @Category("Bank")
    @ExpectException(NotEnoughMoneyException.class)
    public void testWithdraw() {
        account.deposit(100);
        account.withdraw(150);
        assertTrue(account.getBalance() > 0);
    }

    @Test
    @Category("Bank")
    @ExpectException(NotEnoughMoneyException.class)
    public void testWithdrawReuse() {
        Account fullAccount = this.testDeposit();
        fullAccount.withdraw(150);
        assertTrue(account.getBalance() > 0);
    }
}
```

2.2. Problems

The main problem with testing frameworks is that the test does not know its method under test. And a method does not know which test is accessing it. Because the link between test and method is inexistent they never know exactly about each other. The information can be obtained by searching the senders and implementors (static analysis) or by tracing the test on run-time (dynamic analysis). Static analysis can be inaccurate, dynamic analysis can be slow.

Another problem is that a test has no return value. Current test scenarios have a flat hierarchy. Each test can run stand-alone. But most test methods could return a useful value that can be reused to compose another tests and test scenarios.

The testing frameworks have most features that are needed, but the user interfaces are limited. There are not many tools beside the test runners. The user has to type in the code of a test and is not supported by a tool.

3. Meta-Model

This chapter describes the meta-model we implemented in Smalltalk. The meta-model is based on commands and has different command types. The model is processable by a programming language and is also human readable.

There are three main problems with unit testing:

- Methods do not know their tests and vice versa. It is not evident which method is tested by which test.
- Tests are not reused.
- The user interfaces do not support a developer in writing unit tests.

The first two problems we solve with our meta-model. The meta-model holds an abstract description of the test. It helps to connect the methods and their tests. The meta-model of a unit test knows exactly which method is tested by this test [GND04]. Each test has a return value, so the tests are reusable [GGN05].

Our user interface encourages the developer in writing tests. The process of creating a test is simplified. The developer should type as few as possible. Existing tests can be reused in new tests and composed with drag and drop. The navigation between test and implementation can be improved in both directions.

3.1. Commands

We built a meta-model with the concept of commands [GLN05]. A command is an object that understand the message run. A command has a *result*. By running the command, the return value of the method is stored in the result temporary variable. Because commands have a return value, they can be used to compose other commands. Commands belong to a *package* and have a *name*.

The commands can be divided into two groups: method commands and multiple method commands. Method commands are using a single method, multiple method commands are using more than one method. The most important classes are shown in figure 3.1 (p.18).

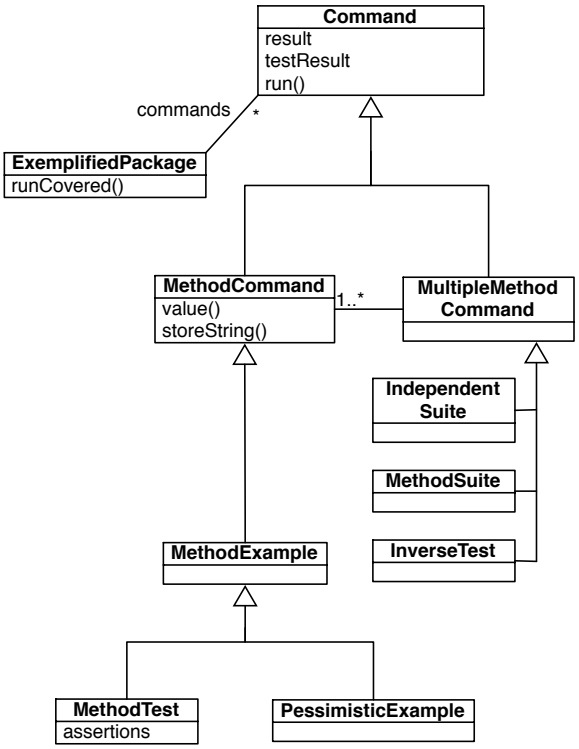

Figure 3.1.: Meta-model command types

3.2. Persistence of Commands

There are two approaches to write a command. Commands can be created with the meta-model's *factory methods* or by typing the *source code* of a command. The meta-model code is suitable for business methods, scripts and the user interface. But humans can comprehend a command faster if it is written as source code. It is possible to write a command in source code if the correct syntax is used. The commands also need a source representation if a *versioning system* is used. All changes are stored as source code in the image.

Commands are stored as source code. The meta-model can generate the source code of a command. This code is compiled as a *class method* of the result without parameters. The example methods can be run at any time because they do not have any parameters. Since the command is stored on the class side, the name of the command needs to be unique for the result class. The compiled method is an *example* of the implementing class.

The code is annotated with the <egClass:method:> pragma. The pragma holds the command type and the method under test. The command type is used to parse the command

to the meta-model. Guessing the type from the code is ambiguous. The method under test is annotated for better comprehension. It is not used by EG because we assume that the last method call is the method under test. The compiled example methods can be used with another tool; the code does not have any EG specific syntax.

3.3. Method Commands

A method command focuses on a *single method call* [GND04]. A method call is a selector and its parameters sent to a receiver object. A method command knows the *receiver*, *selector* and *parameters*. The receiver is the object under test. The message is composed of a selector and parameters and is sent to the receiver. The result is the return value of the message.

First studies [GLN05] indicate that method commands are the most used form of commands or are decomposable into method commands. Most large test scenarios can be decomposed to method commands. A scenario is a sequence of method calls. Each method call can be transformed to a method command.

Persistence of meta-model variable values Figure 3.2 (p.19) shows the meta-model of a method command. All objects used as receiver or parameters need to have a source code representation because the command is stored as code. The message storeString is sent to the object to get the source code. The receiver is modeled as a block. Because commands are composable, method commands can also be used as input for other commands. The commands are cascaded like a pipe in Unix.

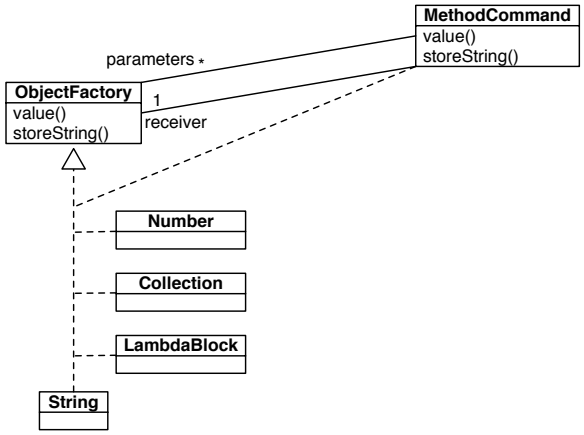

Figure 3.2.: Meta-model of method commands

3.3.1. Method Examples

A method example is the simplest form of a method command. A method example shows the usage of a selector. The result is also an example instance of the result class.

There are two ways to create a method example. A method example can be built with the factory methods of the meta-model:

```
Eg.MethodExample new
    forPackage: 'Eg-Bank'
    receiver: [Account new]
    selector: #deposit:
    parameters: (OrderedCollection with: 100)
```

The selector #deposit: needs one parameter.

The second possibility is to write the source code directly:

```
Account class ≫ exampleDeposit
    <egClass: #'Eg.MethodExample' method: #deposit: >
    | aReceiver aResult |
    aReceiver := Account new.
    aResult := aReceiver deposit: 100.
    ^aResult
```

The result of this method example is the account. Hence the method example is stored on Account class as example of an object returning an account. If no example name is provided, the name is composed from an example prefix and the selector.

Method examples can also act as unit tests. The simplest form of a test is a *checked method example*. A checked method is a method that has a *postcondition*. The postcondition assures that the method is working for every use of it. Hence all calls of a checked method are tests. A method example calling a checked method is a test without an assertion, because the test logic is done in the implementation of the method already.

3.3.2. Method Tests

A method example can be extended with one or more *assertions* and becomes a method test. Method tests can be used as *unit tests*. The assertions are blocks expecting a boolean value. The assertion can be used to define a postcondition. If an assertion fails on running the test, it raises an exception.

```
Eg.MethodTest new
    forPackage: 'Eg-Bank'
    receiver: [Account new]
```

```
    selector: #deposit:
    parameters: (OrderedCollection with: 100)
    assertions: (OrderedCollection with: [aResult balance = 100])
```

is equal to

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

Persistence of assertions Assertions can access the temporary variables aReceiver, someParameters and aResult. Assertions written as strings can use all variables. An assertion with two variables can look like this: `'(aReceiver includes: (someParameters at: 1))'`. Simple assertions can be modeled also with blocks. Because the VisualWorks[1] block decompiler loses the variable names, using different variables is ambiguous. Blocks with one variable replaces the name with aResult, because this is the most used variable.

3.3.3. Pessimistic Examples

A pessimistic example is a command that *should raise an exception*. If no exception is caught, it raises a new exception. A pessimistic example shows what happens in case of a misuse of the selector or an invalid receiver. A method under test fails if the preconditions do not hold. There is no result because the message should fail and does not have a return value. The example is compiled to the receiver because the result is not an example for this test. A pessimistic example does not have assertions.

Because the bank does not give credits on the accounts, the following example is a pessimistic example:

```
Eg.PessimisticExample new
    forPackage: 'Eg-Bank'
    receiver: [Account withBalance: 50]
    selector: #withdraw:
    parameters: (OrderedCollection with: 100)
```

is equal to

[1] `http://smalltalk.cincom.com`

```
Account class ≫ exampleWithdraw
    <egClass: #'Eg.PessimisticExample' method: #withdraw: >
    | aReceiver aResult |
    aReceiver := Account withBalance: 50.
    aResult := [aReceiver withdraw: 100 ] shouldFail.
    ^aReceiver
```

The pessimistic message is inside a block that should fail. The withdraw: message raises an exception if the amount is negative. This exception is caught by the shouldFail message.

```
BlockClosure ≫ shouldFail
    [self value] on: Error do: [:err | ^true].
    self halt.
    ^false
```

3.3.4. Cascaded Commands

A method command with another command as receiver or parameter is a cascaded command. Additionally to the features of a method command a cascaded command can build the expanded source code.

If a cascaded command uses commands as receiver or parameters, the source code of these commands gets inlined in the expanded source code. The expanded code is not compiled as example because there is no use for this and the duplicated code is redundant. Parsing an expanded code would be much more complicated.

The expanded code is used for a better understanding of a command because the cascaded command is visible and does not need to be browsed in another window.

The following example has a command as receiver. The receiver command can also be a block. The command is expanded in the source code.

```
Eg.MethodTest new
    forPackage: 'Eg-Bank'
    receiver: [Account exampleDeposit]
    selector: #withdraw:
    parameters: (OrderedCollection with: 20)
    assertions: (OrderedCollection with: [aResult balance = 80])
```

The command used as receiver has the following code:

```
Account class ≫ exampleDeposit
    <egClass: #'Eg.MethodTest' method: #deposit: >
    | aReceiver aResult |
    aReceiver := Account new.
```

```
    aResult := aReceiver deposit: 100.
    self assert: aResult balance = 100.
    ^aReceiver
```

The default source code that is compiled as example is

```
Account class ≫ exampleWithdraw
    <egClass: #'Eg.MethodTest' method: #withdraw: >
    | aReceiver aResult |
    aReceiver := Account exampleDeposit.
    aResult := aReceiver withdraw: 20.
    self assert: aResult balance = 80.
    ^aReceiver
```

The expanded source code that can be displayed in the browser is

```
Account class ≫ exampleWithdraw
    <egClass: #'Eg.MethodTest' method: #withdraw: >
    | aReceiver2 aResult aResult2 |
    aReceiver2 := Account new.
    aResult2 := aReceiver2 deposit: 100.
    self assert: aResult2 balance = 100.
    aResult := aResult2 withdraw: 20.
    self assert: aResult balance = 80.
    ^aReceiver
```

All temporary variable names are replaced to use the values from the cascaded command. The variable aResult2 is the result of the first command. The value is reused as receiver of the second command. Note that the result of the first command needs to be the same type as the receiver of the second command, else the method under test is not defined.

3.4. Multiple Method Commands

Multiple method commands are collections of one or more commands. Multiple method commands have a result, but they are rarely used. Some multiple method commands return the result of the last command or a boolean. Not all multiple method commands can generate their source code, because the meta-model constructor code is already human readable. Generating another source code representation does not clarify the command. The example of section 3.4.1 (p.24) has only a single source code representation.

3.4.1. Independent Suites

An independent suite is a collection of independent commands. A suite can run all including commands. It returns the last command as result, therefore it is not adept to use it for another test. An independent suite is equivalent to a test suite in SUnit. A suite has the following source code representation.

```
Eg.IndependentSuite class ≫ exampleBankSuite
    | suite |
    suite := Eg.IndependentSuite new.
    suite packageName: 'Eg-Bank'.
    suite
        addCommand: Eg.MethodTest testAccountDeposit;
        addCommand: Eg.MethodExample exampleAccountDeposit.
    ^suite
```

3.4.2. Multi Facet Command Suites

A multi facet command suite consists of different commands that share the same setup method. In our meta-model the receiver can be another command instead of using a setup method. Therefore a multi facet command suite consists of individual commands that can reuse the same receiver as setup method.

3.4.3. Method Command Suites

A method command suite consists of different commands focusing on the same selector. The method command suite collects all available commands in the system with the given selector. The method command suite is a generated view. Therefore it does not have a source code representation and is not compiled as an example. A method command suite is used to compare the different receivers and parameters of a selector. The following method return the values from table 3.1 (p.24):

```
Eg.MethodCommandSuite new forPackage: 'Eg-Bank' class: Account selector: #deposit:
```

Table 3.1.: Method command suite of Account ≫ deposit:

Name	Receiver	Selector	Parameters	Result
exampleDeposit	Account new	#deposit:	100	Account with 100
exampleDeposit2	Account with 100	#deposit:	200	Account with 300

3.4.4. Inverse Tests

An inverse test is similar to a method test, except it has a second selector which is the *inverse selector* of the first one. Inverse tests can be used to test two inverse functions. Mathematically: $f^{-1}(f(x)) = x$, where f^{-1} is the inverse function of f. By running the inverse test the first selector is sent to the receiver. The inverse selector then is sent to the result of the first selector. The inverse selector accepts the same parameters as the selector. Finally a predefined assertion checks if the result is equal to the receiver. The receiver objects needs to implement the equal operator (object = anotherObject), because it is used in the assertion.

The following inverse test deposits an amount on an account and withdraws it again:

```
Eg.InverseTest new
    forPackage: 'Eg-Bank'
    receiver: [Account new]
    selector: #deposit:
    parameter: 20
    inverseSelector: #withdraw:
```

is equal to

```
Account class ≫ exampleDepositWithdraw
    <egClass: #'Eg.InverseTest' method: #deposit: >
    | aReceiver aResult aFinalResult |
    aReceiver := Account new.
    aResult := aReceiver deposit: 20.
    aFinalResult := aResult withdraw: 20.
    self assert: [ aReceiver = aFinalResult ].
    ^aFinalResult
```

We can also use inverse tests to test Eg itself. The example in figure 3.3 (p.26) shows an inverse test with the inverse selectors parse and buildCode.

This tests the compiler and code generator. The receiver is a source string. The generated source code is equal to the starting code.

```
Eg.InverseTest new
    forPackage: 'Eg-Examples'
    receiver:
        'exampleAdd
            <egClass: #'Eg.MethodTest' method: #add: >
            | aReceiver aResult |
            aReceiver := Set new.
            aResult := aReceiver add: 10.
            self assert: [aResult = 10].
```

```
        self assert: [aReceiver includes: 10].
        ^aResult'
    selector: #parse
    inverseSelector: #buildCode
```

is equal to

```
ByteString class ≫ exampleParseBuildcode
    <egClass: #'Eg.InverseTest' method: #parse >
    | aReceiver aResult aFinalResult |
    aReceiver := 'exampleAdd
        <egClass: #'Eg.MethodTest' method: #add: >
        | aReceiver aResult |
        aReceiver := Set new.
        aResult := aReceiver add: 10.
        self assert: [aResult = 10].
        self assert: [aReceiver includes: 10].
        ^aResult'.
    aResult := aReceiver parse.
    aFinalResult := aResult buildCode.
    self assert: [ aReceiver = aFinalResult ].
    ^aFinalResult
```

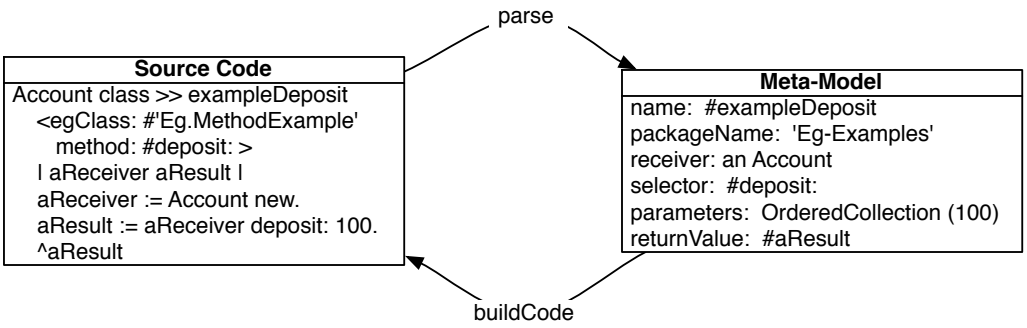

Figure 3.3.: Inverse test

3.5. Emerged Meta-Model

The meta-model is displayed in figure 3.4 (p.27). The model consists of the different command classes, the object factory and the runtime informations. The object factory interface ensures that the meta-model variable values can be compiled as source code.

During the execution of a command, additional information can be gathered. A method command calls one method. So the method command is 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.

The meta-model is built with pure Smalltalk syntax and does not need any external libraries. The generated commands can be used without the meta-model or with other implementations.

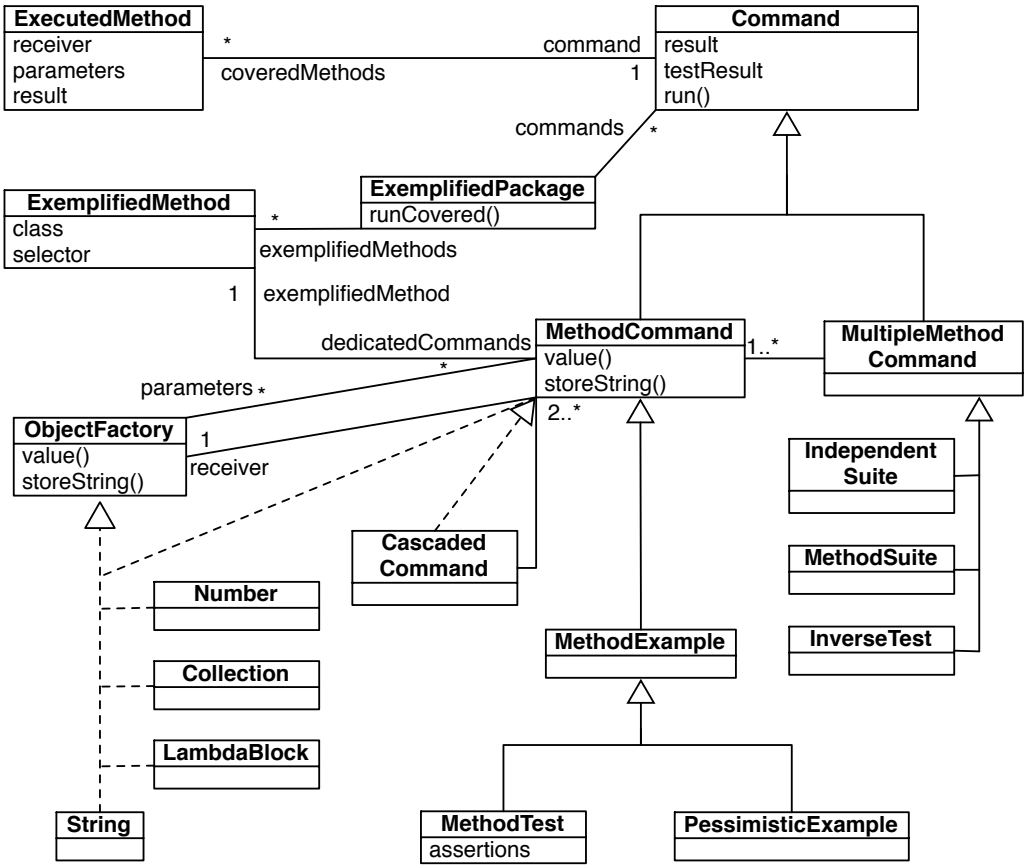

Figure 3.4.: Meta-model of commands

3.6. Prepared Method Implementation

Method commands can generate the method implementation of their selector. If the selector is missing, this code is compiled to the class of the receiver object. The vari-

able names are suggested from the type of the parameter given by the command. The variable names reflect the type of the parameters and should be a hint for the developer. The developer is encouraged to change the names according to the role of the parameters.

With this automatic implementation, example-driven design is possible: A developer can do a test for a yet missing selector. During the creation of a method command the meta-model checks if the selector is present. If the selector is missing, it is compiled to the class of the command receiver. Creating a command runs it, so a debugger opens on the new method. The developer has to implement the body of the method and can resume the command.

The following example code is generated if a command uses the selector #withdraw: with the parameter 50.

```
Account ≫ withdraw: aSmallInteger
    self halt.
```

3.7. Prepared Class Implementation

When developing in a genuine test-driven design, the test is created before the class under test even exists. In the meta-model this means the command is built before the class. The receiver of a command is referencing the not yet existing class.

The meta-model can generate the class skeleton for the referenced class under test. The new class does not have any instance variables or methods.

4. User Interface

A good user interface is important for the acceptance of a software project. But it is difficult to build a fully functional user interface that is easy to use. Some users prefer rich interfaces with menus, buttons and a lot of mouse work. Others favor slim, text-based interfaces. A tradeoff is needed to satisfy a large user group.

This chapter shows how to use the test interface we built for the meta-model. The meta-model is modular and can also be used without interface or with another interface.

4.1. Problem

There are only few user interfaces for unit tests available today. SUnit comes with a simple test runner as seen in figure 2.1 (p.8). The developer has to open an extra test runner window. This is a context switch and slows down the workflow. Other tools are trying to integrate a testing environment into the class browser. But most tools (*e.g.,* XUnit) cannot browse from a method to its test and vice versa, because a meta-model is missing. The developer has no overview which methods are tested and which tests are covering a method.

Some tools can run and debug tests, but none can create tests. Unit tests are written as source code and the developer has to know the correct syntax of a test. Beginners cannot use the full functionality of the testing framework, because they do not know the syntax.

4.2. Interface Requirements

A new test editor should achieve the following requirements:

- Easy to use
- Minimal mouse movement and keyboard typing
- Support for drag & drop
- Interface should propose valid input values
- Should generate source code

- Support for reusable examples and tests
- Link method implementation and test without context switch
- Graphical feedback of covered methods
- Integrated test runner

Our test browser aims to improve the process of creating a test. The interface can propose useful values for the input fields. Because tests have a result, they can be reused. New tests can be composed by drag and drop. The browser also shows which method is covered by a test. The meta-model knows exactly which methods are called by a certain test. But the meta-model still has a source representation of the test. The source code is human readable and can be reified into objects according to the meta-model.

4.2.1. Star Browser

Star Browser[1] by Roel Wuyts [WD03] is a Smalltalk class browser. The goal of Star Browser is the ability to classify anything while browsing the system. Figure 4.1 (p.31) shows a Star Browser while browsing a class.

The classified items are arranged in a tree view widget. Items can have children. New items can be added by drag and drop. The underlying classification model can collect all objects matching a given pattern. Each item has a context menu and can have a different browser view opened in the right pane when it is selected.

4.3. EgBrowser

The EgBrowser is a modification of the Star Browser. We choose Star Browser for its tree view and drag and drop support. The classification tree view from Star Browser is used to classify and display the various commands of the meta-model. The visitor model of Star Browser is extended to handle commands from the meta-model.

Figure 4.2 (p.32) shows the EgBrowser in action. The root item in EgBrowser holds the packages. Each package includes commands and can run them from the context menu. The package icon turns green if all commands run without failures or else red. Clicking on the package icon opens an inspector.

Inside the package the method commands are grouped by the method under test. Selecting a method reference icon opens the browser on the implementation of this method.

Inside the method reference are the commands that focus on this method. Commands are displayed as items with their variables as sub-items. While a command runs, the

[1] `http://homepages.ulb.ac.be/~rowuyts/StarBrowser/`

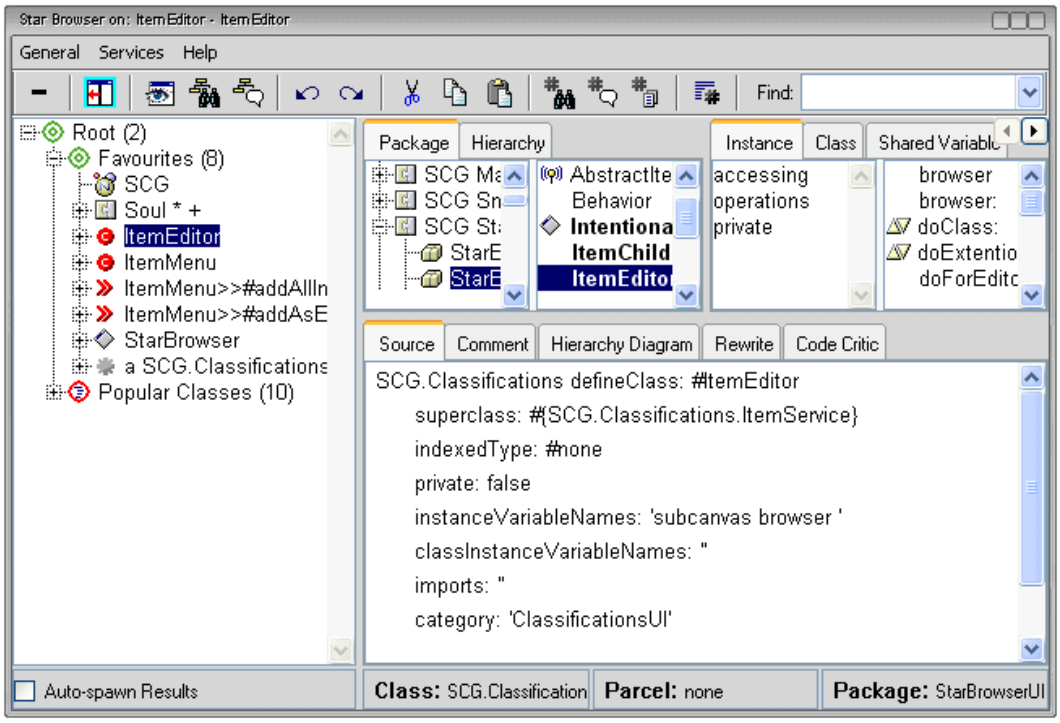

Figure 4.1.: Star Browser

icon color can change: Green means the command runs without problems, else the icon becomes red.

The icon letter reflects the command type. The following icons are available:

- **E**: Method Example
- **T**: Method Test
- **I**: Inverse Test
- **S**: Suite, any other multiple method command

Clicking on a command opens the browser on the compiled command source. The variables in the sub-items can be changed by drag and drop. The command is recompiled if a change occurs.

If a command is available for a visible selector in the class browser, its type icon is displayed before the selector name. If multiple commands exists for a selector, the icon shows a **M**.

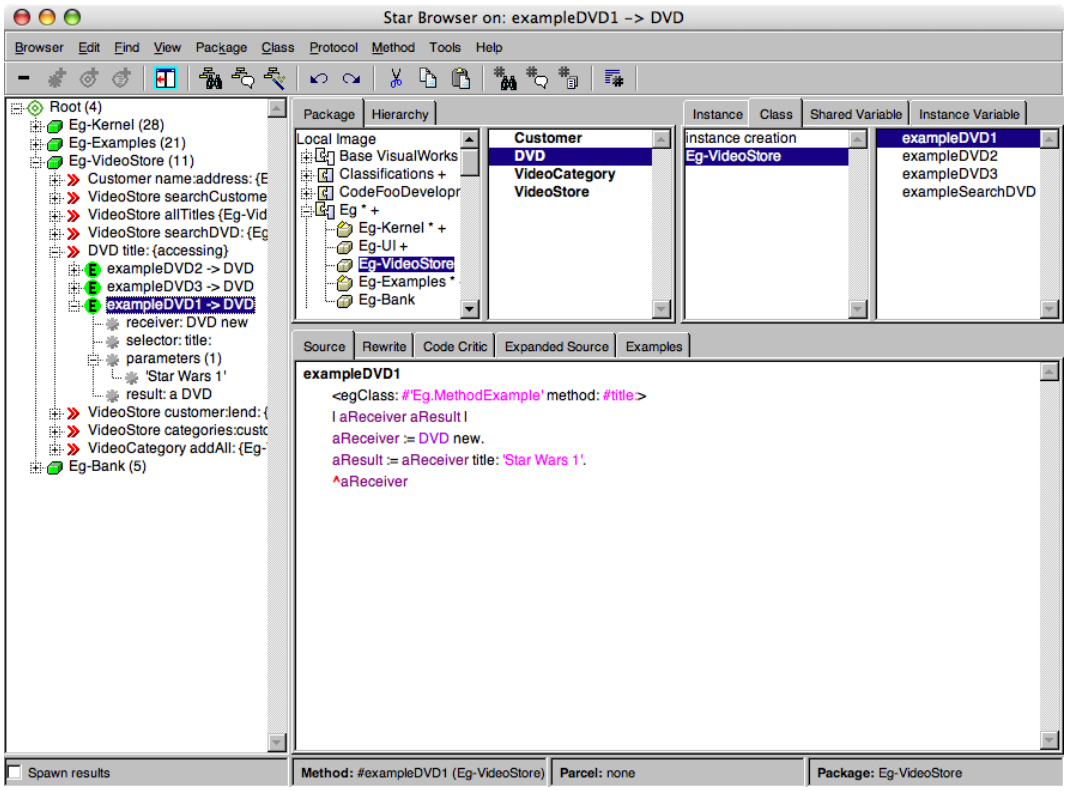

Figure 4.2.: EgBrowser

4.4. Video Store – an Example Application

The video store is a simple application that demonstrates the use of the EgBrowser. The application is built test-driven. A test scenario is created simultaneously. This means by creating examples and tests, the application is implemented and tested along the way.

The video store application consists of the classes Customer, DVD, VideoCategory and Videostore. A customer has a name and an address. The name is the primary key used by the video store. The DVD has a title, date and description. A video category has a name and a collection of DVDs. The video store holds all together. It has a collection of customers and a collection of video categories. The video store can lend a DVD to a customer. Items and customers can be searched by name. The video store has various business methods.

4.4.1. Creating Examples

The classes Customer, DVD, VideoCategory and Videostore are created with the wizard from the class browser or the traditional way by typing the source code into the class browser.

Another way is to let the Eg meta-model generate the class. If the receiver of a command is an unknown class, this class is generated. The class does not have any instance variables or methods, probably the command fails until the methods are implemented.

After creating the classes, we can create examples. A simple DVD instance holds the instance variables. Testing getter and setter methods is not worthwhile, therefore an example is adequate. An empty EgBrowser can browse the classes by clicking on the root item in the command tree.

Command editor Selecting the examples tab opens the command editor as seen in figure 4.3 (p.34). We built the command editor as an additional code tool for the Refactoring Browser[2]. The editor is a stand-alone tool and can be used without the EgBrowser.

The content of the editor is depending on the selection in the code browser above. If a class is selected without any selector, it displays a command editor for a new method command. If a selector is chosen it searches all commands of this selector in the system and loads the first command into the editor. The commands can be switched with the drop down menu on top of the editor.

New commands New commands can be created using the command editor. The input fields for *name* and *selector* are suggested if a selector is selected. The *receiver* has a drop down list of possible statements of existing tests. The list displays all commands where the class of the command result is the current selected class. The number of *parameter* input fields is adjusted to the selector.

If no selector is selected, the missing selector is generated as seen in section 3.6 (p.27). This is called test-driven design, because the test is created before the method implementation. A debugger will open, because the method body needs to be implemented.

If an *assertion* is given, the editor builds a *method test* (section 3.3.2 (p.20)), else it builds a *method example* (section 3.3.1 (p.20)). A check box transforms the command to a *pessimistic example* (section 3.3.3 (p.21)). The assertions field is invisible in examples.

The *return value* of the command can be either the *receiver* or the *result* of the command. The input fields expect valid Smalltalk code, strings have to be quoted. The assertion text field expects blocks or strings. The assertions are separated by a dot. Assertion

[2] `http://www.refactory.com/RefactoringBrowser/RefactoringBrowser.html`

blocks can use one variable, which is replaced to aResult. Assertion strings can access the variables aReceiver, someParameters and aResult.

The receiver and parameters accept drag and drop from other commands of the tree view. If the shift key is held down it prevents a context switch while dragging.

The editor can also be used to view or modify existing commands in the system.

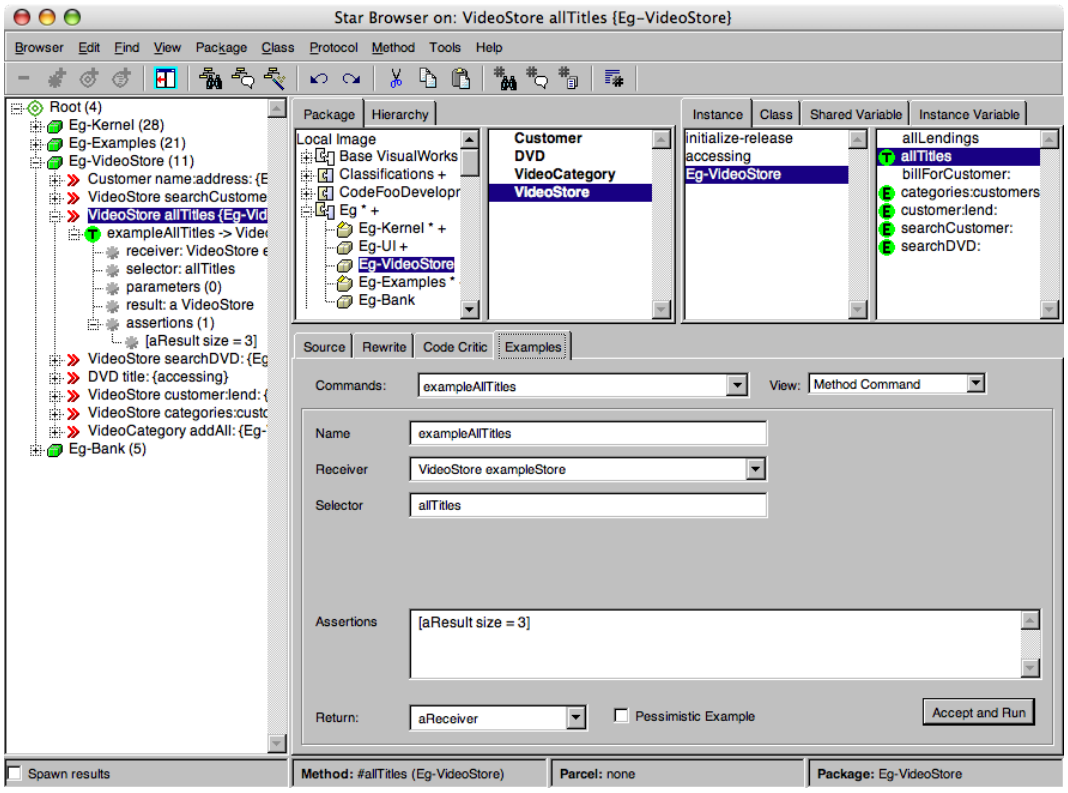

Figure 4.3.: Command editor for a method test

Video store examples With commands we can build up an example video store that is also a test scenario. A DVD example can be modeled with the parameters exampleDVD1 as example name and DVD new from the drop down list as receiver. The selector is title: with the parameter `'Star Wars'`. As return value we need the receiver which is the created object. This example does not need an assertion, thus the assertion field can be left empty. The action button creates the command and runs it. The button changes its color to green if the command is successful.

Figure 4.4 (p.35) shows the command editor while editing a method example.

Multiple DVD examples can be collected in a category example with the receiver Category

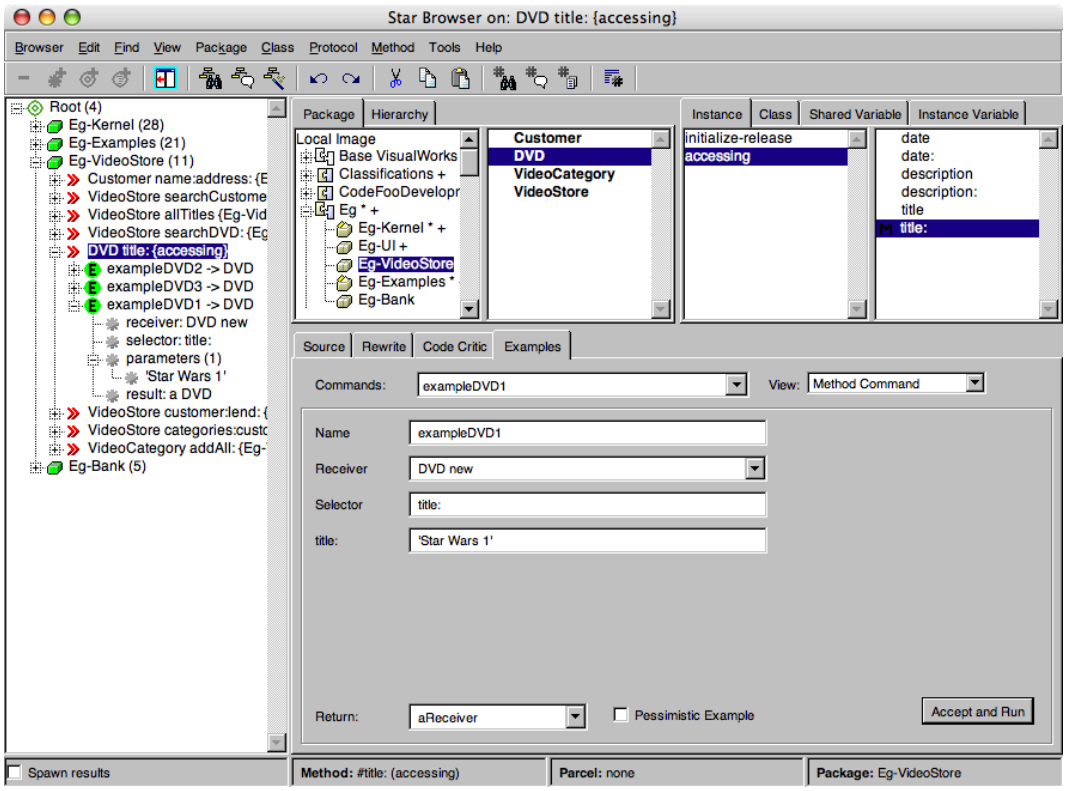

Figure 4.4.: Command editor for a method example

new, selector #addDVD: and (Set with: DVD exampleDVD1 with: DVD exampleDVD2 with: DVD exampleDVD3) as parameter.

In the same manner categories can be added to a video store. At the end there is a video store object with different categories, each category has different DVDs.

```
VideoStore class ≫ exampleStore
    <egClass: #'Eg.MethodExample' method: #categories:customers: >
    | aReceiver aResult |
    aReceiver := VideoStore new.
    aResult := aReceiver
            categories: (Set with: VideoCategory exampleAddAll)
            customers: (Set with: Customer exampleCustomer1
                with: Customer exampleCustomer2).
    ^aReceiver
```

Now business methods can be defined, which use this example store as data for their tests. An example of a loan looks like the following:

```
VideoStore class ≫ exampleLend
```

```
<egClass: #'Eg.MethodExample' method: #customer:lend: >
| aReceiver aResult |
aReceiver := VideoStore exampleStore.
aResult := aReceiver customer: VideoStore exampleSearchCustomer lend: VideoStore
    exampleSearchDVD.
^aResult
```

Figure 4.5 (p.36) shows how drag and drop can be used to create a new command. Both the receiver and parameter are dragged from the tree view.

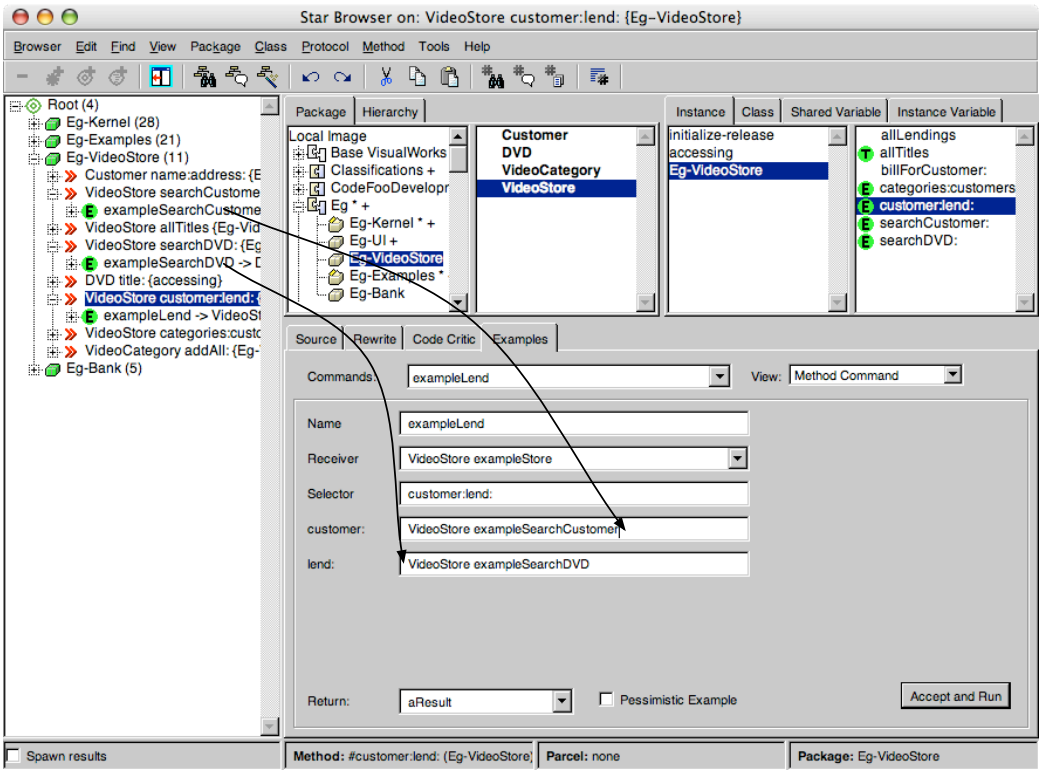

Figure 4.5.: Drag and drop commands to compose a new command

The parameters of the selector #customer:lend: need to be the result of the search commands. Because Customer exampleCustomer1 returns a new object with a different hash, it cannot be used as parameter value. The example VideoStore exampleSearchCustomer returns the customer instance created in the exampleStore command.

4.4.2. Method Suite View

A method suite is a command suite focusing on the same selector. The full description of a method suite is in section 3.4.3 (p.24). All commands in the system with the selected selector are collected to a method suite and displayed in a table. The table shows the commands name, receiver, selector, parameters and result. All listed commands have the same selector, because they belong to the same method suite. The interesting fields are the receiver, parameters and result. They show the used values of the different commands. Figure 4.6 (p.37) shows the table of the method suite with #title: as selector. The table is read only and cannot be edited.

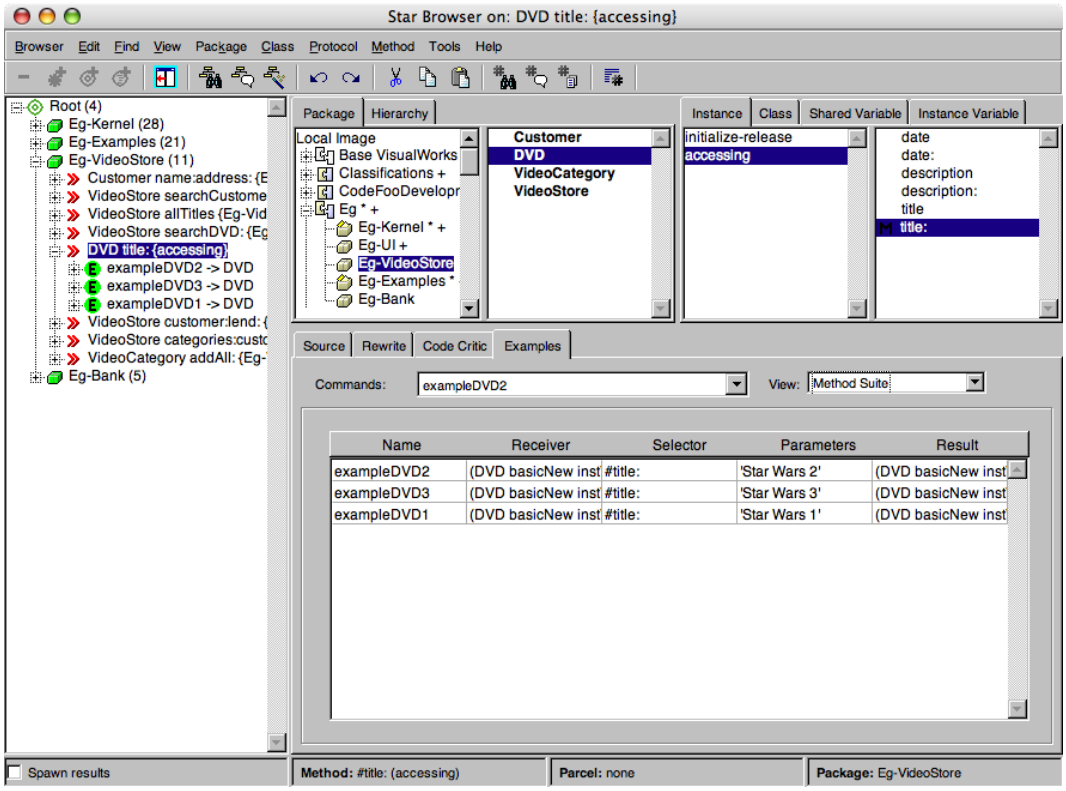

Figure 4.6.: Method suite view

4.4.3. Coverage View

The coverage view lists all methods called by the current selected command. Method wrappers[3] are installed on all selectors of the package containing the command [Duc99, BFJR98]. After running the command, all wrappers are uninstalled. Installing wrappers

[3] `http://www.refactory.com/Software/MethodWrappers/`

on a package slows down the application execution, because all wrapped methods are logging their access.

The coverage view can display the coverage of a command. This coverage displays all access of a command from other commands. But because the meta-model knows the methods of cascaded commands, this is not essential.

The coverage view from figure 4.7 (p.38) shows all classes and selectors called by the exampleLend command. The entries in the table are chronological, the first call is on top. Duplicated entries are removed from the table, the entry is at the first occurrence of a call. The table cannot be modified.

Another approach to display the coverage would be to add a 5th pane to the class browser. The additional pane could display all tests called by the current selected method. The pane could also display the implementing method if a test is selected in the 4th pane. We choose to implement the coverage view as tab instead of an additional pane because it is simpler to implement and the browser remains modular.

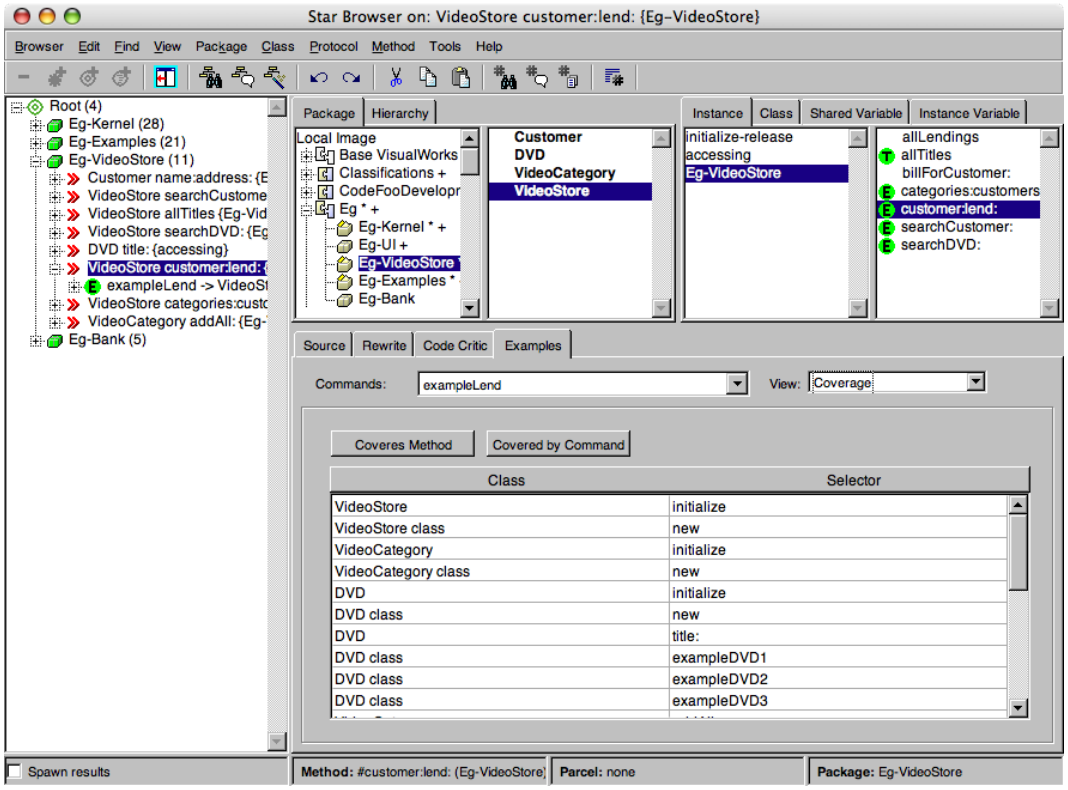

Figure 4.7.: Coverage view

4.4.4. Suite Editor

Commands can be collected to an independent command suite as described in section 3.4.1 (p.24). For example all commands of a package can be bundled to an independent command suite.

Figure 4.8 (p.39) shows a suite editor composing different commands to a suite. The left list displays all commands of the system, the right list displays the commands in the suite. The right arrow button adds the selected command to the suite. The command is removed from the left list. The left arrow button removes the selected command in the right list and adds it to the left list again. The accept button adds the suite to the command tree of the browser.

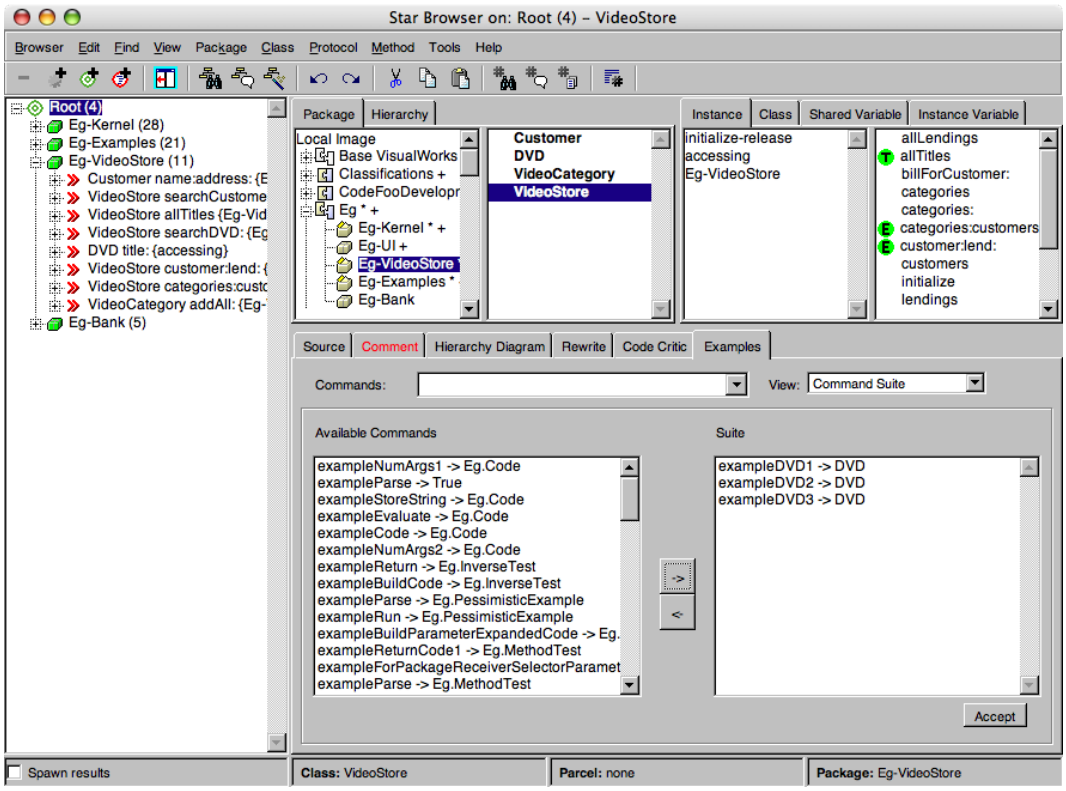

Figure 4.8.: Command suite view

4.5. Importing SUnit tests

Most of the existing Smalltalk unit tests are written with SUnit. These tests are difficult to convert to the meta-model. SUnit does not denote which method is under test, so it

is hard to recognize and classify the tests automatically [Mar05, GLN05]. SUnit tests are scattered, they can use a setup method and inherit functionality from their parent classes. Multiple methods under test can exist in the same test case and assertions can be placed anywhere. A method under test can even be called in an assertion.

To convert a SUnit test to an Eg test, the user has to select the method under test with the mouse. Figure 4.9 (p.40) shows the context menu in the browser, where the selected code can be converted to a command. The code needs to be a simple statement. Multiple methods under test cannot be split into multiple method tests automatically.

The converter tries to gather all informations needed from the SUnit test. The setup method is inlined and the parameters are expanded. All assertions before the selected method are ignored in the current implementation, because Eg does not have preconditions. The assertions after the method are adopted.

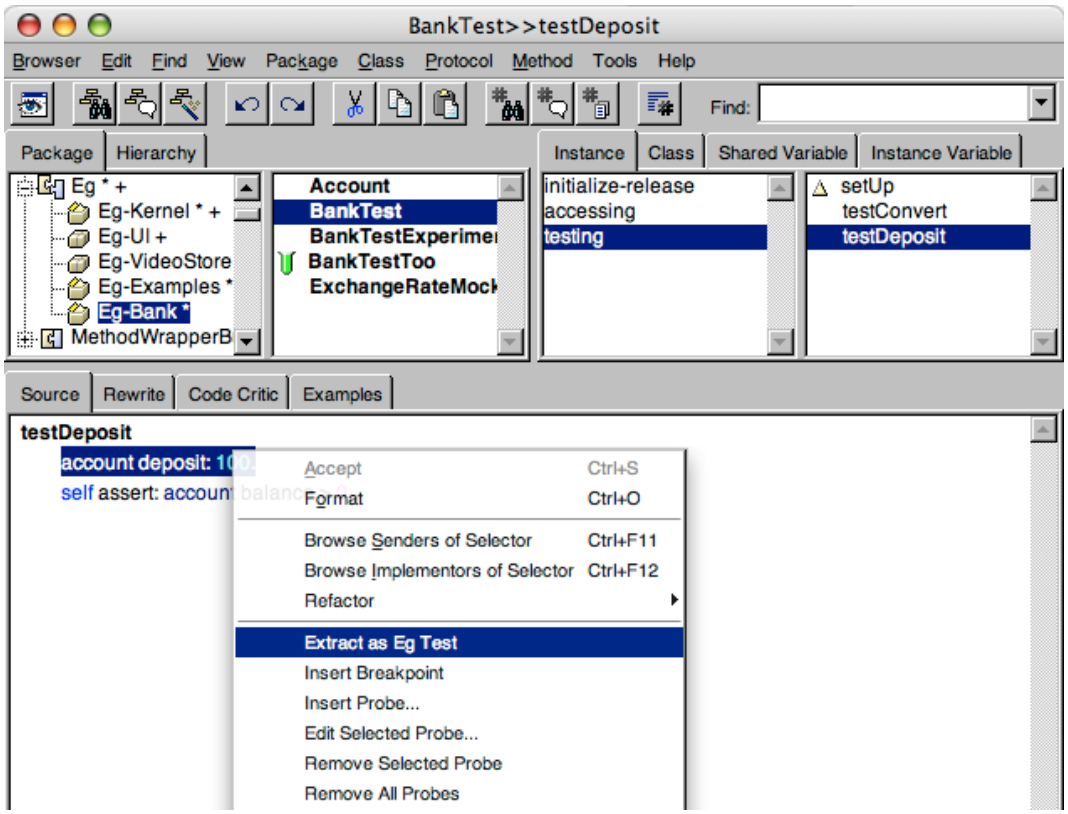

Figure 4.9.: Import a SUnit test to Eg

This is the code of a simple SUnit test. The test message is account deposit: 100. The test uses a setup method. All assertions are separated and no statements are nested.

```
BankTest ≫ testDeposit
    account deposit: 100.
```

```
    self assert: account balance > 0
```

The converter creates the following Eg method test. The receiver value is taken from the setup method of the SUnit test. The variable in the test message and assertion changes to aReceiver.

```
Account class ≫ exampleDeposit
    <egClass: #'Eg.MethodTest' method: #deposit:>
    | aReceiver aResult |
    aReceiver := (Account new).
    aResult := aReceiver deposit: 100.
    self assert: aReceiver balance > 0.
    ^aReceiver
```

5. Validation

We introduced some testing tools in chapter 2 (p.7) and our tool in chapter 4 (p.29). But can we compare the different tools? Is the EGBROWSER more efficient or does the user prefer the existing tools?

This chapter tries to measure the different tools. We compared different features with different techniques.

Software validation is an important part of building an application. Especially the *user interface* needs a lot of validation because this is the connection to the end user. The user does not care about the underlying code structure or model, he wants a simple and functional user interface!

The validation helps to match the user's requirements and detects misunderstandings and defects. The evaluation process should be done while developing the application. It is too late to evaluate a final product.

Dix *et al.* [AD04] state three main goals of evaluation:

- Assess accessibility of the system's functions
- Assess user's experience of interaction
- Identify any problems

Metrics A metric is a set of *numerical parameters* that can be measured. The parameters have a unit and can be compared. Metrics can compare the *performance* of a process.

Software metrics are popular in software design. These metrics can assess code quality by counting the size of the code, number of methods, coherence of classes and a lot of other metrics. But evaluating code quality with metrics is dangerous because not all metrics are meaningful and accurate.

The interface look and behavior is significant for the user of an application. Hence it is important to validate the user interface. Ivory *et al.* [IH01] say it is difficult to find good parameters for a user interface metric. Counting the number of clicks or keyboard inputs is a simple metric, but it does not state something about the usability of a system. Measuring the time a user needs to fulfill a certain process can benchmark the usability,

but this depends on how fast the user can interact with the system. Interviews are useful to assess the interface, but they need an analysis and cannot be compared.

The main problem is that a user interface always needs some *user interaction*. The user interaction cannot be simulated by an application, it requires a *real person*. The test person has consequences to the evaluation results. The results are not the same if they are done with a novice user or an expert.

5.1. GOMS

GOMS stands for Goals, Operators, Methods, and Selection rules. GOMS was developed in 1983 by Card, Moran and Newell [CNM83]. They conducted research in human computer interaction and psychology. Today GOMS is the best known theoretical concept of human computer interaction. It is used to measure user interaction in software development [JK94, JK96, AD04]. There are four different variations of GOMS. They are different in the level of detail and complexity.

A *goal* is what the user has to accomplish. A goal can be divided into sub-goals. An *operator* is a single step to reach a goal, for example a mouse click or key-stroke. A *method* is a sequence of operators. If more than one method for a goal exists, the *selection rule* is applied.

GOMS is a model-based evaluation and does not need a user to participate. The GOMS evaluation is done by an expert. The expert knows how to interact with the system. He does not have the same problems than a novice user.

5.1.1. KLM-GOMS

The Keystroke-Level Model is the simplest GOMS variant. It is a simplified version of CMN-GOMS. It eliminates the goals, methods, and selection rules. Only the operators are measured. To estimate the time spent for a task, the execution time of the operations is summed.

KLM has five different operators, listed in table 5.1 (p.45). Each operator has an execution time that is a mean value for an average user. K is the time needed to press a key or mouse button. The time for performing a keystroke can vary from 0.12 (good typist) to 1.2 seconds (non-typist). P is to point the mouse to an object on the screen. If this time needs to be more accurate, Fitts' law [Fit54] can be applied: $t = 0.1 * log_2(Distance/Size + 0.5)$. The time needed for pointing on an object depends on the object's size and the distance. H is the time needed to home the hands on the keyboard or mouse. A lot of homing actions can be annoying. M is the mental preparation the user needs before executing an action. Mental preparation is used before each P and K if it is not part of a string. R is the time the system needs to react on the

user's input. This time is task dependent and is negligible in most cases because today's computers are fast enough.

Table 5.1.: KLM-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's input

To demonstrate the different variants of GOMS we use a small example. The goal is to save a document under a certain name in a common text editor. Table 5.2 (p.45) shows the KLM-GOMS steps. This table is not in a consistent form because KLM-GOMS does not define a syntax.

Table 5.2.: KLM-GOMS example

Description	Operator	Time (sec)
Mental preparation	M	1.3
Hit ctrl, shift and s to open save dialog	3*K	0.6
Find a name	M	1.3
Enter the name	10*K	2
Prepare to close dialog	M	1.3
Hit enter	K	0.2
Total predicted time		6.7

5.1.2. CMN-GOMS

CMN-GOMS is the original GOMS proposed by Card, Moran and Newell. It has a strict *goal hierarchy* and considers also the selection rules. Thus it is more flexible and more complex. The depth of the goal hierarchy can be used to estimate the memory usage. CMN-GOMS does also predict the *operator sequence*.

The CMN-GOMS code is in program form and uses a pseudo code language. It does not define the syntax, but provides a guide for the selection rules.

Table 5.3 (p.46) is the same example from above. The syntax is not defined, but this notation is mostly used in publications. The hierarchy is indented with dots.

Table 5.3.: CMN-GOMS example

Description	Time (sec)
Goal: save file	
. Goal: open save dialog	
. . Mental preparation	1.3
. . Hit ctrl, shift and s to open save dialog	0.6
. . Find a name	1.3
. . Enter the name	2
. . Prepare to close dialog	1.3
. . Hit enter	0.2
Total predicted time	6.7

5.1.3. NGOMSL

Natural GOMS Language is like CMN-GOMS, but has a high level syntax. Therefore it is easy to use like KLM-GOMS, but has the power of CMN-GOMS. NGOMSL uses the *cognitive complexity theory (CCT)*. CCT assumes that the actions are serial and the memory triggers the production rule at a fixed rate [JK96]. The mental preparation operator is different from the previous GOMS variants and is used before every step.

NGOMSL can also predict the learning time of an individual task. The time is computed with heuristics and the number of subtasks and bases on CCT [JK96]. The execution time prediction is similar to the previous GOMS variants.

Table 5.4 (p.46) shows the example in the natural GOMS language. It assumes that the mental preparation time in CCT is the same as with KLM-GOMS.

Table 5.4.: NGOMSL example

Description	Executions	Time (sec)
Method for goal: save file		
Step 1: Mental preparation	1	1.3
Step 2: Hit ctrl, shift and s to open save dialog	1	0.6
Step 3: Find a name	1	1.3
Step 4: Enter the name	1	2
Step 5: Prepare to close dialog	1	1.3
Step 6: Hit enter	1	0.2
Total predicted time		6.7

5.1.4. CPM-GOMS

CPM stand for Cognitive-Perceptual-Motor and Critical Path Method. Critical Path Method is a mathematical algorithm for scheduling a set of project activities. The critical path predict the task time. CPM-GOMS can also make time predictions, but assumes that the tasks can be executed in parallel. The CPM-GOMS model is based directly on the *Model Human Processor (MHP)* and is much more detailed. To compute a simple read from screen task, even the eye movement time is considered.

5.2. Validation of the EgBrowser

The GOMS model can predict execution times. We use GOMS to compare the EgBrowser with SUnit with Refactoring Browser Extensions support. SUnit is the most used testing tool in Smalltalk and has enough features for a comparison. The other tools mentioned in chapter 2 (p.7) are similar to SUnit and would get a likewise time prediction.

Creating a test is a rather long task for a GOMS validation. Thus we choose the simplest variant, the KLM-GOMS. It is accurate enough to assess the tools, but the disadvantage is that the learning time cannot be predicted. All further mentions of GOMS refer to KLM-GOMS.

We use the GOMS execution time prediction as a metric; less time means better values. Three typical tasks of the EgBrowser are selected and compared to SUnit. Eg uses a meta-model and SUnit inheritance. The tools are not built for the same tasks. Therefore SUnit will have problems with tasks that Eg is specialized on. Where a feature is missing in SUnit we use the simplest way to get a similar result.

The GOMS results of the comparison are in table 5.5 (p.47), the different steps are in appendix A (p.63).

Table 5.5.: GOMS results

Task	Tool	K	P	H	M	Time
5.2.1	SUnit	134	11	12	21	71 sec
	Eg	32	6	4	8	25 sec
5.2.2	SUnit	125	10	4	17	60 sec
	Eg	91	11	10	15	54 sec
5.2.3	SUnit	3	2	0	3	7 sec
	Eg	3	3	0	2	7 sec

5.2.1. 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 uses the bank account example. 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. The several steps are in appendix A.1 (p.63).

SUnit used 178 steps in 71 seconds. A lot of mental work and typing is needed with SUnit. A person would have longer to create this test because the mental work assumes the person knows exactly what to. But coding is rarely straight forward, often the formulation changes during development.

The class and the setup method can be shared by different SUnit tests. If they are already 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 leads the developer in writing tests it uses less mental work than in a free text field like SUnit.

5.2.2. Creating a test for a new method

The same scenario as in section 5.2.1 (p.48), except that the method #withdraw: is not yet implemented. The browser has the bank test class selected. The test class and setup method of SUnit can be reused from the first comparison. The several steps are in appendix A.2 (p.65).

SUnit used 156 steps in 60 seconds. A lot of steps are reused from section 5.2.1 (p.48) as a new test class and setup method is 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.

5.2.3. Browsing between test and implementation

The starting point is a newly created test similar to the end position of section 5.2.1 (p.48). The several steps are in appendix A.3 (p.67).

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.

5.3. Usability Experiment

The best usability tests are done with real users. We built a tool for unit testing, so developers are our users.

The experiment is done with real user participation under *laboratory conditions*. Laboratory conditions mean the user tests the tool in a prepared setup and not in daily work (field study). This conditions and the experiment in general can affect the results.

5.3.1. 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 is easier to use than SUnit.
- It is faster to accomplish the tasks.
- The EgBrowser is easy to use without previous knowledge.

Techniques We did the experiment with *pair programming*. Pair programming is a common way to develop software. Two developers are drawn by lot, resulting in five teams. We used *think-aloud* as an observation technique as Dix *et al.* [AD04] propose. Holzinger [Hol05] mentions that think-aloud is easier to arrange with pair programming. The users should describe and explain their thoughts while interacting with the system.

To analyze the record later, the process is filmed with a video camera. The camera records the conversation of the think-aloud, but also the screen where the interaction with the system happened. The screen is recorded with a screen capture application, too. Additionally the participants answer a short questionnaire after the experiment.

5.3.2. 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.

The solutions using both tools are in appendix B (p.69).

5.3.3. Questionnaire

These questions are answered by the participants after doing the experiment. Each question is answered for SUnit and the EgBrowser using 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

5.3.4. Questionnaire Analysis

The answers of the questionnaire are shown on figure 5.1 (p.51). The results are biased because most participants are SUnit experts and use SUnit every day. Another caveat is that SUnit is a well proven tool. 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 users, it is unambiguous they prefer SUnit and believe it is easier to use. Some users like the idea of tests and examples as

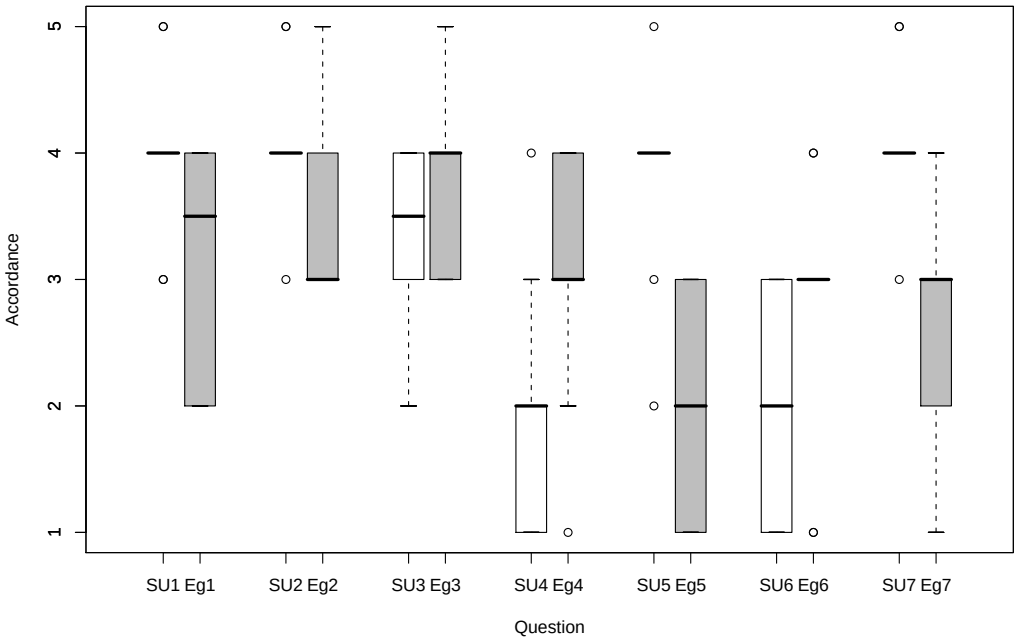

Figure 5.1.: Results of the questionnaire displayed as box plots

reusable commands. The limiting factor was mainly the user interface, because this was the first test they implemented with the EgBrowser and the user interface was not intuitive.

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.

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 neither tool provides a 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 offers tooltips 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. The users did not know what the underlying meta-model was doing. And the participant knew what SUnit was doing in the background.

The goal of the EgBrowser is to support the developer in writing tests. In both questions concerning the workflow and user support (questions 3 and 4) the EgBrowser scored higher than SUnit.

5.3.5. Video Analysis

The same tasks were performed with two different tools. SUnit is an approved tool. 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 was 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 saved 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 bugs is subtracted from the EgBrowser time. The estimation of the bug time is ambiguous and might be too optimistic in same cases. 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 task was done twice, it was implemented faster 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 4 had problems defining a class because neither team member was familiar with the VisualWorks environment.

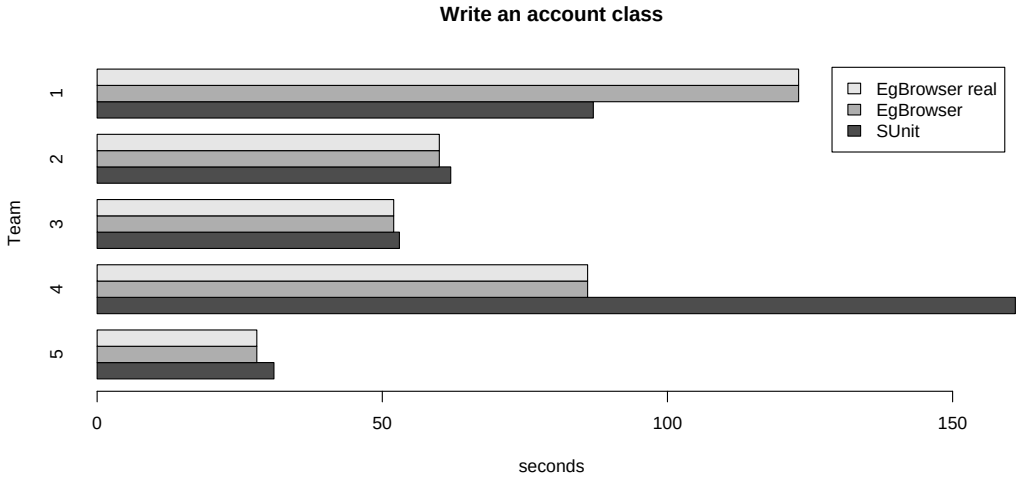

Figure 5.2.: Analysis of task 1

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 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 comprehend 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 was not working because the meta-model compiled the example to the wrong class.

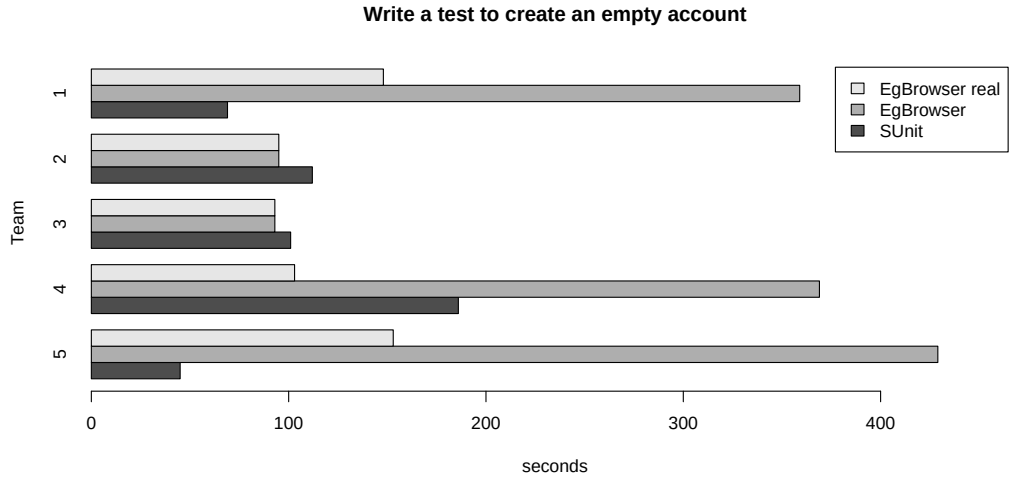

Figure 5.3.: Analysis of task 2

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 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.

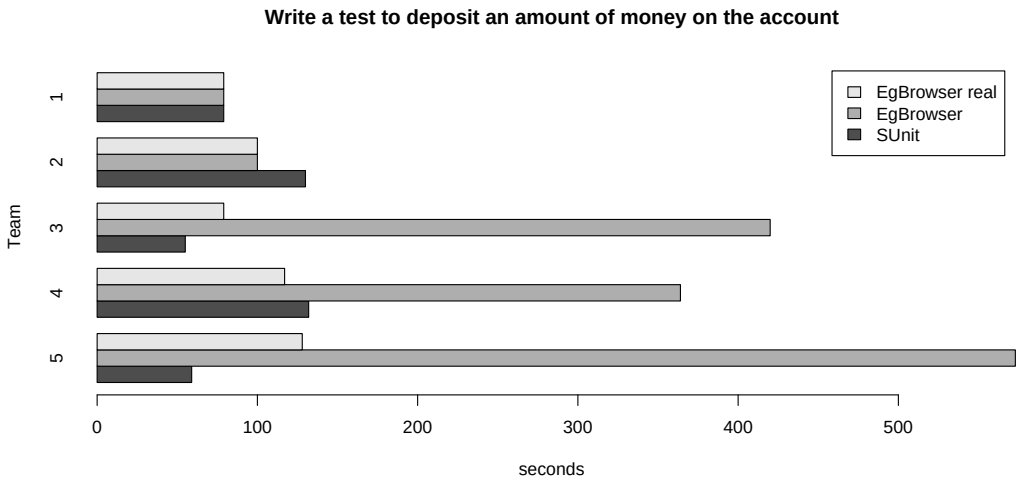

Figure 5.4.: Analysis of task 3

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.

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

Most teams did not remember how to raise an exception and how to catch it with a SUnit test.

Team 2 needed to browse the SUnit class to look up the syntax for the failed test.

Team 3 needed much time to implement a working withdraw method. They had to re-implement the command because it was not saved while fixing the method. Finally they overwrote the command from task 4 because they had chosen the same name.

Team 5 needed to redo the command because of the interface problems.

Task 6: 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 tests and implementations with the meta-model. SUnit does not have this feature and thus is not comparable.

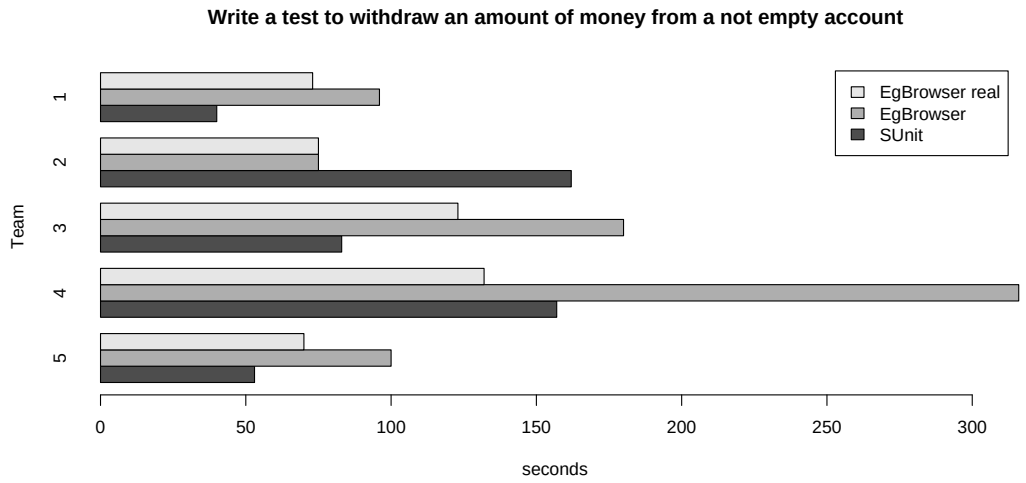

Figure 5.5.: Analysis of task 4

5.3.6. Conclusion

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 the usage of the 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 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.

5.4. Case Study: Manually Converting SUnit Tests to Eg

To prove the usability of our meta-model we tried to convert some existing SUnit tests to the meta-model. The application to convert should not have too simple tests, but needs to be well tested. The tests should not be too complicated, else it is difficult to understand the content in a small amount of time.

We chose Mondrian as application for the case study. Mondrian is a scriptable, dynamic

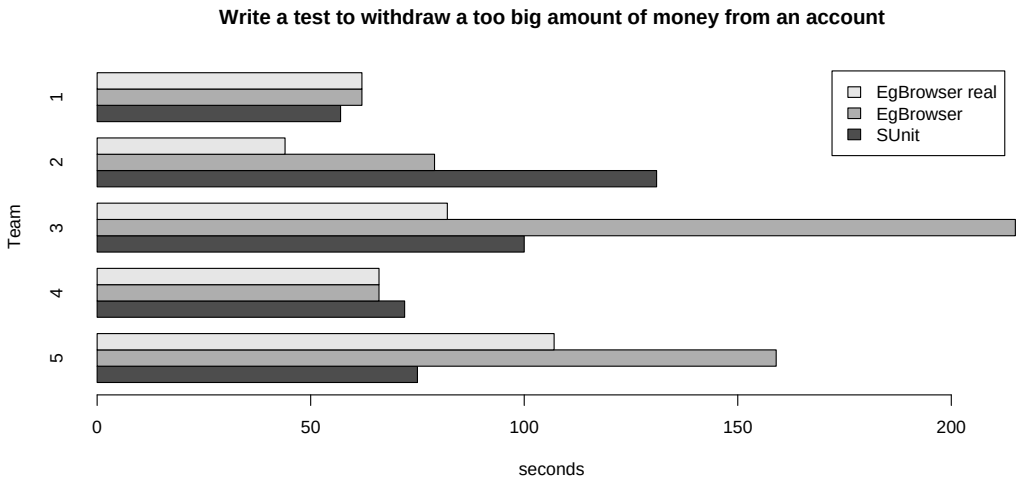

Figure 5.6.: Analysis of task 5

visualization framework developed by Meyer *et al.* [MGL06]. Mondrian can handle all sorts of data and supports different charts as output. Currently Mondrian has about 190 SUnitToo tests. We selected a sample of 35 tests to categorize to the meta-model. Figure 5.7 (p.58) shows an overview of the resulting test types.

A majority of 18 test cases can be converted to method tests. The problem 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. The assertions checks some value of them. This is rather black box testing because the really interesting methods are not visible in the test case. It is even hard to find the real method with a debugger.

Another 9 test cases we categorized into two method tests. 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. In our model we can combine them as a method suite.

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 of 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.

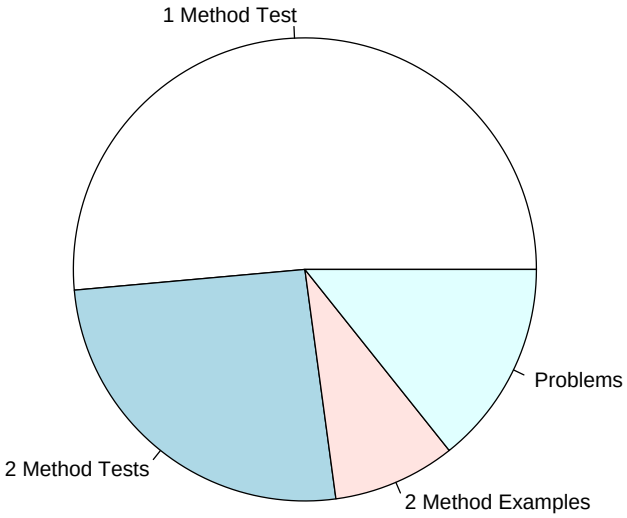

Figure 5.7.: Converted test types of Mondrian

6. Conclusion

As stated in chapter 1 (p.1), the main problems of unit testing are the missing link between test and implementation, the flat test hierarchy and the missing user interfaces.

The tests cannot be reused with existing frameworks, the resulting test hierarchy is flat. Often code is duplicated to achieve an identical setup in different tests. But duplicating code leads to an unmaintainable project.

The test hierarchy is flat because the frameworks do not allow one to build a test scenario. On the other hand the developers are used writing flat test scenarios. It is doubtful if the developers create the better test scenarios if a better test hierarchy is possible. The hierarchy brings more order to the tests, but leads to a more abstract test environment.

In this thesis we introduced our meta-model for commands. The meta-model holds the link between test and implementation. The tests from the model have return values, therefore the commands can be cascaded to test scenarios.

Our experiments show that most tests can be transformed according to our meta-model. Because a test often tests a single method only, it can be formulated as a method test. More complex tests can be transformed to method commands or cascaded method command scenarios, but the resulting method command can be much longer or complicated than the original test.

We built a test editor based on the meta-model. The editor supports the developer by providing useful input values in the text fields. The developer can compose commands by drag and drop, therefore the workflow is improved.

The test editor has proven to work for inexperienced users. The users in our experiments have learned the functionality quickly and were able to implement the tests with it as quickly as with SUnit. But some developers have mentioned that a text based editor is more powerful for the advanced user.

6.1. Future Work

A lot of additional features could be implemented in the editor.

Currently there is no debugger support. In the debugger the instance variable values can be seen and modified. If the object in the debugger is a good example of this class, the user should be able to generate the command out of the debugger.

The meta-model encourages the reuse of commands. The test hierarchy becomes a tree. The command tree can be visualized with Mondrian [MGL06]. The test runner then should be able to execute a subtree only if the required commands run without error.

Because most existing tests are written in SUnit, an importer to Eg is important. SUnit tests can be complicated and it is difficult to write an automated SUnit importer. The importer would use a lot of artificial intelligence to detect the method under test, because it can be in an assertion or nested in another method. The current importer can convert well formed tests. User interaction is needed to identify the test statement.

The Smalltalk class browser has 4 columns: package, class, protocol and selector. We can add a 5th column for the tests. This column shows all tests of the selector of the 4th pane. If the 4th pane shows a test method, the 5th pane shows the selectors tested by this test. Figure 6.1 (p.60) shows an implementation of this browser in Squeak. The last column displays the command Account class ≫ withdrawOkFrom123. It is a test for the #withdraw: method in column 4.

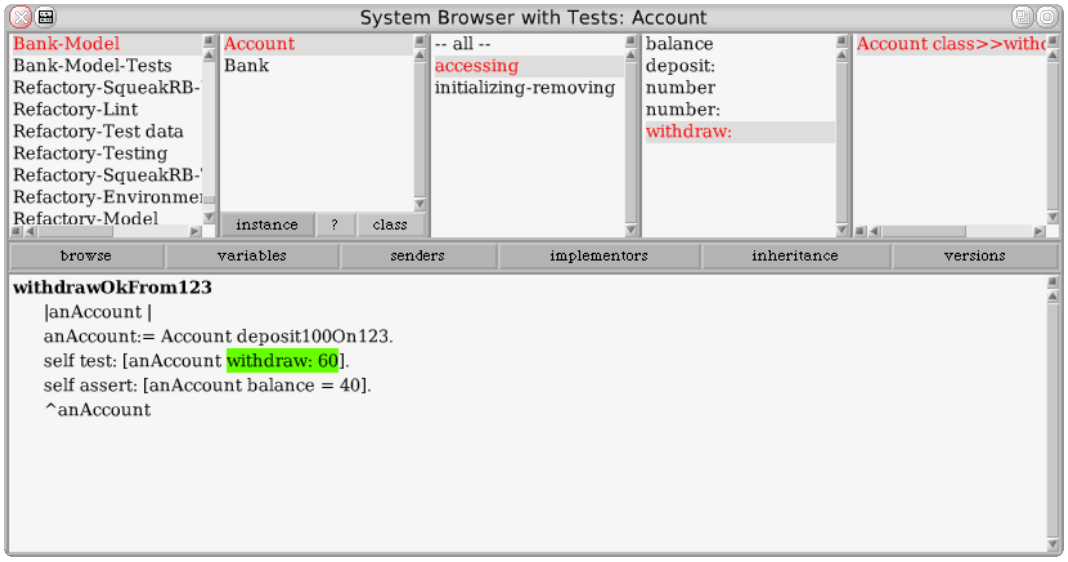

Figure 6.1.: 5th pane Browser

6.2. Further Validations

In our first usability study (section 5.3 (p.49)) we received good feedbacks from the participants. After implementing the new ideas, the next step would be to do another usability study.

We would do this further usability study again with pair programming under laboratory conditions. This setup has proved to be efficient. But we would give a longer EgBrowser crash-course or distribute the application before the study so that the developers could already play around and get used to it. This could decrease the knowledge difference between the EgBrowser and SUnit. In the questionnaire we would focus more on usability, work flow and user guidance.

As the EgBrowser is more evolved now, we could try a field study to detect if the EgBrowser is usable for daily work.

A. GOMS Steps

A.1. Creating a test for an existing method

SUnit

1. M: Think how to create a new test class
2. P: Point the mouse to the classes in the browser pane
3. K: Right click to open the context menu
4. M: Find the right entry
5. P: Point to New Class...
6. K: Click. The new class wizard opens
7. M: Find a name for the test class
8. H: Hands to the keyboard
9. 8*K: Enter the name on the keyboard. BankTest has 8 characters
10. M: Think about a superclass. We need TestCase as superclass
11. H: Hand to the mouse
12. P: Point to the 3 dots button right to the superclass input field
13. K: Click. A class selection tool with a search box opens
14. M: Figure out how this works
15. H: Hands to the keyboard
16. 4*K: Enter test in the input field
17. M: Find right entry
18. H: Hand to the mouse
19. P: Point to TestCase
20. K: Doubleclick on it
21. M: We need an account as instance variable
22. P: Point to the instance variable input field
23. K: Click
24. H: Hands to the keyboard
25. 7*K: Enter account
26. M: Think how to finish this wizard
27. H: Hand to the mouse
28. P: Point to the OK button
29. K: Click. The wizard closes
30. M: We need a setup method for the account instance variable.
31. P: Point to the protocols
32. K: Right click to open the context menu
33. M: Find the New.. entry
34. P: Point to it
35. K: Click
36. M: Find a name
37. H: Hands to the keyboard
38. 7*K: Enter testing
39. K: Hit enter
40. M: Think what to do next
41. H: Hand to the mouse
42. P: Select the code from the method stub
43. M: Remember the selector
44. H: Hands to the keyboard

45. 6*K: Enter setUp and return
46. M: Think about the code we need
47. 22*K: Enter account := Account new
48. M: Compile the code
49. 2*K: Hit ctrl-s
50. M: Now create the test method
51. H: Hand to the mouse
52. P: Select the code from the setup method and overwrite it with the method code
53. M: Find a valid test selector
54. H: Hands to the keyboard
55. 12*K: Enter testDeposit and enter
56. M: Add money to the instance variable initialized in the setup method
57. 22*K: Enter account deposit: 100. and return
58. M: Write an assertion
59. 32*K: Enter **self** assert: account balance > 0
60. M: Compile the code
61. 2*K: Hit ctrl-s
62. M: Figure out how to run the test
63. H: Hand to the mouse
64. P: Point to the run button on the bottom of the browser
65. K: Click the button. The test should run without an error

EgBrowser

1. M: Think how to use the EgBrowser
2. P: Point to the Examples tab
3. K: Click. The command editor opens on a new command
4. M: We agree with the recommended name exampleDeposit
5. M: Think about the receiver
6. P: Point to the triangle left to the receiver input field
7. K: Click to open the menu
8. M: Find the right entry (only one is available)
9. P: Point to Account new
10. K: Click to accept
11. M: The selector has a parameter
12. P: Point to the parameter input field
13. K: Click
14. H: Hands to the keyboard
15. 3*K: Enter 100
16. M: We need an assertion
17. H: Hand to the mouse
18. P: Point to the assertions input field
19. K: Click into it
20. M: Remember how to formulate an assertion
21. H: Hands to the keyboard
22. 23*K: Enter `'aReceiver balance > 0'`
23. M: Figure out how to finish the command
24. H: Hand to the mouse
25. P: Point to the run button
26. K: Click the accept button

A.2. Creating a test for a new method

SUnit

1. M: Create a new test for a not yet defined method
2. P: Point to the testing category
3. K: Click
4. P: Select the code stub
5. M: Find a valid test selector
6. H: Hands to the keyboard
7. 13*K: Enter testWithdraw and enter
8. M: Remove money from the instance variable initialized in the setup method
9. 23*K: Enter account withdraw: 100. and return
10. M: Write an assertion
11. 32*K: Enter **self** assert: account balance < 0
12. M: Compile the code
13. 2*K: Hit ctrl-s
14. M: An error pops up because the method #withdraw: is not yet known
15. K: Hit enter
16. M: Figure out how to run the test
17. H: Hand to the mouse
18. P: Point to the run button on the bottom of the browser
19. K: Click the button. The test fails
20. M: Think how to debug the test
21. P: Point to the debug button
22. K: Click
23. M: The debugger opens on Account ≫ doesNotUnderstand:. We need to fix this bug
24. P: Point to the selected message
25. K: Right click. The context menu opens
26. M: Find the entry Define Method
27. P: Point to the entry
28. K: Click
29. M: The debugger opens on Account ≫ halt. We need to select the message before
30. P: Point to the message before the halt
31. K: Click
32. M: The method stub for the withdraw: arg1 opens
33. P: Select the argument, comment and halt statement
34. H: Hands to keyboard
35. 7*K: Enter amount as argument for the method. Enter
36. M: Find the code for the method body
37. 29*K: Enter balance := balance − amount. and enter
38. M: Write also a return statement
39. 8*K: Enter ^balance
40. M: Compile the code
41. 2*K: Hit ctrl-s
42. M: Resume the test
43. H: Hand to the mouse
44. P: Point to the run button (big right triangle)
45. K: Click
46. M: Run the test again
47. P: Point to the run button on the bottom of the browser
48. K: Click the button. The test passes

EgBrowser

1. M: Think how to use the EgBrowser
2. P: Point to the Examples tab
3. K: Click. The command editor opens on a new command
4. M: Find a name for the test
5. P: Point to the name field
6. K: Click
7. H: Hands to the keyboard
8. 8*K: Append Withdraw
9. M: Think about the receiver
10. H: Hand to the mouse
11. P: Point to the triangle left to the receiver input field
12. K: Click to open the menu
13. M: Find the right entry
14. P: Point to Account new
15. K: Click to accept
16. M: The selector field is empty
17. P: Point to the selector field
18. K: Click
19. H: Hands to the keyboard
20. K: Enter withdraw:
21. M: The selector has a parameter
22. H: Hand to the mouse
23. P: Point to the parameter input field
24. K: Click
25. H: Hands to the keyboard
26. 3*K: Enter 100
27. M: We need an assertion
28. H: Hand to the mouse
29. P: Point to the assertions input field
30. K: Click into it
31. M: Remember how to formulate an assertion
32. H: Hands to the keyboard
33. 23*K: Enter `'aReceiver balance < 0'`
34. M: Figure out how to finish the command
35. H: Hand to the mouse
36. P: Point to the run button
37. K: Click the accept button
38. M: The debugger opens on Account ≫ halt. We need to select the message before
39. P: Point to the message before the halt
40. K: Click
41. M: The method stub for the withdraw: anInteger opens
42. P: Select the argument, comment and halt statement
43. H: Hands to keyboard
44. 7*K: Enter amount as argument for the method. Enter
45. M: Find the code for the method body
46. 29*K: Enter balance := balance − amount. and enter
47. M: Write also a return statement
48. 8*K: Enter ^balance
49. M: Compile the code
50. 2*K: Hit ctrl-s
51. M: Resume the test
52. H: Hand to the mouse
53. P: Point to the run button (big right triangle)
54. K: Click. The test runs without an error

A.3. Browsing between test and implementation

SUnit

1. M: Think how to browse the method under test
2. M: Look for the method under test in the test code
3. P: Point to the method #deposit:
4. K: Select the method name
5. K: Right click to open the context menu
6. M: Find the Browse Implementors of Selector entry
7. P: Point to the right class
8. K: Click. The source code is displayed in the window

EgBrowser

1. M: Think how to browse the method under test
2. P: Point to the package icon in the tree view
3. K: Click to open the subtree
4. M: Find the right method reference
5. P: Point to the method
6. K: Click to open the subtree
7. P: Point to the method or command to browse
8. K: Click to open the source code in the editor

B. Experiment Solution

B.1. SUnit

1. Account class:

```
Smalltalk defineClass: #Account
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: 'balance '
    classInstanceVariableNames: ''
    imports: ''
    category: 'Eg-Bank'
```

   Accessor:

```
Account ≫ balance
    ^balance
```

   Initialize method:

```
Account ≫ initialize
    balance := 0
```

   Finally overwrite the new method:

```
Account class ≫ new
    ^super new initialize
```

   As an alternative solution the wizard from the Refactoring Browser can be used.

2. SUnit test class:

```
Smalltalk defineClass: #BankTest
    superclass: #{XProgramming.SUnit.
        TestCase}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: ''
    category: 'Eg-Bank'
```

   Test method for empty account:

```
BankTest ≫ testEmpty
    | account |
    account := Account new.
    self assert: account balance = 0
```

3. Deposit test with test-driven design. Create test first:

```
BankTest ≫ testDeposit
    | account |
    account := Account new.
    account deposit: 100.
    self assert: account balance = 100
```

   Create method in debugger:

```
Account ≫ deposit: amount
    balance := balance + amount
```

4. Withdraw test:

```
BankTest ≫ testWithdraw
    | account |
    account := Account new.
    account deposit: 100.
    account withdraw: 80.
    self assert: account balance > 0
```

   Again define method in debugger:

```
Account ≫ withdraw: amount
    balance >= amount
        ifTrue: [balance := balance −
            amount]
        ifFalse: [RangeError raiseSignal]
```

5. Withdraw too much to raise error:

```
BankTest ≫ testWithdrawTooMuch
    | account |
    account := Account new.
    account deposit: 100.
    self should: [account withdraw: 120]
        raise: Error
```

6. Manually browse to the method.

B.2. EgBrowser

1. Same steps as in appendix B.1 (p.69).
2. Create a new method test:
   Name: exampleEmptyAccount
   Receiver: Account new
   Selector: balance
   Assertion: aResult = 0
3. Create a new method test:
   Name: exampleDeposit
   Receiver: Account exampleEmptyAccount
   Selector: deposit:
   Parameter: 100
   Assertion: aResult balance = 100

   Create method in debugger:

   ```
   Account ≫ deposit: amount
       balance := balance + amount
   ```

4. Create a new method test:
   Name: exampleWithdraw
   Receiver: Account exampleDeposit
   Selector: withdraw:
   Parameter: 80
   Assertion: aResult balance > 0

   Again define method in debugger:

   ```
   Account ≫ withdraw: amount
       balance >= amount
           ifTrue: [balance := balance −
               amount]
           ifFalse: [RangeError raiseSignal]
   ```

5. Create a pessimistic example:
   Name: exampleWithdrawTooMuch
   Receiver: Account exampleDeposit
   Selector: withdraw:
   Parameter: 120
   Check the pessimistic example box
6. Use the tree to browse to the method of a command. The method is the parent of the command.

List of Tables

List of Figures

Bibliography

[AD04] Gregory D. Abowd Alan Dix, Janet E. Finlay. *Human-Computer Interaction (3rd Edition)*. Prentice Hall, 2004.

[Bec] Kent Beck. Simple Smalltalk testing: With patterns. `http://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.

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

[Csi91] Mihaly Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Harper Perennial, March 1991.

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

[Fit54] Paul M. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology*, 47(6):381–391, 1954.

[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 taxonomy of SUnit tests. In *Proceedings of ESUG 2005 (13th International Smalltalk Conference)*, September 2005.

[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.

[Hol05] Andreas Holzinger. Usability engineering methods for software developers. *Commun. ACM*, 48(1):71–74, 2005.

[IH01] 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.

[JK94] 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.

[JK96] Bonnie E. John and David E. Kieras. The GOMS family of user interface analysis techniques: comparison and contrast. *ACM Transactions on Computer-Human Interaction*, 3(4):320–351, 1996.

[Lan] Michele Lanza. Codecrawler. `http://www.iam.unibe.ch/~scg/Research/CodeCrawler/`.

[Mar05] Philippe Marschall. Detecting the methods under test in Java. Informatikprojekt, University of Bern, April 2005.

[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.

[Rob04] Romain Robbes. Browse Unit: Integrating SUnit into the Smalltalk Browser, 2004. http://minnow.cc.gatech.edu/squeak/3113.

[WD03] Roel Wuyts and Stéphane Ducasse. Unanticipated integration of development tools using the classification model. *Computer Languages, Systems & Structures*, 30:63–77, 2003.

[Wes05] Frank Westphal. Junit 4.0, July 2005. `http://www.frankwestphal.de/JUnit4.0.html`.