RWTH AACHEN UNIVERSITY

Reducing mental context switches during programming

Supporting Code Comprehension using Semantic Links in Software Development Artefacts

Diploma Thesis at the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Henning Kiel

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Lichter

Registration date: Oct 21, 2008
Submission date: Jun 02, 2009

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, June 2nd, 2009
Henning Kiel

Contents

List of Figures

Listings

Abstract

Test cases can be a key artifact for program comprehension. By identifying and presenting test cases relevant to the source code currently worked on by the user, program comprehension can be improved. Especially in test-driven development this can be of great use as test cases already serve as low-level specification of the implementation and are always kept up-to-date.
In the work presented here, a prototype was realized implementing above idea as a plugin for a widely used text editor. The given prototype was evaluated in a user study. While no impact on program comprehension was found, the plugin improved navigation between source code and test cases.

Überblick

Testfälle können ein wichtiges Artefakt für Programmverständnis darstellen. Durch Identifizierung und Darstellung von zum aktuell bearbeiteten Programmcode relevanten Testfällen kann Programmverständnis verbessert werden. Besonders für Projekte, in denen Test-driven development angewendet wurde, kann dies von großem Nutzen sein. In diesem Fall dienen Testfälle bereits als Spezifikation der Implementierung und sind immer aktuell.
In der vorgestellten Arbeit wurde ein Prototyp entwickelt, der obige Idee als ein Plugin für einen verbreiteten Texteditor realisiert. Dieser Prototyp wurde in einer Benutzerstudie evaluiert. Es wurde keine signifikante Auswirkung auf Programmverständnis festgestellt. Das Plugin verbesserte jedoch die Navigation zwischen Quellcode und Testfällen.

Acknowledgements

First, I would like to thank my parents Susanne and Henry Kiel for giving me the opportunity to study Computer Science.

Thanks to Prof. Dr. Jan Borchers, Jonathan Diehl, and all the people at the Media Computing Group for their support and the nice working environment.

Special thanks to Nan Mungard and Mathias Funk for their motivation and insistence. This work would not have been possible without you!

Conventions

Throughout this thesis we use the following conventions.

Text conventions

Definitions of technical terms or short excursus are set off in coloured boxes.

EXCURSUS:
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition:
Excursus

Source code and implementation symbols are written in typewriter-style text. If not noted otherwise, all source code is written in Ruby.

```
myClass
```

The whole thesis is written in American English.

Chapter 1

Introduction

Unfamiliar code is a key obstacle for programmers

Most programmers have to work with unfamiliar code. Reasons for this include using third-party libraries, code created by other team members, or even own code created long ago. Especially during software maintenance, these characteristics are commonly found. The passed time since initial development in many cases means that the programming team has changed and the original programmers are not available anymore.

Bug fixing in unfamiliar code involves many different activities

Determining the cause of a bug and creating a fix involves many different and challenging activities. Presented with a completely unknown program, the programmer first has to comprehend the general architecture. She can then drill down into specific parts of the program she deems as possible candidates for the location of the bug. She has to understand how these parts interact with the rest of the program to be able to pinpoint the unwanted behavior and also to formulate a fix which does not corrupt existing behavior.

Programmers constantly switch between different documents

During these activities a programmer constantly switches between different documents like external documentation (architecture diagrams, requirements specifications) or source files and their corresponding test cases. Because the investigated program is unknown, searching relevant documents is done by trial and error. This is not a very efficient method for large software projects, making tool support for this activity necessary.

Certain process models ensure availability of up-to-date test cases

The available development artifacts and their structure depend on the underlying software process model. The software process model also influences how the documents are kept up-to-date during development and maintenance. Test-driven development (TDD) is a programming model in which code is only changed after corresponding test cases have been written. By employing this model, one makes sure that the test case artifacts are always up-to-date with regard to the source code.

Test cases can act as example code

Such an existing test suite can improve program comprehension during the software maintenance phase. The programmer in above example can use test cases to better understand how parts of a program have to be initialized or interact with each other. The test cases act as a kind of example code for different parts of the program.

The approach presented in this thesis tries to facilitate the use of test cases to improve program comprehension. This is done by presenting the programmer a filtered list of test cases relevant to the currently inspected source code. It was assumed that search efforts would decrease and that relevant information could be identified more easily.

In this thesis, I present above idea and explain how it can be integrated into projects employing TDD. A working prototype is implemented which integrates into a widely-used text editor. This prototype is evaluated with a group of student and professional developers. After discussing the results of this evaluation, possible ideas for further investigation and future work are outlined.

1.1 Structure

- In Chapter 2—"Theory" the theoretical basis of this thesis is explained. A definition for program comprehension is given and different measures are presented how to assess the level of program comprehension. Afterwards, the general concept of software testing is explained. An introduction to test-driven development is given and the main differences to software testing are highlighted.

- In Chapter 3—"Related Work" I present existing work in the fields of program comprehension and test-driven development. Several studies are briefly discussed which show that TDD is a valuable programming model. Finally, different tool prototypes to improve program comprehension are shown.
- In Chapter 4—"Prototype" the idea for a novel tool to support program comprehension is presented. The underlying idea is to provide the user with a list of existing test cases which are relevant to the currently inspected program part. After stating the requirements for a prototype, I explain the prototype implementation.
- In Chapter 5—"Evaluation" a study to evaluate the prototype is presented.
- In Chapter 6—"Discussion" the results obtained in the user test are discussed and put into context regarding the initial goals of the prototype.
- Finally, in Chapter 7—"Conclusion" the obtained results are summarized and an overview of possible future work is given.

Chapter 2

Theory

This chapter lays the theoretical basis for the work presented in this thesis. First, the concept of program comprehension is explained along with its importance for software engineering tasks. Next, a general introduction to software testing is given. Finally, the test-driven development (TDD) programming model is explained and its differences to classical software testing highlighted. To document the practical relevance of TDD two Open Source projects employing TDD as their programming model are presented. These projects are also used for the evaluation of the prototype.

2.1 Program Comprehension

Program comprehension is critical during maintenance

There are many situations in which programmers have to work with unfamiliar code, for example when working in a team, integrating external libraries, or during maintenance. A key challenge in all these cases is program comprehension. Especially during software maintenance it plays a major role. According to Arthur [1988] program comprehension can take up to 90% of the total time spent during software maintenance. Several psychological models have been proposed on how a programmer understands code and many tools have been developed to support program comprehension.

Definition of program comprehension

Program comprehension is defined in Koenemann and Robertson [1991] as the process of understanding program code unfamiliar to the programmer. Some reasons for the challenge of program comprehension are given in Layzell and Macaulay [1990] as changing team members as well as lack of documentation and communication in the team.

Measurements of program comprehension

The level of program comprehension can be measured using different methods: In maintenance tasks participants have to add features or fix bugs while the task completion time is measured. Another possibility is to let participants correctly fill in missing parts of the code and to evaluate the correctness of their solution. A commonly used method for small bodies of code is to ask participants to recall a chunk of code memorized in an earlier stage and evaluating the completeness. These and other program comprehension measurements are described and evaluated in Dunsmore and Roper [2000].

Psychological models of program comprehension

An overview of current research on code comprehension is given by Von Mayrhauser and Vans [1995]. They identify six different cognition models and categorize them according to how they incorporate programmer knowledge, mental representations and mental processes. They identify several key obstacles for empirical validation, foremost that most models have not been validated with experiments using large code bases. They criticize the small code bases (less than 900 SLOC) used in experiments, since it is not clear how they apply to real work situations. Finding experienced programmers willing to participate in such experiments is another obstacle.

Definition: *SLOC*

SLOC:
Source lines of code (SLOC) is a software metric to measure the size of a software program. In Conte et al. [1986] a line of code is defined as any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line.

Common problems for program comprehension

A qualitative study of code comprehension was conducted by Sillito et al. [2006]. They observed professional programmers during maintenance tasks. The programmers were instructed to think-aloud, i.e. to articulate their current thoughts during the task. It was found that different programmers asked similar questions. These questions were grouped in four categories:

- Finding initial focus points. These types of questions are mainly asked by newcomers with mostly no existing knowledge about the program. The questions revolve around finding entry points into the subgraph of related program elements related to a certain type of functionality. Typical questions are "Where is the code involved in the implementation of this feature" or "Which type represents this domain concept".

  Where to begin

- Building on above focus points. After an initial focus point has been found, using the type of questions in this category, developers try to get a grasp of the program elements related to the entry point. Common questions involve the relationships between different program elements like "Who implements this interface" or "Where is this method called".

  Finding related program elements

- Understanding a subgraph. With questions from this category, programmers try to get a better understanding of how a certain subgraph of elements inside the program works. The questions are about dynamic behavior, for example "Which execution path is taken in this case" or "What is the correct way to use or access this data structure?".

  Understanding a set of related elements

- Questions about groups of subgraphs. The questions in the last category are about the relationships between parts of the program. Typical questions are "What is the difference between these similar parts of the code" or "What will the total impact of this change be".

  Understanding how parts of the program work together

Project documentation can help with program comprehension

There are several ways to improve program comprehension during maintenance. As stated in Sillito et al. [2006] frequent questions include the location of a certain feature and how different parts of the program work together. Documentation created during the initial development can provide valuable information to resolve these questions. Use Case and Class diagrams, for example, give a high-level overview of the functionality of the program and its architecture. In Tenny [1988] it was shown that code comments significantly improve readability and hence program comprehension.

Project documentation is often outdated

In reality, though, documentation like UML-diagrams or code comments are often not available during maintenance or outdated. Code often evolves faster than its documentation. A programmer might change some code lines but not update the corresponding code comments. Or the interface of a class is changed without updating the Class diagrams created in earlier development phases. Consequently effective use of such information during maintenance is hard.

Program comprehension improves with tool support

Therefore other approaches incorporate tool support to improve program comprehension. There exist several ideas and prototypes in this area. In Section 3.2—"Tools for Program Comprehension" I present ideas relevant to this thesis.

2.2 Software Testing

Software testing is an essential phase in every software development process. In most software development processes it happens after the implementation phase. The motivation for software testing is to raise the quality of the software.

Definition of software testing

Myers [1979] defines software testing as the process, in which a program is executed with the goal to find errors. A test case is successful if it has detected an error. If all test cases pass, the probability is higher that the program is correct according to the specification.

Correctness cannot be proved with test cases

In general, software testing cannot prove the correctness of a program, though. For most non-trivial programs the set of possible inputs is so large that creating tests for all possible inputs is either impossible or too expensive.

Selection of test cases is a crucial task

Therefore the selection of appropriate test cases is such a crucial and difficult task. There exist several methods for creating test cases, for example selecting test inputs using equivalence classes or finding additional test cases by analyzing the code coverage of existing test cases.

2.2.1 Test Process

Most formal testing is done in a way similar to the following process model. In order to create a test case, the tester first has to determine the expected output according to the specification. Ideally the expected output is derived automatically, for example from a formal specification. However, in most cases this has to be done manually by the tester.

TEST CASE:
A test case contains a specification of test input and the expected output. It is identified by a unique name. Test cases concerning the same functionality aspect are usually organized in test suites.

Definition:
Test case

The next step is to generate inputs which should result in pre-defined output. This step also involves setting up the test environment, to automate the test. The tester then runs the test, compares the resulting output with the expected output and creates the test protocol.

If the expected output does not match the generated output, it still does not mean that an actual program error has been found. One should first check if the expected output is correct. If a real program error has been found, it will be passed back to the development team to fix it.

2.2.2 Classification of Test Cases

Process to create test cases

Test cases can be classified in several dimensions. In the following, a selected set of criteria is presented. A more thorough classification can be found in Ludewig and Lichter [2007]. One aspect is if the test cases are created as part of a formal test process:

- *Execution tests* and *throw-away-tests*: These tests are normally done by the programmer. Execution tests just check if the program compiles and starts without an error. Throw-away-tests also perform basic checks if the generated output matches the expectation of the tester. Both tests are neither documented nor created according to a formal process.
- *Systematic tests*: These tests are generally not created by the developers themselves. As described above, the test input and expected output is generated from the requirement specification documents. The whole test is documented, the results are logged and the test is easily repeatable.

Information used to create test cases

Another aspect is how they are created:

- *Black-box test* cases are created without any knowledge of the actual implementation of the program. In this case, the tester normally creates different tests to cover all functionality and different input and output classes, according to the specification.
- *White-box test* cases. When creating white-box test cases, the inner workings of the program are known. Test cases are created until defined code coverage goals are fulfilled, as explained in Section 2.2.3—"Code Coverage".

Scope of test cases

Test cases can also be differentiated using the scope of the test case.

- A *unit test case* only tests a single part of the program. Dependent parts are only simulated.
- An *integration test case* focuses on the interaction of several program components.
- A *system test case* finally tests the entire program against the specification. A special form are acceptance test cases which test if the program is in accordance with the client expectations and the contractual obligations of the developer are met.

A special test form is the *regression test*. It is used to ensure that existing behavior is not corrupted in new releases of the software. This type of test should ideally be completely automatic and integrated into the build process, so that changes in behavior are detected at the earliest moment possible.

Regression tests to not corrupt existing behavior

2.2.3 Code Coverage

Code coverage analysis is used to measure how much of a program's code is tested by a suite of white-box tests (see Ludewig and Lichter [2007]). There exist several different types of code coverage. Common types are:

- *Statement coverage* is achieved when every statement of the program has been executed once.
- *Decision coverage* is reached when at every branch decision in the program every possible branch has been executed once.
- *Condition coverage* is reached when every boolean term in all branches has evaluated to both true and false once.

In practice, not even the weakest code coverage type, statement coverage, is always achieved. Ludewig and Lichter [2007] give as possible reasons, that some code parts are only used to check for errors in other parts of the program or are dependent on circumstances which are difficult to reproduce in tests. Project manager who want to define a minimum level of coverage usually do so by defining a percentage of statements, decisions or conditions which should be covered by the test cases.

Complete code coverage is rarely reached

2.3 Test-Driven Development

Test-driven development (TDD) originated in the Test-first practice of the Extreme Programming movement. Test-first

TDD is a programming model

is a programming model where developers write test cases and only afterwards start with the actual program implementation. Today TDD has developed into an independent field as its usage is not necessarily dependent on other Extreme Programming practices.

Tests drive code creation

Contrary to the role of the test cases described in Section 2.2—"Software Testing", the primary role of the test cases in TDD is to specify what program part has to be implemented or changed next. Kent Beck describes this in Beck [2002] as follows:

> We drive development with automated tests, a style of development called test-driven development (TDD). In test-driven development, we
>
> - Write new code only if an automated test has failed
> - Eliminate duplication

The continuously extended automated test suite determines the next program part to be developed, as described in the first point. A new feature or requirement is specified with a set of new tests, which initially all fail. The programmer then writes code which ideally should not contain more functionality than necessary to pass all tests.

Tests help during refactoring

The second point in Beck's citation implies code refactoring, which should be done when all tests pass. By continuously testing, the programmer has more confidence that the changes made to the implementation do not corrupt existing behavior.

Tests act as a low-level specification

TDD is an evolutionary development process with very small iterations. During the repeated creation of tests and code a programmer discovers new requirements for the program, which are specified with failing tests, too. Thus, one of the advantages of TDD according to Bhat and Nagappan [2006] is that the whole test suite represents a kind of executable specification for the low level design of the program.

2.3.1 Process Model

The typical iterative workflow of TDD according to Beck [2002] is shown in Figure 2.1. This workflow is also referred to as the "red-green cycle". During the red part of the cycle one or more test cases fail. Code is changed until all test cases are passing and the green part of the cycle is entered. While in the green part of the cycle, code is refactored to remove duplication or to improve the design.

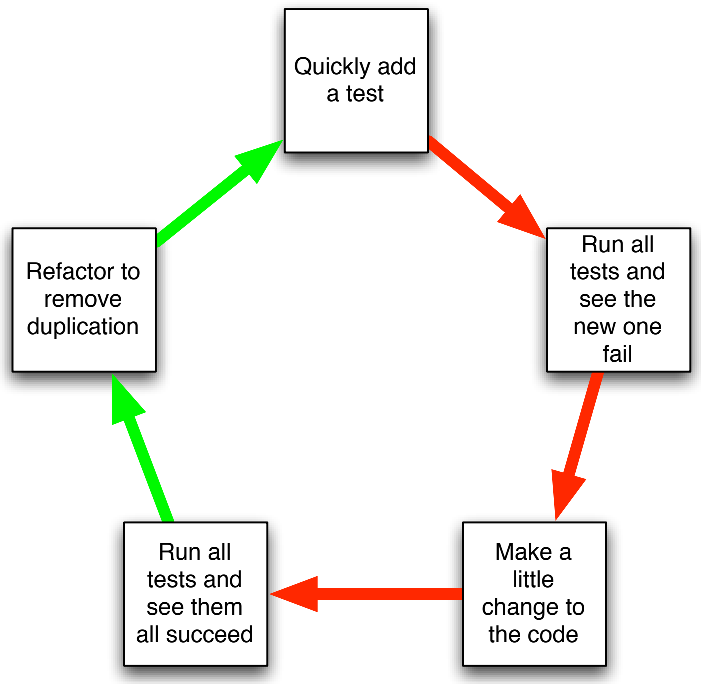

Quickly add a test
Run all tests and see the new one fail
Make a little change to the code
Run all tests and see them all succeed
Refactor to remove duplication

Figure 2.1: The iterative workflow of TDD as described in Beck [2002]

The first step in the TDD workflow is to write a failing test case. The ideal test case should be small so that it clearly defines what code to write next. The finished test case will initially fail, as the relevant code has not been written yet.

Initially a new test always fails

The next step is to write the least amount of code to make the test case pass. Beck notes that it can be surprising what little code is needed to make a test case pass. Take for example the simple test case in Listing 2.1. This test case can

Code is only written to pass tests

be passed with the naive implementation in Listing 2.2.

```
def test_length_of_new_list_should_be_zero
  list = List.new
  assert_equal 0, list.length
end
```

Listing 2.1: Test case example

```
class List
  def length
    return 0
  end
end
```

Listing 2.2: Simplest code to fulfill the above test case

Refactoring is only done while all tests pass

After doing the necessary changes to make the test suite pass again, the next step is to reconsider the resulting implementation and make changes, if necessary. During this refactoring the existing test suite helps to make sure that the behavior of the program element is the same.

The implementation of a list in Listing 2.2 is correct with regard to the defined test case from Listing 2.1, but it is obvious that such a list is essentially useless. In order to create a usable list implementation, more iterations of creating test cases and writing the corresponding code would be done until the required functionality is implemented.

This iterative creation of failing test cases and the code to make them pass is essential to the TDD model. The automated test suite drives development and potential changes in behavior during refactoring can be detected by the test cases.

2.3.2 Classification

Using the classification of software tests presented in Section 2.2.2—"Classification of Test Cases" the tests generated during TDD can be categorized as follows:

- Systematic. The test cases are created as part of the TDD programming model. They are created by the developers themselves, though, as they are an integral part of the implementation phase.
- A form of white-box test, although the relationship to the implementation is reversed, meaning that the implementation is based on the construction of the test.
- Unit tests, as they are designed to guide the implementation of a specific part of the program.
- Regression tests. When the implemented code is finished according to the related tests, these tests are then used as regression tests, to make sure that later refactoring does not corrupt existing behavior.

2.3.3 Test-Driven Development in Practice

TDD is employed in many projects

Today TDD is used in many projects. Williams et al. [2003] and Bhat and Nagappan [2006] describe their experiences applying TDD in industry environments at IBM and Microsoft, respectively. Both studies are presented in more detail in Section 3.1—"Evaluating the Effectiveness of Test-Driven Development".

TDD is also employed in open source projects. In the following, two example open source projects using TDD are presented. These examples are documented in detail as they are used for the prototype evaluation in Chapter 5—"Evaluation".

Example: RSpec

RSpec is a framework for creating TDD tests

RSpec[1] is an Open Source library to support test-driven development of Ruby programs by providing developers with functionality to easily create and execute test cases. It was created by Steven Baker in 2005. In the mean time, development has been passed to David Chelimsky, Pat Mad-

[1]http://rspec.info/

doc, Aslak Hellesøy and other contributors. RSpec consists of about 16.000 SLOC. About two thirds are used for tests.

Naming conventions used in existing TDD frameworks can lead to confusion

RSpec was inspired by an article published by Astels [2005]. In this article Astels criticizes the common notion that TDD represents some form of software testing to find errors as explained in Section 2.2—"Software Testing". In contrast the primary goal of test cases in TDD is to specify behavior of code. As a possible reason for this misconception Astels mentions the common practice to use terms like test case, test suite or prefixing test case methods with "test_".

RSpec's naming conventions better convey the spirit of TDD

RSpec is a library which facilitates the use of TDD. It provides functionality to specify the behavior of code and to define expectations, which the implementation has to fulfill. The main difference to other TDD frameworks is the different naming convention. Instead of talking about tests and test suites, RSpec uses names like behavior, examples and expectations. Furthermore, RSpec tries to facilitate code which resembles plain english sentences.

```
describe Bowling do
  it "should score 0 for gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should == 0
  end
end
```

Listing 2.3: RSpec example

The main elements of RSpec documents are shown in Listing 2.3. A "describe" block is used to organize examples related to the same functionality. Here it contains the description of the behavior for objects of the "Bowling" class. An example of this behavior is defined by using the "it" method. This method receives a string as a parameter describing the central expectation. This string is only used when printing failed or passed examples. According to the convention, the string should be constructed such that it forms a readable, english-like sentence starting with the word "it". This sentence forms the human readable requirement for the program.

The first line in the do-end-block prepares the object we want to test. In the second line we put the object into the state for which we want to set an expectation. Finally, in the last line we construct the expectation. The example will fail if this expectation is not met or if a runtime error occurs.

Most of RSpec's functionality is implemented in such a way that it more or less resembles an english sentence, so that its semantic should be clear even to readers not familiar with RSpec. In Listing 2.4 some examples for this are given.

RSpec code resembles english sentences

```
hash.should have_key("some key")
variable.should_not be_nil
lambda {
  object.method_call
}.should change(object, :value).by(2)
```

Listing 2.4: Different ways to set expectations in RSpec

Today RSpec is one of the most used frameworks for TDD in the Ruby world. Some examples are:

- Merb[2] , a framework for creating web applications.
- The rubyspec project[3] , which aims to create an executable specification of the Ruby language.
- Mephisto[4] , a web application to create blogs.

Example: Liquid

Liquid[5] was created by jadedPixel Inc. in 2006 and published under an Open Source license. It is a library to employ templates in a web application without compromising the safety of the server. Liquid has about 3.500 SLOC, of which two thirds are used for tests.

Liquid provides security and user-generated design

[2]http://merbivore.com/
[3]http://rubyspec.org/
[4]http://mephistoblog.com/
[5]http://www.liquidmarkup.org/

Liquid was created out of the necessity to allow users of web application to create and edit templates by themselves. An example is the creation of a new design for a blog or web shop. Such a template normally consist of mostly HTML, with some instructions in the templating language. When a webpage is opened, these instructions are executed to fill in the HTML code for the dynamic parts of the page.

As arbitrary users of such a web application should be able to edit these templates, the templating engine must provide a clearly defined interface to the host application, but should disallow any access to APIs reserved for internal use.

Using Liquid a clearly defined public API is exposed

To allow for this isolated interpretation of templates, Liquid defines a simple programming language which is interpreted when loading a template. Liquid also offers an interface so that developers of web application can easily define the API for their application.

The example template in Listing 2.5 is used to generate a list of products in HTML. The variable `products` is assigned by the host application. Using a simple for-loop, the template outputs the name and description for each product. If a product contains a link to an image, the template will create an image tag as well.

```
{% for product in products %}
<h1>{{ product.name }}</h1>
<p>{{ product.description }}</p>
{% if product.image %}
<img src="{{product.image}}"/>
{% endif %}
{% endfor %}
```

Listing 2.5: A simple Liquid template

```
products
  1
    name = "Bike"
    description = "long text"
    image = "/bike.jpg"
  2
```

```
      name = "Car"
      description = "longer text"
      image = NULL
```

Given the Liquid template in Listing 2.5 and above variable assignment, one can render the template which would result in the following HTML snippet:

```
<h1>Bike</h1>
<p>long text</p>
<img src="/bike.jpg"/>
<h1>Car</h1>
<p>longer text</p>
```

Liquid's programming language consists of two main components:

- Commands included in {% and %}. These commands are used for flow control, assignment or inclusion of other files, but produce no visible output in the resulting rendered output.
- Commands included in {{ and }}. During rendering these commands are replaced with their value according to the current variable assignment.

Liquid is currently used in several web applications. Some examples are:

- Shopify[6] , a fully hosted eCommerce solution.
- Mephisto[7] , a web application to create blogs.
- 3sellers[8] , a CMS and eCommerce solution.

[6]http://www.shopify.com/
[7]http://mephistoblog.com/
[8]http://www.3sellers.com/

Chapter 3

Related Work

In the following, selected studies about the effectiveness of test-driven development are presented. Most studies find a decreased defect rate when employing TDD in a software project. Furthermore, different existing approaches to improve program comprehension with tool support are shown.

3.1 Evaluating the Effectiveness of Test-Driven Development

TDD lowers defect rates

There is previous research on the effectiveness of TDD with both students and professional programmers. Most studies found a smaller defect rate compared to programming models in which no tests are required, though in many cases an initially lower productivity was found.

Comparing TDD with classical development

Muller and Hagner [2002] conducted one of the first experimental studies about the effectiveness of TDD. Students were divided into two groups. One group developed according to the TDD model, the control group according to the classic programming model. All participants had to develop the main class for a graph library. They were given method declarations and had to implement the respective method body. The experiment consisted of two phases. In the first phase, the participants were allowed to work freely, until they considered their implementation to be correct. In the second phase, both groups had to fix bugs in their code from the first phase discovered by a suite of acceptance test cases. These acceptance tests had been developed a priori by the experimenters. The participants did not know about the second phase beforehand.

Dependent variables were the total time for finishing both phases, the amount of bugs discovered in the second phase, and the amount of wrong calls to existing methods. A call to an existing method in the graph library was considered wrong when it caused compilation or runtime errors. The environment with which the participants worked was instrumented to log these types of errors. The participants were not instructed about this logging. The wrong calls were divided into calls which were only used wrongly once (a) and calls which were repeatedly wrong (b). The latter two variables were used to judge the level of program comprehension. The authors argue that participants which better comprehend the existing code of the graph library would make fewer mistakes reusing existing code, hence committing fewer repeated errors.

A statistically significant difference between the two groups in the variable b was discovered. While both groups had a similar number for a, the TDD group had a significantly lower number of repeated wrong calls. The authors argue that a possible reason for above difference is, that, through repeated testing, the participants learned more quickly about the existing interfaces in the graph library, and thus introduced less repeated errors.

TDD experiences in the industry

Williams et al. [2003] present one of the first TDD study with professional programmers. An IBM division developing device drivers for various platforms had to release a new version. All earlier releases were developed using "ad-hoc unit testing", that is each programmer developed a set of unit tests after having written code. Most of these unit tests were not even automated. For the new version, management introduced the TDD programming model. Afterwards, they compared the defect rates (measured in defects per SLOC) of new release and old release.

TDD significantly lowers defect rate

They discovered that the version developed with TDD had a 40% lower defect rate. The authors also noted that changes in requirements in the middle of the project did not have a big impact. As the authors highlight, this was no formal experiment, but a case study. However, this shows that TDD can have significant advantages in an industrial environment.

Test first vs. test last

George and Williams [2003] present an experiment with professional Java programmers within three different companies. The programmers were divided into two groups. One group used the TDD programming model. The other used the classic waterfall model with tests, which were created after the programming phase. The assignment was to develop a bowling program according to a given set of requirements. There was no time limit for task completion. When the participants regarded their program correct, the experimenter measured the defect rate according to a set of previously created black-box tests.

Dependent variables were the time to finish the program, the number of successful black-box tests and qualitative feedback from the participants about TDD as a programming model. Furthermore, the quality of the tests created by the TDD group was measured using a code coverage tool. The tests were rated as having a higher quality if they covered more of the actual source code of the program.

It was shown that the programs created by the TDD groups had an 18% lower defect rate, as measured by the number of passed black-box tests. This difference is statistically significant. The authors take this as a confirmation of the hypothesis that TDD usage results in higher code quality. The measured code coverage of the test cases in the TDD group was very high, surpassing even the industry average.

TDD conveys better sense of correctness of a program

At the same time, the TDD group needed 16% more time to finish the program. This correlates with the higher quality noted above. While the authors admit that a longer completion time might be a reason for the higher quality, they emphasize that each group was free to hand the program to the experimenter whenever they felt it was finished. The control group was free to use more time in order to create a program with a lower defect rate. The authors argue that the TDD group seemed to have a better understanding of the current correctness of their program.

3.2 Tools for Program Comprehension

As discussed in Section 2.1—"Program Comprehension", program comprehension is a key obstacle in large projects and especially in maintenance tasks. Several existing software tools address this issue. In the following, a selection of different approaches is presented which focus on improving program comprehension during maintenance.

3.2.1 Software Reconnaissance

Wilde and Scully [1995] present a technique called "Software Reconnaissance". Its goal is to help locating the implementation of a particular feature in the source code.

Finding answers to "Where to begin"-questions

The underlying idea is that a programmer prepares several test cases and some of these test cases use the feature in question, some explicitly do not. These test cases are then run by a code coverage tool, which tracks the code paths used by each test case.

If a is the set of code paths for test cases using the feature, and b the code paths for test cases not using the feature, the tool subtracts b from a, resulting in a set of paths which likely belong directly to the feature. The tool presents these code paths next to a list of functions and files used.

Wilde and Casey [1996] document the experience of using the Software Reconnaissance tool in a number of different programs. In most cases, the number of files suggested by the tool for further inspection was about one sixth the total number of files in the project.

3.2.2 Automatic Generation of Suggestions for Program Investigation

Finding a set of related program elements

Robillard [2005] presents an algorithm which, given some program elements like methods or variables, can be used to find other related elements. The goal is to improve understanding of the possible impact of code changes. The user interface of an Eclipse plugin implementing this algorithm can be seen in Figure 3.1

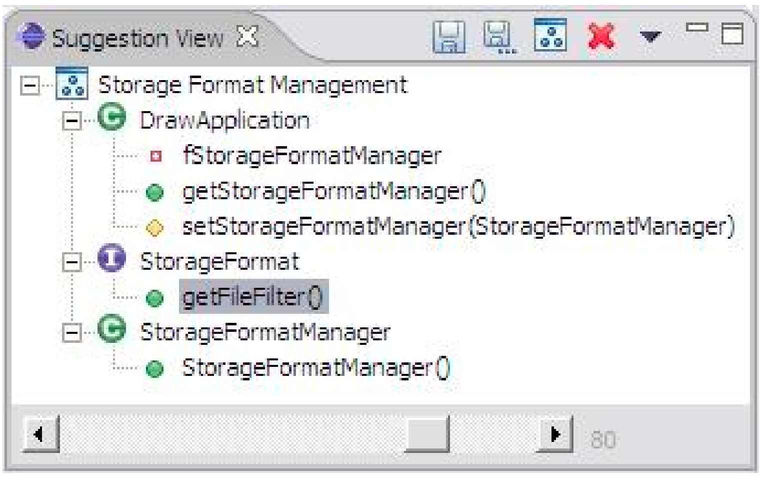


Figure 3.1: Eclipse Plugin with automatically generated suggestions for program investigation

The programmer has to start by defining a set of program elements she is interested in. Using static code analysis, the algorithm finds other elements which are in some relation to the initially specified elements. These relations can be the use of a variable or a method call, for example. To order related elements, the algorithm calculates a measure of relevancy using the following two criteria:

- Specificity: An element y is relevant in respect to a set of elements I, if every element in I which is related to y is only related to few other elements outside of I, and if y is only related to few other elements.
- Reinforcement: An element y is relevant with respect to a set I, if most other elements related to y are also in I.

The authors evaluated the quality of the algorithm's results in a study. They selected a function inside an Open Source program and presented the suggestions to two students which had a good knowledge of the used program. The students had to judge each suggestion if it helped in comprehending the function or not.

Only 26% of the algorithm's suggestions were judged as being not relevant by the experts. This number was even lower when only considering the suggestions rated most relevant by the algorithm.

3.2.3 Evolutionary Annotations

Information about why a piece of code developed into its current state

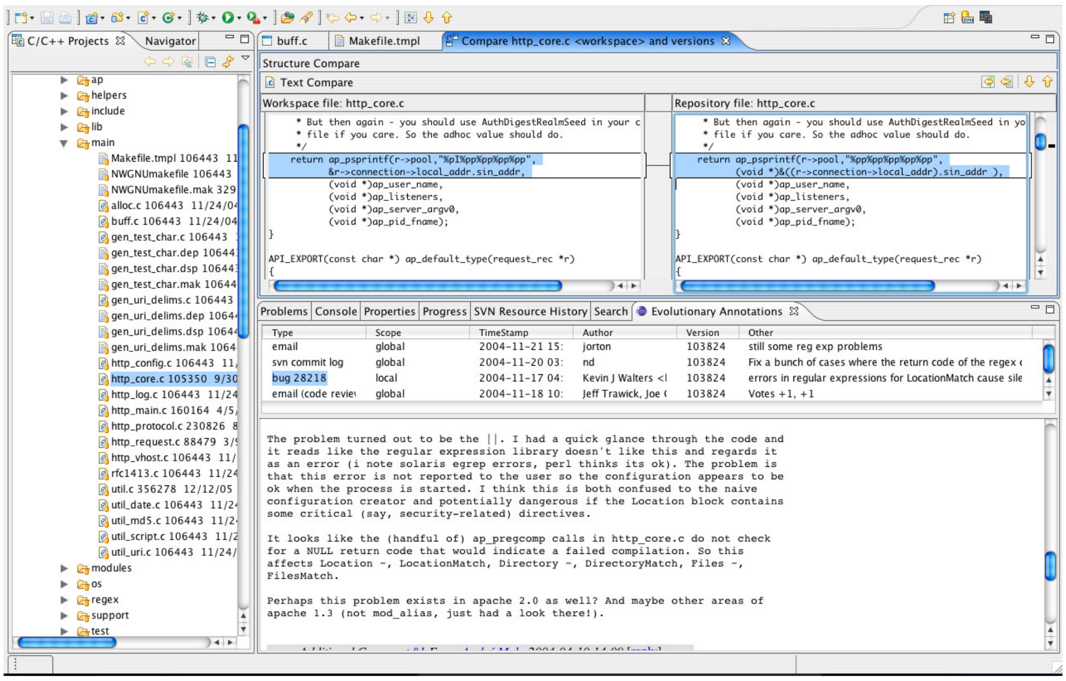
German et al. [2006] present Evolutionary Annotations, which are contextual information computed from artifacts created during most software projects. Contrary to documentation or code comments, Evolutionary Annotations help to understand the evolution of code over time. Especially during maintenance, they can be used to determine why certain code parts have been developed the way they exist in the released version.

Evolutionary Annotations are generated from artifacts like bug reports, commit messages, TODO comments in the code or messages on mailing lists.

Heuristics to relate different information sources

The authors suggest different heuristics to relate such artifacts to a code line or file. For example, when discussing a patch for an Open Source project on a mailing list, many developers include a listing of changes in question. By searching in the code for the given changes, one can relate such discussions to code location.

A problem is still how to weight the different annotations. Some form of weighting has to be done, as for example a central file in a project will accumulate many commit messages over time. Depending on the maintenance work to be done, only a subset of those commit messages might be relevant.



Workspace file: http_core.c

```
     * But then again - you should use AuthDigestRealmSeed in your c
     * file if you care. So the adhoc value should do.
     */
    return ap_psprintf(r->pool,"%pI%pp%pp%pp%pp",
           &r->connection->local_addr.sin_addr,
           (void *)ap_user_name,
           (void *)ap_listeners,
           (void *)ap_server_argv0,
           (void *)ap_pid_fname);
}

API_EXPORT(const char *) ap_default_type(request_rec *r)
{
```

Repository file: http_core.c

```
     * But then again - you should use AuthDigestRealmSeed in yo
     * file if you care. So the adhoc value should do.
     */
    return ap_psprintf(r->pool,"%pp%pp%pp%pp%pp",
           (void *)&((r->connection->local_addr).sin_addr ),
           (void *)ap_user_name,
           (void *)ap_listeners,
           (void *)ap_server_argv0,
           (void *)ap_pid_fname);
}

API_EXPORT(const char *) ap_default_type(request_rec *r)
{
```

Type	Scope	TimeStamp	Author	Version	Other
email	global	2004-11-21 15:	jorton	103824	still some reg exp problems
svn commit log	global	2004-11-20 03:	nd	103824	Fix a bunch of cases where the return code of the regex (
bug 28218	local	2004-11-17 04:	Kevin J Walters <I	103824	errors in regular expressions for LocationMatch cause sile
email (code revie\	global	2004-11-18 10:	Jeff Trawick, Joe (	103824	Votes +1, +1

```
The problem turned out to be the ||. I had a quick glance through the code and
it reads like the regular expression library doesn't like this and regards it
as an error (i note solaris egrep errors, perl thinks its ok). The problem is
that this error is not reported to the user so the configuration appears to be
ok when the process is started. I think this is both confused to the naive
configuration creator and potentially dangerous if the Location block contains
some critical (say, security-related) directives.

It looks like the (handful of) ap_pregcomp calls in http_core.c do not check
for a NULL return code that would indicate a failed compilation. So this
affects Location ~, LocationMatch, Directory ~, DirectoryMatch, Files ~,
FilesMatch.

Perhaps this problem exists in apache 2.0 as well? And maybe other areas of
apache 1.3 (not mod_alias, just had a look there!).
```

Figure 3.2: List of Evolutionary Annotations for selected code

3.2.4 Whyline

Ko and Myers [2008] present an approach to find the location of errors in Java programs called the Java Whyline. This debugging environment allows developers to ask questions about the output of the program. Whyline then searches through data generated during program execution to identify locations in the execution trace which resulted in the specific output.

Determining how program output was created

Whyline works by instrumenting a Java program so it can trace its execution. The developer runs the program and executes the sequence of events leading to the error. In the background, Whyline continuously records the output, input events and the executed code.

After quitting the program, the actual Whyline interface is started. The developer can scroll through a timeline of program execution to reach the point where the unexpected behavior occurred. She then can select elements of the program's output and select possible questions suggested by

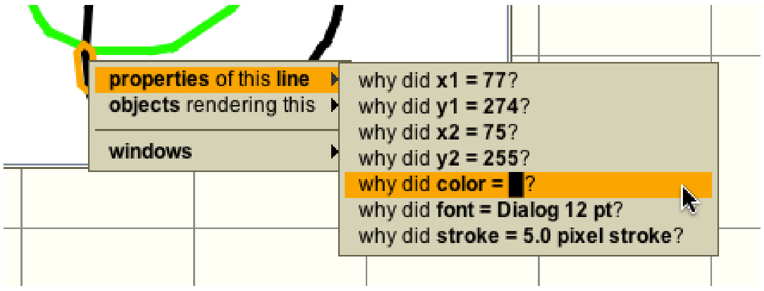


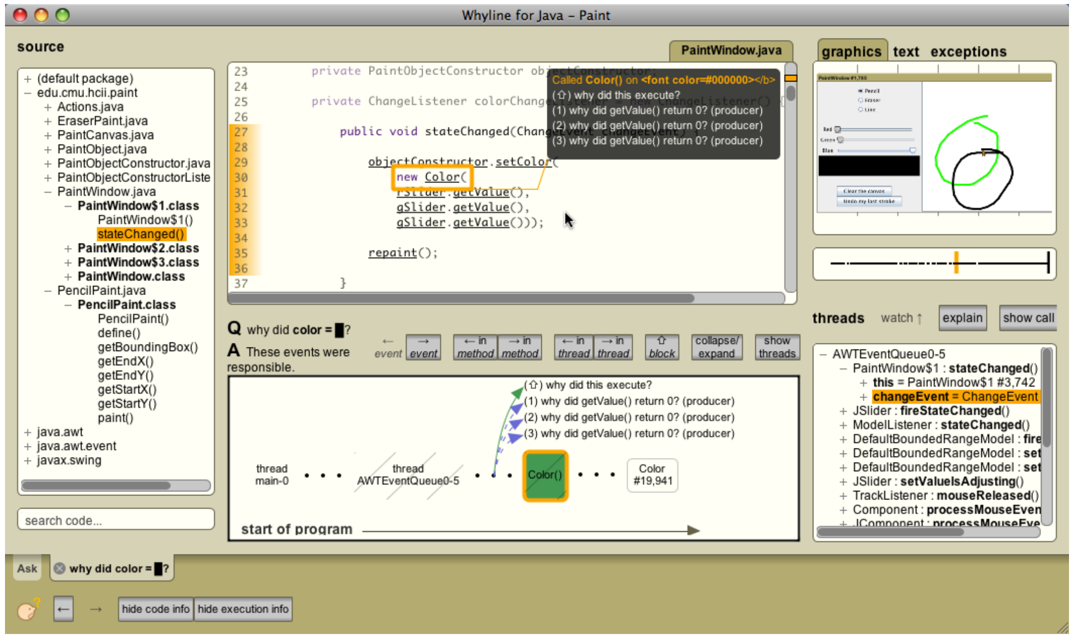
Figure 3.3: Proposed questions by Whyline for the canvas of a drawing program

Whyline, as seen in Figure 3.3. Whyline's capability to generate these questions depends on knowledge about the user interface types used. All basic Java output types like strings, graphic primitives or GUI widgets are currently supported.

The program displays information about the question with the user interface seen in Figure 3.4. It shows the timeline of events leading to the selected output in the bottom middle of the window. When selecting one of these events the corresponding source code is displayed above. Developers can also ask follow-up questions proposed by the tool.

In Ko and Myers [2009] the authors evaluate the Whyline debugging tool by comparing it to classic debugging done with breakpoints. Each evaluation group consisted of ten students. All participants had to solve an existing bug ticket of the Open Source Java program ArgoUML. The participants had to find the cause of the bug and propose a change.

The results show that the group using the Whyline tool had both a statistically higher success rate in finding and fixing the error as well as a statistically lower completion time. For the first task, the files viewed by the participants were also on average more relevant to the bug in question then for the control group. This indicates that the tool helps in quickly finding the correct place to correct an inspected bug.



```
private PaintObjectConstructor ob
private ChangeListener colorChang
    public void stateChanged(Chang
        objectConstructor.setColor(
            new Color(
            rSlider.getValue(),
            gSlider.getValue(),
            gSlider.getValue()));
        repaint();
    }
```

```
Called Color() on <font color=#000000></b>
(⇧) why did this execute?
(1) why did getValue() return 0? (producer)
(2) why did getValue() return 0? (producer)
(3) why did getValue() return 0? (producer)
```

Q why did color = █?
A These events were responsible.

```
(⇧) why did this execute?
(1) why did getValue() return 0? (producer)
(2) why did getValue() return 0? (producer)
(3) why did getValue() return 0? (producer)
```

```
+ (default package)
– edu.cmu.hcii.paint
  + Actions.java
  + EraserPaint.java
  + PaintCanvas.java
  + PaintObject.java
  + PaintObjectConstructor.java
  + PaintObjectConstructorListe
  – PaintWindow.java
    – PaintWindow$1.class
        PaintWindow$1()
        stateChanged()
    + PaintWindow$2.class
    + PaintWindow$3.class
    + PaintWindow.class
  – PencilPaint.java
    – PencilPaint.class
        PencilPaint()
        define()
        getBoundingBox()
        getEndX()
        getEndY()
        getStartX()
        getStartY()
        paint()
+ java.awt
+ java.awt.event
+ javax.swing
```

```
– AWTEventQueue0-5
  – PaintWindow$1 : stateChanged()
      + this = PaintWindow$1 #3,742
      + changeEvent = ChangeEvent
  + JSlider : fireStateChanged()
  + ModelListener : stateChanged()
  + DefaultBoundedRangeModel : fire
  + DefaultBoundedRangeModel : set
  + DefaultBoundedRangeModel : set
  + JSlider : setValueIsAdjusting()
  + TrackListener : mouseReleased()
  + Component : processMouseEven
  + JComponent : processMouseEve
```

Figure 3.4: Whyline interface for answering questions about program output

Chapter 4

Prototype

4.1 Motivation

Software maintenance is still one of the most important phases in a software's life cycle, and program comprehension, as explained in Section 2.1—"Program Comprehension" a key aspect. Program comprehension is often hindered by missing or badly maintained documentation. Documentation can exist in the form of external documentation like requirements specifications, API documentation, or architecture diagrams. Code comments can also be useful to understand specific parts of the code. As code evolves, both types of documentation are often left behind and not kept up to date.

Program comprehension hindered by badly maintained documentation

A project employing TDD as its development practice creates a big set of test cases. These test cases are automatically kept up-to-date, as one of the key factors in TDD is that any code change must not let test cases fail. As explained in Section 2.3—"Test-Driven Development", the test suite can be regarded as a form of executable specification.

Test cases are always up-to-date

Because the test cases are artifacts which are always kept up-to-date, they are an interesting candidate for use during software maintenance. For instance, they are already used in the Software Reconnaissance work by Wilde and Scully [1995] to locate the implementation of certain features. An-

Test cases are already used in software maintenance tools other example are the results of the user study performed in Muller and Hagner [2002], where the authors argue that continuous testing raises program comprehension of existing code as developers get more exposure of the existing code while writing new test cases.

Test cases as documentation

I think that even existing test cases can serve as a form of example code showing how to use different parts of a program. In each test case the object to be tested has to be instantiated, its connections to other relevant objects are set up, and some methods are executed. As the source code development is driven by those test cases, the actual usage of the objects or classes in the project is reflected.

Finding relevant test cases can be time-consuming

Since test cases are usually maintained in different files than the actual code, direct navigation between the code and the relevant test cases is often not possible. Some projects use a similar directory structure and filenames for the source code tree and the test case tree as a policy.

Differences between test and source file organization

But this does not work in every case. A programmer might decide to split the test case file for a given class into several files according to different functional aspects of the class. Or a class might be used in different contexts, and a programmer creates test cases for each context in different files.

Another problem which arises is that a single file can contain too many test cases. If a programmer is only interested in a certain functionality subset of a class or part of a program, it can be hard to find the relevant test cases.

I propose a new method to ease the navigation between the source code and the related test cases, and the discovery of relevant test cases. The underlying idea is to only show the test cases which use a given method during their execution and order them according to a relevancy measure.

Test cases are no substitute for program comprehension

A typical use case could be, that a programmer has already identified the class where the source of a bug is located. But as she is unfamiliar with the usage of said class it is not clear to her how to fix the bug. By changing the class she could corrupt other parts of the program depending on this class. The existing test suite can detect some unwanted side-effects. But to have a good understanding of how the class is used is still very important in order to create a fix which integrates well into the program.

To get a better feeling for how the class might interact with the rest of the program, the programmer can consult the list of relevant test cases which show how the class has to be instantiated and what other parts of the program it might affect.

4.2 Requirements

Based on the motivation for the tool, I defined several initial requirements and use cases for the tool which will be presented in the following.

Test case suggestions are shown for the selected method

Given a method in a source file, the tool shows all test cases which use this method during their execution. The test cases are shown ranked by relevance. Initially, the relevance will be a simple count of how often a test case has (directly or indirectly) called the method. This is determined by the statement coverage (see Section 2.2.3—"Code Coverage") for each test case.

Integration in existing editor

The tool is integrated into an existing editor, as it complements the functionality used for editing code. The list of relevant test cases is updated automatically while the programmer navigates through the source code.

The tool offers the possibility to easily switch between the source code and any relevant test case by double-clicking on one of the test case suggestions.

4.2.1 Non-functional Requirements

For the initial prototype the only important point is that the normal workflow of the programmer is not to be hindered by the tool. Performance is of lesser relevance.

The calculation of the code coverage for each test case is a very expensive operation. Since the goal of the prototype is to validate the basic idea, in the initial implementation the tool will not immediately update its suggestions for relevant code changes when the source code or the test cases are changed. It is expected that this will not be a problem when using the prototype in the evaluation, as participants will spend most of their time navigating the existing source code without modifying it.

The tool is adaptable to different programming languages. As long as it is possible to calculate the statement coverage (see Section 2.2.3—"Code Coverage") for a given language, it should be possible to use the tool.

4.3 Design

Adaptable to different programming languages

As the tool needs to know the syntax of the used programming language, the functionality to parse a source file is realized as an exchangeable module. The tool requires two things from given source code: The names and line numbers of every test case in a file, and, given a line number, the corresponding method.

For simplicity, I do not use a full parser, but regular expressions to find the definitions of test cases and methods. This has proven to be sufficient for a prototype. It also simplifies adapting the tool to different programming languages, since changing regular expressions is less work then implementing a complete parser.

Prototype works only with Ruby

While the tool can be used for different languages, I only implemented the functionality to process Ruby source code. One reason for this is that TDD is a common practice in the Ruby development community, so it is easy to find Open Source projects with large test suites and good code coverage. Another reason is that with RCov[1] an easy code coverage tool for Ruby exists.

Test suggestions are calculated offline

Calculating code coverage is an expensive operation. I therefore divided the tool into two parts: First, a small

[1]http://eigenclass.org/hiki/rcov

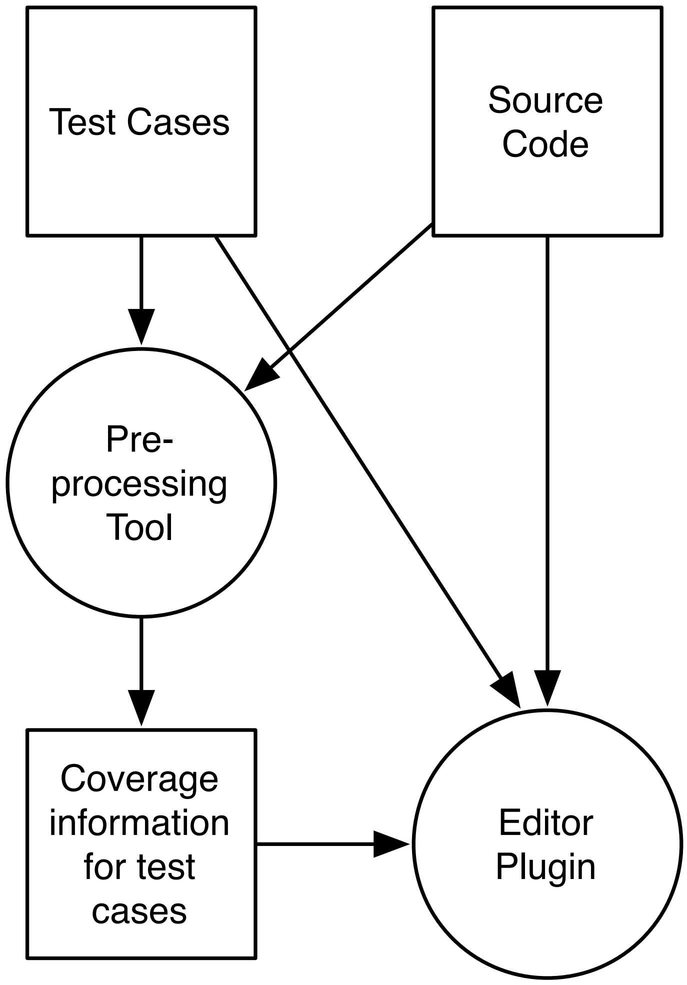


Figure 4.1: High-level design of the prototype

command-line program that generates an XML file containing the list of relevant test cases for each method in the project. Second, the actual editor plugin, which displays the relevant test cases to the user while navigating the project's source files.

I chose to develop a plugin for the TextMate[2] editor, a widely used editor in the Ruby world. TextMate is a text editor running on the Mac OS X operating system. It is de-

[2]http://macromates.com/

veloped in Objective C and although it does not have an official plugin interface, it is easily extendable due to the dynamic nature of Objective C.

The source code for the preprocessing tool and the editor plugin can be downloaded on the chair's wiki[3] .

4.3.1 Preprocessing

Before one can use the editor plugin prototype in a software project, one has to generate the code coverage data for the test cases in the project. The command-line program leverages existing code coverages tools, as it is nontrivial to calculate code coverage.

The command-line program consists of several processing stages:

A list of all methods is created

1. First the program creates a list of all existing methods in all source files. The directory where the source files are located is supplied as a command line argument. To keep the prototype simple, this is currently done with a simple regular expression, which checks each line if it contains the Ruby keyword `def` to declare a method followed by whitespace and a name. If a method definition is found, a new method entry is created and saved with the range of lines until the next method definition. One obvious disadvantage of this solution is that it regards the white space in between method definitions as belonging to the first method. It has proven sufficient for the evaluation, though.

A list of all test cases is created

2. Next, the program creates a list of all test cases. The directory where the test cases are located is supplied as a command line argument. Similar to the method detection above, a regular expression is used to detect the definitions of test cases. For each test case the following information is saved: The containing file, the name of the test case, and the line of definition. This

[3]http://hci.rwth-aachen.de/tiki-download_wiki_attachment.php?attId=765

information is used in the following stage to execute only a single test case and not the whole file or suite.

3. Afterwards, the program executes each of the above test cases. Most TDD frameworks provide a so called "test runner" which is used to execute the whole test suite or individual tests. The test runner is invoked inside the code coverage tool. Each test case is executed individually to be able to relate method invocations to a single test case. The code coverage tool is configured such that it calculates the number of times each source line been executed. This stage is the main reason that the preprocessing can take hours, depending on the speed of the code coverage tool, the number of test cases and the speed of test case execution.

Coverage for each test case is calculated

4. In the next preprocessing stage the program analyzes the output of the code coverage tool. In this output, the coverage tool returns a listing of each source file and indicates for each line the number of times it was executed. Since we created the list of all methods in the first step, saving the execution amount for each method is easy now.

Coverage data is analyzed and saved for each method

5. Finally, the data is saved into an XML file which later can be accessed by the editor plugin.

TEST RUNNER:
A test runner is a tool used to execute a collection of test cases or test suites. In general it is provided by the used testing framework. Most test runners offer the possibility to execute only a single test case identified by its name or by the filename and line number of its definition.

Definition:
Test runner

4.3.2 Data Format

The data format used for the file created by the preprocessing tool is designed according to the requirements of the editor plugin. Because the plugin is written in Objective C using the standard Mac OS X frameworks, the data file

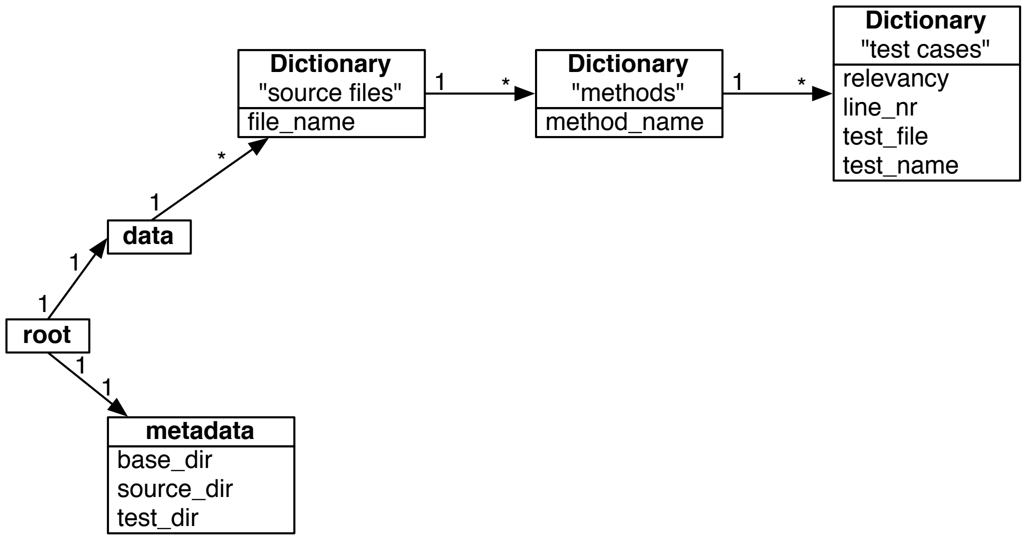


Figure 4.2: XML structure of the data format

should be easily readable in this environment. Additionally, it should be easy to get the list of relevant test cases given a source file and method name.

The data file format is XML and its structure is shown in Figure 4.2. At the top level it contains two entries, one containing the actual coverage data, the other entry containing metadata, such as relative paths to the source and test directories.

The actual data is listed by source file name. For each source file, every contained method is listed. For every method, a list of relevant test cases and their relevancy for the method is saved.

4.3.3 Editor Integration

The TextMate editor is very configurable through the use of bundles. A bundle can, for instance, define new language syntax highlighting, code completion triggers or keyboard shortcuts. But reacting automatically to the user moving through source code is not supported.

TextMate also has the possibility to use plugins which offer enhanced functionality not possible to implement with a bundle. However the only official API TextMate offers to plugins is a single method to load the plugin's main class. No public API exist for plugins to integrate with TextMate.

Editor plugin uses private APIs

Due to the dynamic nature of the Objective C runtime it is possible, though, to hook into existing, internal methods of TextMate. Using a tool called classdump[4] one can generate the class declarations for any given Objective C program. One can then search these class declarations for possible entry points to integrate the plugin using techniques like "method swizzling" or "class posing".

Definition: *Method Swizzling*

METHOD SWIZZLING:
Method Swizzling is a technique to override a method during runtime without subclassing. The Objective C runtime allows to change the implementation of a method using the `method_exchangeImplementations` function. This exchanges the code being executed when being send one of the two messages. This can be useful if one wants to add functionality to a method in a class in such a way, that the new functionality is used whenever said class is used. Subclassing would not work in this case, as other program parts would need to instantiate from the new subclass.

Definition: *Class Posing*

CLASS POSING:
Using Class Posing, a subclass can take the place of one of its superclasses. Every class in Objective C has a method called `poseAsClass:` which takes as an argument the class to pose as. After calling this method, every message sent to the superclass will be received by the subclass. Consequently instantiating new objects from the superclass will create objects from the posing subclass instead.

The plugin needs to be informed about file changes and cursor movement to update the list of relevant test cases. It then displays this list in a table shown in a new window.

[4]http://www.codethecode.com/projects/class-dump/

When the user double-clicks on one of this test cases, the plugin lets TextMate open the test definition. The resulting interface can be seen in Figure 4.6.

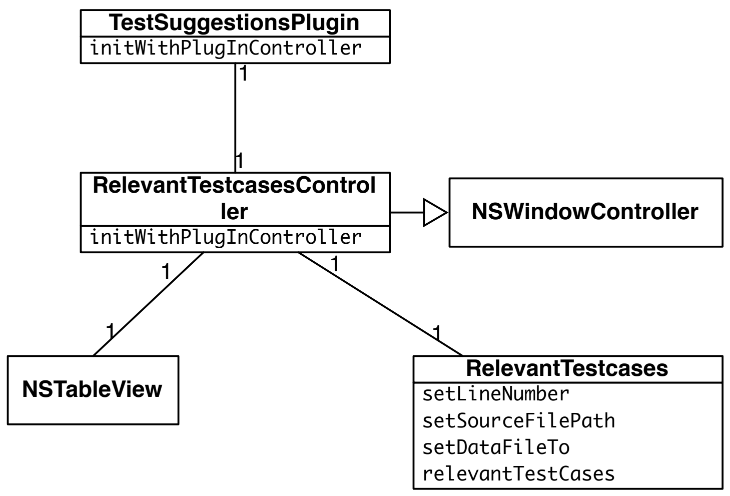


Figure 4.3: Architecture of the TextMate plugin

The main class of the plugin is called `TestSuggestionsPlugin`. After loading the plugin code, TextMate initializes this class by calling its `initWithPlugInController:` method. In this method, the plugin creates the menu entries to display the suggestion window and to load one of the data files generated by the tool described in Section 4.3.1—"Preprocessing". It also hooks into existing classes of TextMate to receive the current file and row index when the user navigates through the source.

TextMate displays the current row index in a status bar at the bottom of each window. Using the Method Swizzling technique I exchanged this method with a newly created method which gets called every time TextMate updates the current row index in its status bar. The new row index is saved and then the old implementation called so TextMate's behavior does not change.

In order to let the plugin react to file changes I made use of the Class Posing technique. The window class in the Cocoa framework holds a reference to the name of the current file it displays. I created a subclass which overrides the `setRepresentedFilename` method, saves the new filename, and then calls the same method on the super class, to make the process transparent to TextMate. By letting this subclass pose as the window superclass, every new window in TextMate uses this subclass and the subclass receives the messages about changed files.

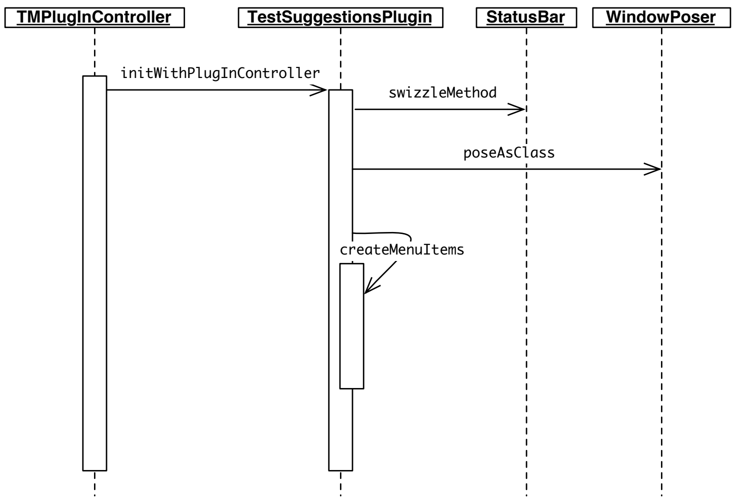


Figure 4.4: Sequence Diagram of plugin initialization

Model layer

The plugin is designed according to the Model-View-Controller paradigm (see Krasner and Pope [1988]). The updated row and file information is routed to an instance of the `RelevantTestcases` class. This instance represents the model layer. Using row index and file, the model determines the method situated at the current location. It then looks up the list of test cases related to that method.

View and Controller layer

This list of relevant test cases is observed by a table view contained in the test case window. Whenever the list changes, the table is automatically updated. The table view also offers the user the possibility to order the test case list by relevancy, test case or file name. The default ordering is by relevancy. In Figure 4.6 the test case window containing a list of test case suggestions can be seen. User interaction is managed by an object of the class `RelevantTestcasesController`. It instructs TextMate to open a test case file when the user double-clicks on one of the displayed test cases.

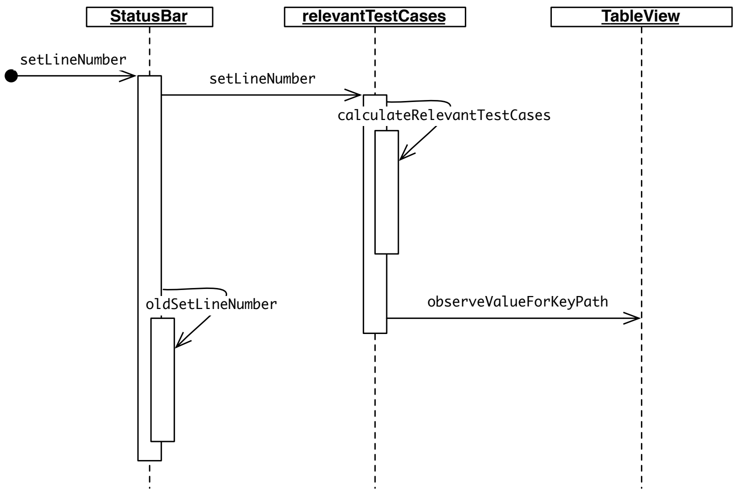
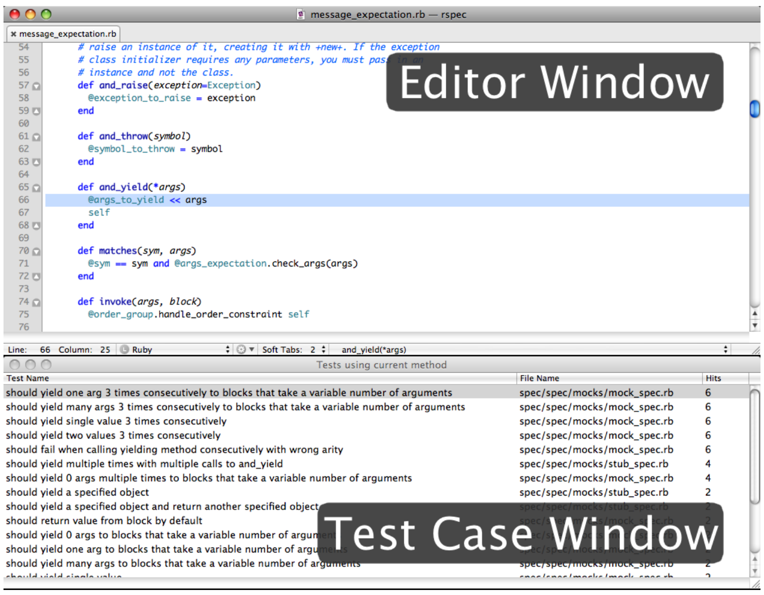


Figure 4.5: Sequence Diagram of messages after user navigation



message_expectation.rb — rspec
Editor Window
Line: 66 Column: 25 Ruby Soft Tabs: 2 and_yield(*args)
Tests using current method
Test Case Window

```
54      # raise an instance of it, creating it with +new+. If the exception
55      # class initializer requires any parameters, you must pass in an
56      # instance and not the class.
57      def and_raise(exception=Exception)
58        @exception_to_raise = exception
59      end
60
61      def and_throw(symbol)
62        @symbol_to_throw = symbol
63      end
64
65      def and_yield(*args)
66        @args_to_yield << args
67        self
68      end
69
70      def matches(sym, args)
71        @sym == sym and @args_expectation.check_args(args)
72      end
73
74      def invoke(args, block)
75        @order_group.handle_order_constraint self
76
```

Test Name	File Name	Hits
should yield one arg 3 times consecutively to blocks that take a variable number of arguments	spec/spec/mocks/mock_spec.rb	6
should yield many args 3 times consecutively to blocks that take a variable number of arguments	spec/spec/mocks/mock_spec.rb	6
should yield single value 3 times consecutively	spec/spec/mocks/mock_spec.rb	6
should yield two values 3 times consecutively	spec/spec/mocks/mock_spec.rb	6
should fail when calling yielding method consecutively with wrong arity	spec/spec/mocks/mock_spec.rb	6
should yield multiple times with multiple calls to and_yield	spec/spec/mocks/stub_spec.rb	4
should yield 0 args multiple times to blocks that take a variable number of arguments	spec/spec/mocks/mock_spec.rb	4
should yield a specified object	spec/spec/mocks/stub_spec.rb	2
should yield a specified object and return another specified object	spec/spec/mocks/stub_spec.rb	2
should return value from block by default	spec/spec/mocks/mock_spec.rb	2
should yield 0 args to blocks that take a variable number of arguments	[illegible]	[illegible]
should yield one arg to blocks that take a variable number of arguments	spec/spec/mocks/mock_spec.rb	2
should yield many args to blocks that take a variable number of arguments	spec/spec/mocks/mock_spec.rb	2

Figure 4.6: Test case suggestions for the selected method starting at line 65

Chapter 5

Evaluation

In order to evaluate the prototype I conducted a user study comparing the effectiveness of debugging with and without the help of the prototype.

5.1 Participants

The prototype was evaluated with eight male participants. Only one participant had prior knowledge in software tests as presented in Section 2.2—"Software Testing". The age range was from 21 to 36 years with an average of 27.5 years and a standard deviation of 5. All but one participant had a professional background in Ruby, working either as a student assistant or as a developer in a company. Three of the participants had already worked with the RSpec framework (see Section 2.3.3—"Example: RSpec") but none had worked with the Liquid library (see Section 2.3.3—"Example: Liquid"). Apart from one, all participants rated themselves as having at least fair knowledge of TDD.

5.2 Set-Up

Each participant completed two tasks

The evaluation was designed with a single independent variable controlling whether participants had access to the prototype or not. Each participants had to work on two software maintenance tasks which involved finding and fixing a bug in an existing software project. Each participant had access to the prototype in one of the two task. The order in which the tasks were presented and access to the prototype were balanced across participants to control possible learning effects.

Program comprehension measured via task completion time

To measure the level of program comprehension the durations until a participant found and fixed a bug were taken. Additionally, the participants were asked to subjectively rate their program comprehension. Several questionnaires and an interview after the evaluation were used to obtain qualitative data about the work of the participants with the prototype.

The evaluation was done on Apple Macs running OS X 10.5. TextMate was used in version 1.5.8, the latest version at the time of writing. The displays used were at least 20″ in size, to allow showing two TextMate windows side by side. The left window was used to display the files containing the test cases while the right showed the actual source files.

Each participant first had to fill out questionnaire A.4 to get basic information about their general knowledge of programming, testing and test-driven development. They were also asked if they already have experience with the used Open Source projects.

Next, I gave the participants the general instructions A.1 applicable to both tasks. They should imagine being part of a team maintaining a certain piece of software. A bug has to be fixed in a component of the software with which they do not have any experience. Since the bug is critical it has to be fixed quickly. They will receive a short description of the bug, expected and received output and a hint where to start. Their work consists in finding the cause of the bug and, if possible, to fix it.

Provided acceptance test indicated task completion

The participants had 30 minutes to complete each task. A special test case was supplied which acted as an acceptance test. I instructed the participants how this test case can be run so they can control if their proposed fix is successful. The existing test cases of each project could be executed, too, so participants could control if their fix affected other parts of the program. Before the first use of the prototype I explained the functionality and let the participants learn how to use it to navigate between source and test files.

Access to debugger or Internet generally disallowed

The participants could at any time access the printed task explanation and were free to take notes while performing the task. In general participants were not allowed to use the Internet. One exception was made for a participant with only basic knowledge of Ruby to look up the definition of a method in the official Ruby documentation. Participants were not allowed to use a debugger but could change code and see the result when running any of the existing test cases.

Immediately after each task, the participants had to fill out questionnaire A.5 about their strategy for resolving the task. This was done directly after each task to avoid confusion with the other task. After both task had been completed the questionnaire A.6 was given comparing the tasks with and without the prototype.

5.2.1 Task RSpec

This task consisted of fixing a bug which occurred in the RSpec project on February 26th, 2008. When setting a negative expectation on the result of a method of an object, RSpec did not handle raised exceptions inside that method correctly. Instead of letting the test fail, it would be marked as passed.

A simple example of how to reproduce the bug is given in Listing 5.1. The class `Person` raises an exception when calling its `has_existing_login?` method. In the test case below a `Person` object is created. Using RSpec's `should_not` negative expectation method and its `has-`matcher it is checked that the new object does not have an existing login. This expectation should fail, as the method neither returns true nor false but instead raises an exception. But the test reports success instead.

```
class Person
  def has_existing_login?
    raise RecordNotFound
  end
end

describe "When creating a new person" do
  it "not have an existing login" do
    person = Person.new
    person.should_not have_existing_login
  end
end
```

Listing 5.1: Minimal example of the RSpec bug

The bug is located inside the `has`-matcher. An instance of this matcher is generated if RSpec encounters a call to an unknown method starting with `have_`. To evaluate the result of the matcher RSpec then calls its `matches?` method, which is shown in Listing 5.2. The matcher then tries to find a corresponding method on the test subject adhering to coding conventions of Ruby. In this case, the unknown method is `have_existing_login` and the test subject is the new `Person` object. It converts the method name to `has_existing_login?` and calls this method on the test subject.

The bug is caused by the exception handling inside the `matches?` method. Any exception thrown inside the test subject is rescued directly in the matcher which then returns `false`. As a negative expectation using `should_not` expects a return value of `false` it incorrectly reports a succeeded test.

The fix used by the RSpec project was to simply remove all exception handling inside the matcher, and let the exception be handled on a higher level.

As the bug has been fixed in the mean time the participants were given a version of the source code just before the fix. The git[1] commit identifier is

[1]http://git-scm.com/

```ruby
def matches?(target)
  @target = target
  begin
    return target.send(predicate, *@args)
  rescue => @error
    # This clause should be empty, but rcov
    # will not report it as covered unless
    # something (anything) is executed
    # within the clause
    rcov_error_report = "dummytext"
  end
  return false
end
```

Listing 5.2: RSpec code containing the bug

12df2fb5feb7f20ba1b3b7d6b2ece4ba5f560b8a. Some existing test cases of this version had to be removed as they were not running successfully. These test cases did not have any relationship to the task. I created the coverage data with the tool described in Section 4.3.1—"Preprocessing" for the remaining tests.

GIT:
Git is a distributed version control system. Contrary to other common version control systems, revisions and commits are identified by a hash of their respective content and not by an incrementing number. This naming scheme allows for globally unique identifiers.

Definition:
git

After explaining the general functionality of RSpec and giving the participants some examples of how it is used, I handed them the task description. This description consisted of an explanation how to cause the bug, what the expected output is and the current, false output. I opened the given initial starting file and started the timer.

5.2.2 Task Liquid

The bug to be fixed in this task was reported to the Liquid project on February 12th, 2009. A fix was attached to the bug report but it is still not integrated into the main repository. If in an `if` statement a variable containing an empty string is compared with an empty string constant the expected result is `true` but `false` is returned instead.

The simplest way to reproduce the bug is given in Listing 5.3. The variable `empty_string` is assigned an empty string. The comparison in line 1 with the empty string constant always yields `false`, which prevents the output of line 2.

```
1 {% if empty_string == "" %}
2 <p>This will not be printed</p>
3 {% endif %}
```

Listing 5.3: Minimal example of the Liquid bug

This bug is caused by one of the regular expression used by Liquid to tokenize the template. The regular expression can be seen in Listing 5.4. It is used to detect single or double quoted strings inside expressions. It matches every substring starting and ending with either a ″ or ′. The bug is introduced in the `[^"]+` and `[^']+` parts. The + operator causes the regular expression to only match if at least one character is contained inside the quotes.

```
QuotedString = /"[^"]+"|'[^']+'/
```

Listing 5.4: Liquid code containing the bug

The proposed fix is quite simple. By using the ⋆ operator instead of the + operator the regular expression will also match empty quoted strings.

As the bug has not been fixed in the Liquid project I provided the participants with the most recent version of the code identified with the git commit hash ed1b542abf73d1d7c1885ee158410c6575a95668. As none of the existing test cases cover the usage of empty strings they all pass.

I explained the general functionality of Liquid with a simple template and gave them the task description. This description contained the template from Listing 5.3 and the expected and resulting output. I opened the initial starting file and started the timer.

5.3 Results

5.3.1 Quantitative

No significant difference in program comprehension

In the RSpec task 63% of participants found the bug and also found a way to fix it. In the Liquid task 75% found the cause of the bug but only 50% found a way to fix it. With respect to the independent variable, 75% found the cause of the bug when using the prototype compared to 63% when not using the prototype. 50% of the participants fixed the bug successfully when using the prototype compared to 63% when not using the prototype. The difference in task completion between both groups is not significant.

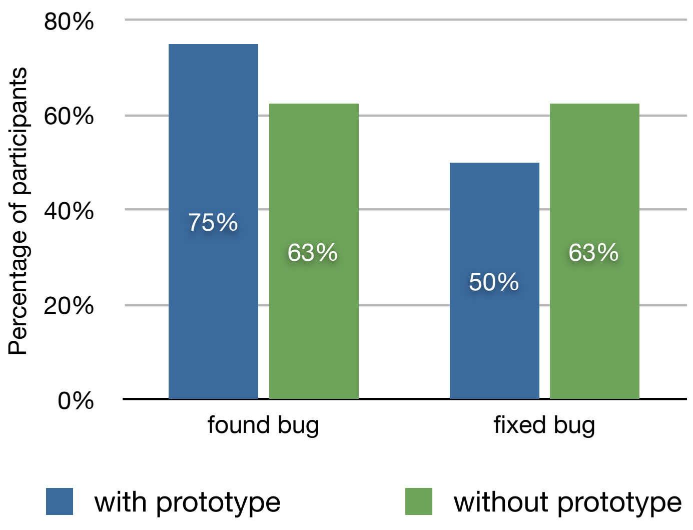


Figure 5.1: Percentage of participants solving each subtask

The average completion times for the bug-finding and bug-fixing subtasks did not vary significantly. A paired, one-sided Student's t-test for each subtask did not reveal any statistically significant difference. The average time to find the cause of the bug was 21.6 min for the prototype group and 20.9 min for the control group with a standard deviation of 4.3 and 5.5 min respectively.

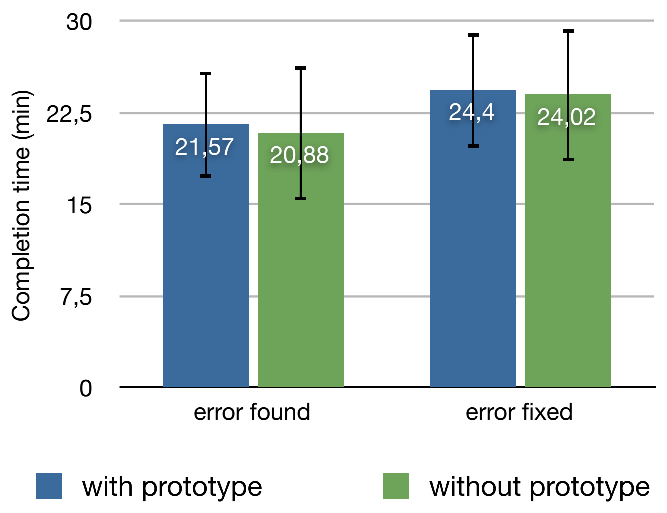


Figure 5.2: Average task completion time

When using the prototype participants spent more time with test cases

The analysis of the questionnaire yielded more interesting results. In question B1 participants were asked how much of the total time used they spent inspecting the test cases. When using the prototype participants spent on average 35% with the test cases compared to only 17.5% without the prototype. Although the standard deviation was relatively high, a paired one-sided Student's t-test confirmed a significant difference.

Availability of test cases highly valued by participants

Most participants, regardless if just having worked with the prototype or not, were in high agreement with question B5, were it was asked if they would always liked to have test cases to consult. No significant difference was found for the other questions.

Questions B2 to B4 concerned the perceived helpfulness of the test cases for understanding the code, finding and fix-

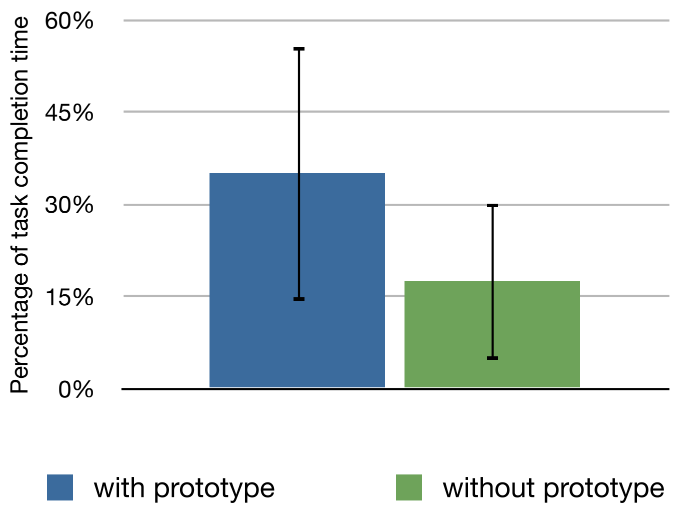


Figure 5.3: Average time spent inspecting test cases

ing the bug. No trend was found in the answers for these questions between groups.

5.3.2 Qualitative

After completion of the user test the participants were asked for general comments about the usage and the effect the prototype had on their maintenance strategy.

- Two participants complained that the measure of relevancy for the test cases was not clear to them. In the presented prototype the raw relevancy measure was shown. In the prototype this was just the number of calls to the currently inspected method during each test case. These participants were not sure how this value was relevant to their work. As the main focus of the prototype was on the interaction and not on the relevancy algorithm this does not come as a surprise. More work is needed to come up with a better relevancy measure and how to convey this to the user.

Ordering of test cases unclear

- One participant mentioned that it was helpful to see a list of test cases. As he was not very familiar with

Being remembered of test cases considered helpful

TDD this reminded him of the additional artifacts available for the maintenance task. This is in accordance with the results for question B1 of the questionnaire, where participants were asked how much time they spent inspecting test cases.

Quality of test cases is important

- Two participants commented that the format how the test cases were programmed did not help them finding out how to use a method or class inside the code. This might be because many test cases use special helper methods to instantiate objects or assert certain results. These helper methods are of no use in the actual program, as they are highly specialized for the test environment. Such they present an additional abstraction to overcome in order to understand the tested code. This might represent a fundamental problem in using test cases as a form of example code.

- Three participants explicitly mentioned that they liked the easy navigation between the currently inspected method and its test cases.

TDD experience affects usage of prototype

- One participant did never look at any test case and did not find any of the bugs. He stated that, as he has no experience with TDD, he just followed his usual bug finding strategy. As it was not possible to just run the presented projects (both are libraries without any UI) he resorted to manually following the flow through the methods. This shows that the prototype requires a basic understanding of what a test cases provides and how to use it.

Provide reverse navigation from test cases to used code

- One participant would have liked a list of relevant methods for each test case, i.e. the reverse of what is currently presented in the prototype. This would result in a more generally usable navigation technique between test cases and corresponding code. It is not as trivially implementable, though, as in general each test cases indirectly calls many methods, so the possible set of relevant methods would be quite big, and a better relevancy algorithm would be needed.

Chapter 6

Discussion

The goal of the prototype was to support program comprehension during software maintenance by improving access to the test cases. The idea was to give better access to the test case artifacts created when developing with the TDD process. To evaluate the prototype, a user study was performed using real maintenance tasks from two Open Source projects. In the following, I will discuss the results of the user study and put them in relation to the requirements (see Section 4.2—"Requirements") of the original prototype.

6.1 User Study Discussion

Performing realistic evaluations is difficult

The results did not yield a clear picture about the effect of the prototype on maintenance performance. One problem was the small sample size, as it was difficult to find programmers with experience in Ruby. More generally, the evaluation of tools supporting software engineering tasks is difficult, since external variables like programmer experience and task difficulty are hard to control

This comes not as a big surprise. The studies presented in Section 3.1—"Evaluating the Effectiveness of Test-Driven Development" which used real software projects as the basis for the tasks showed similar differing results: Some studies demonstrating more productivity and others only better external quality.

The prototype provided a better visibility of the test cases when inspecting source code, as shown by the user comments and by the results for question B1. It seems that especially for people without much experience in TDD it helps to prominently display the test cases. An experienced TDD developer might access the test cases automatically as a part of her maintenance workflow, whereas inexperienced developers can benefit from bringing relevant test cases to their attention automatically.

The decision to focus the initial prototype on Ruby was made as most Ruby projects are developed with TDD practices. As the usefulness of the technique presented in this work depends directly on the quality of the test cases of the underlying software project, focussing on Ruby made the selection of tasks for the evaluation easier.

Another problem during the evaluation was finding the exact time when the source of a bug was located. Participants were instructed to notify the experimenter when they thought to have found the bug. But this event cannot be pinpointed exactly in time. In some cases it turned out that the participants did not find the actual source of the bug but rather another symptom, which was revealed in discussion with the experimenter. In other cases participants forgot to notify the experimenter and directly proceeded to fix the bug.

6.2 Requirements Discussion

Displaying of test cases well received

The first requirement was that relevant test cases should be displayed given a method currently selected by the programmer. This was realized and recognized by user feedback as useful.

Relevancy measure unclear

A problem mentioned by several participants was how to interpret the relevancy measure for each test case. The prototype gave no indication how the relevancy measure for a certain test case was computed. The acceptance of a measure is a critical point for acceptance of the entire tool by developers, though. More work needs to be done in order to communicate this measure efficiently.

Additionally, the employed algorithm to calculate the relevancy measure of a test case was very basic. The algorithm only counts the number of times a certain method is invoked by a given test case. Other measures might give better results. For example, the number of indirect method calls between the call inside the test case and the selected method (i.e. the stack depth) can be used to fine tune the relevancy measure. A test case which calls a given method only through many indirections is probably not as relevant as a test case which directly calls a method.

Prototype integrated well with TextMate

The second requirement was that the tool should integrate well with an existing editor and should automatically update the test case suggestions while the programmer navigates the source code. All participants had at least basic experience using TextMate, and had no problems integrating the tool into their normal workflow.

The last functional requirement was that the tool should offer easy navigation between a given method and its relevant test cases. This requirement was fulfilled by the prototype. Some participants explicitly mentioned that they liked the easy navigation. It was mentioned, though, that reverse navigation (from a test case to methods it uses) might be practical. This can be easily achieved with the data currently used and should probably be implemented in future versions of the prototype.

Test case suggestions are currently calculated by a separate tool and thus not automatically updated when code is changed. As expected, this was not a problem during the user study. In both tasks participants only had to change a small part of a method in order to fix the bug. This change did not have a great effect on the code coverage of the existing test cases and hence the relevancy measure. But further work is needed to reduce the time complexity of the relevancy algorithm in order to use the editor plugin during real software maintenance.

6.3 Summary

While no hard data about the effect on maintenance performance was found, the overall positive user comments encourage further work on the better integration of TDD artifacts into the maintenance process. As shown in Robillard et al. [2004], methodical investigation of an unknown code base is key to better maintenance performance. Easier navigation between related parts of the code base can help developers investigate source code more efficiently.

Participants especially liked the navigational capabilities of the prototype and gave valuable feedback on how to improve the navigation further. More work is needed to come up with a better and clearer relevancy measure.

Chapter 7

Conclusion

7.1 Summary and Contributions

Improving program comprehension is an active field of research. Beginning from models how programmers comprehend code, over ways to measure program comprehension to tools for improving comprehension, future research is needed.

In this thesis, I presented a novel approach to improve program comprehension by using a newly developed tool that allows for directly accessing test cases from source code. Existing test cases are used as example code on how to use certain parts of the code.

This approach is targeted to projects employing TDD. In doing this, one can expect to have an existing test suite covering most if not all parts of a program. This has the advantage of being able to expect certain artifacts like test cases to always be available, making them good candidates for improving comprehension.

After having explained the basic idea behind this approach, I defined several requirements for an initial prototype. The prototype was then developed and evaluated in a user study. The user study did not show a significant positive impact on program comprehension. Qualitative user feedback was positive, though.

7.2 Future Work

7.2.1 Relevancy Measure

The used measure to determine the relevancy of test cases for a given method was not clear to every participant. Better ways to communicate the relevancy measure should be investigated in the future. This could involve simply finding better means to communicate to the user how the relevancy measure is computed. Another possibility is to change the visualization of test case suggestions: Instead of a simple ordered list, one could present the test cases at the top of their call graph and show the investigated method at the bottom. This approach would have the advantage that other relevant parts connected to this method can be displayed inside the call graph, as well.

The actual relevancy measure can be improved, too. Currently, only the number of times a method is called by a certain test case is regarded. But a test case for a high-level function of a program might indirectly call many subsystems which this high-level function depends on. Other approaches could take the stack depth between a test case and a method into account as a measure of relevancy.

7.2.2 Relevancy Calculation

In the realized prototype, as of now, the calculation of relevancy is not performed in real-time. For example, calculating the relevancy for all test cases for the RSpec project used in the user study takes more than 2 hours on a current generation processor. This is why the prototype performs this calculation offline and does not update the suggestions automatically when code is changed.

No automatic update of suggestions will be a problem if the presented tool is to be used for real software maintenance tasks. While the developer changes code, the suggestions will become stale and in the worst case convey wrong information.

A possible solution to allow automatic updates of suggestions is to improve the efficiency of the relevancy calculation. For example, when changing existing test cases, relevancy calculation only has to be repeated for those test cases. The relevancy could also be updated in the background without interrupting the programmer's workflow.

7.2.3 Navigation

Based on the user feedback, the navigational capabilities of the editor plugin were well accepted. Additional navigation options would be useful, though. A simple improvement would be to offer navigation from a test case to its used methods, based on the same relevancy measure. This could help in easier discovery of related program elements.

Other possible future improvements include the integration of the plugin into the editing environment. Instead of showing the test cases in a separate window they could be accessed directly from inside the editor window.

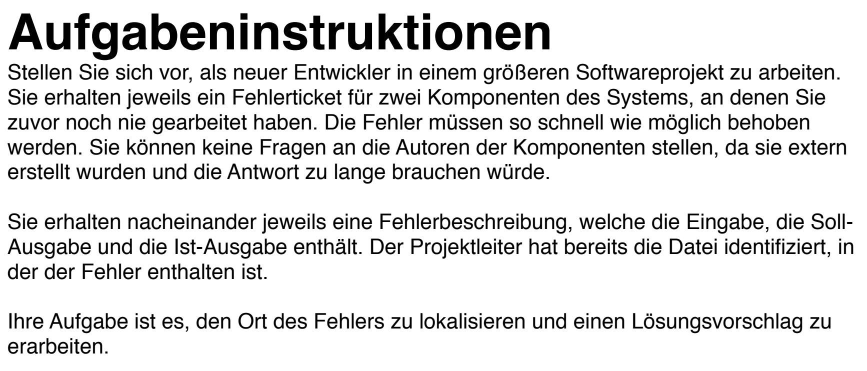
7.2.4 Quality of Test Cases

User feedback revealed that not all test cases are equally helpful for program comprehension. Macros or special helper functions only used inside test cases can negatively affect the usefulness of test cases as example code.

Future work could determine impacts on the readability of test cases. The results of this work could be used as recommendations how to construct better test cases when employing TDD.

Appendix A

Evaluation Materials



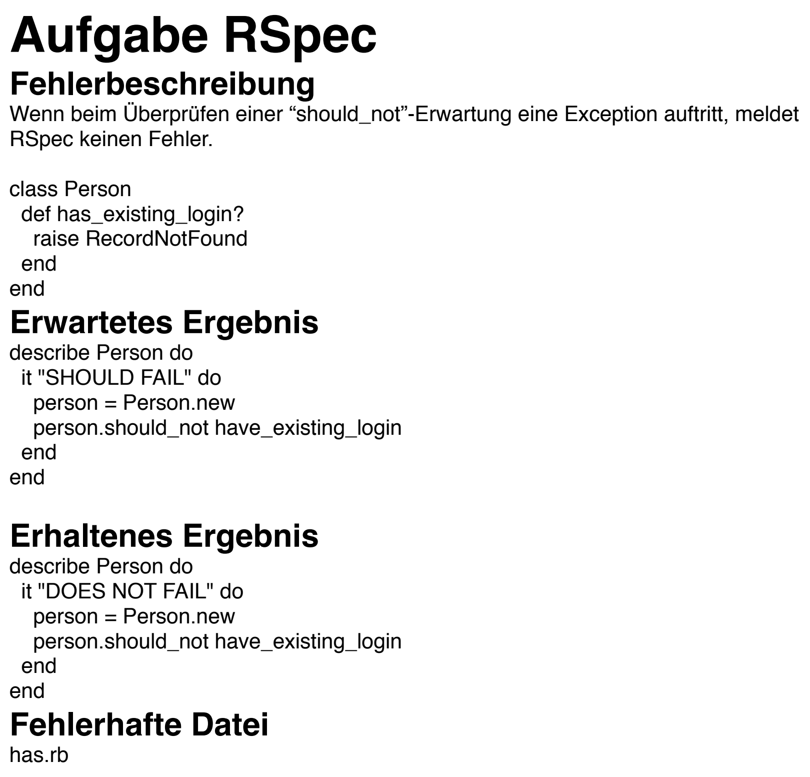
Aufgabeninstruktionen

Stellen Sie sich vor, als neuer Entwickler in einem größeren Softwareprojekt zu arbeiten. Sie erhalten jeweils ein Fehlerticket für zwei Komponenten des Systems, an denen Sie zuvor noch nie gearbeitet haben. Die Fehler müssen so schnell wie möglich behoben werden. Sie können keine Fragen an die Autoren der Komponenten stellen, da sie extern erstellt wurden und die Antwort zu lange brauchen würde.

Sie erhalten nacheinander jeweils eine Fehlerbeschreibung, welche die Eingabe, die Soll-Ausgabe und die Ist-Ausgabe enthält. Der Projektleiter hat bereits die Datei identifiziert, in der der Fehler enthalten ist.

Ihre Aufgabe ist es, den Ort des Fehlers zu lokalisieren und einen Lösungsvorschlag zu erarbeiten.

Figure A.1: General instructions for user test participants



Aufgabe RSpec

Fehlerbeschreibung

Wenn beim Überprüfen einer “should_not”-Erwartung eine Exception auftritt, meldet RSpec keinen Fehler.

```
class Person
  def has_existing_login?
    raise RecordNotFound
  end
end
```

Erwartetes Ergebnis

```
describe Person do
  it "SHOULD FAIL" do
    person = Person.new
    person.should_not have_existing_login
  end
end
```

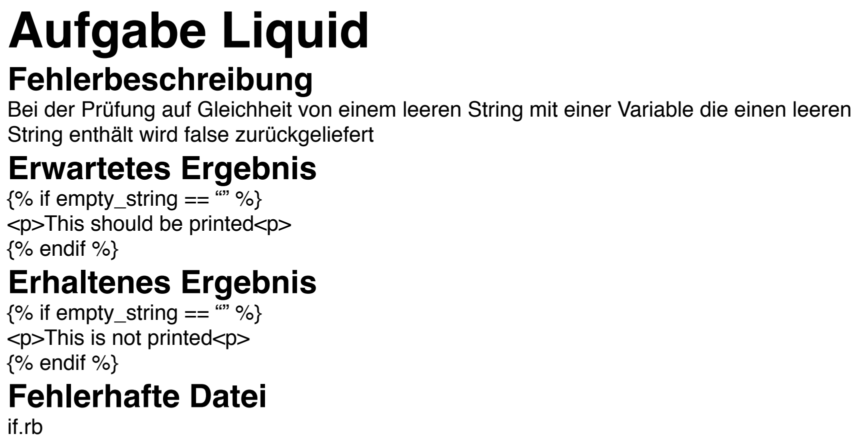
Erhaltenes Ergebnis

```
describe Person do
  it "DOES NOT FAIL" do
    person = Person.new
    person.should_not have_existing_login
  end
end
```

Fehlerhafte Datei

has.rb

Figure A.2: Instructions for the RSpec task



Aufgabe Liquid

Fehlerbeschreibung

Bei der Prüfung auf Gleichheit von einem leeren String mit einer Variable die einen leeren String enthält wird false zurückgeliefert

Erwartetes Ergebnis

```
{% if empty_string == “” %}
<p>This should be printed<p>
{% endif %}
```

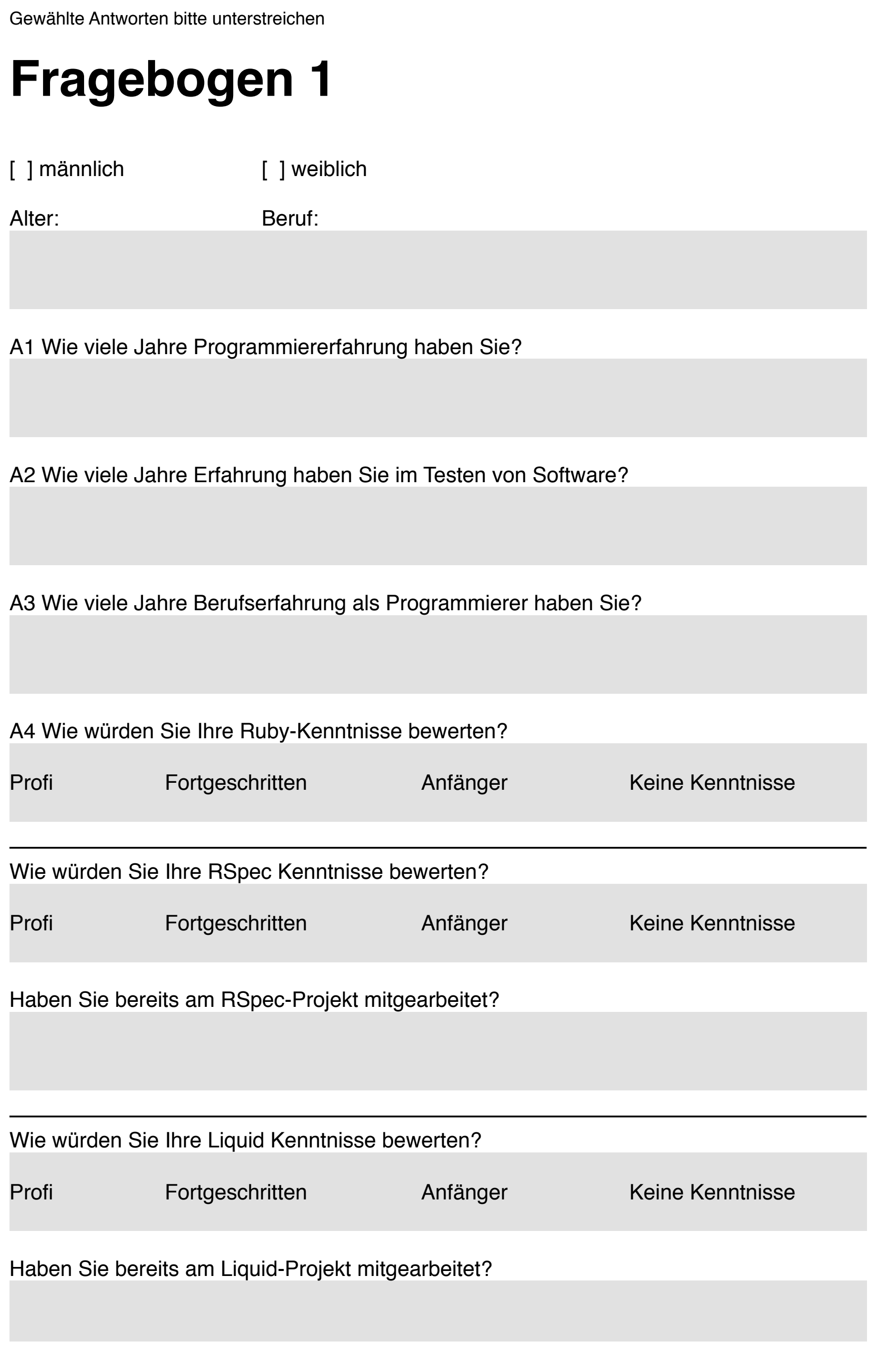
Erhaltenes Ergebnis

```
{% if empty_string == “” %}
<p>This is not printed<p>
{% endif %}
```

Fehlerhafte Datei

if.rb

Figure A.3: Instructions for the Liquid task



Gewählte Antworten bitte unterstreichen

Fragebogen 1

[ ] männlich [ ] weiblich

Alter: Beruf:

A1 Wie viele Jahre Programmiererfahrung haben Sie?

A2 Wie viele Jahre Erfahrung haben Sie im Testen von Software?

A3 Wie viele Jahre Berufserfahrung als Programmierer haben Sie?

A4 Wie würden Sie Ihre Ruby-Kenntnisse bewerten?

Profi	Fortgeschritten	Anfänger	Keine Kenntnisse

Wie würden Sie Ihre RSpec Kenntnisse bewerten?

Profi	Fortgeschritten	Anfänger	Keine Kenntnisse

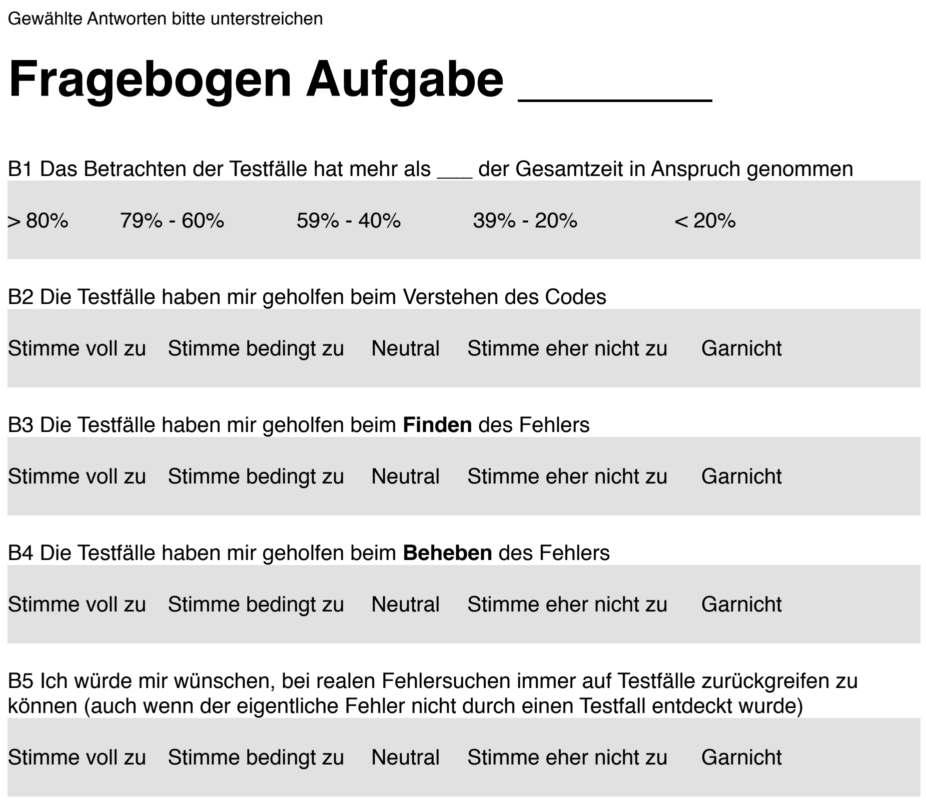
Haben Sie bereits am RSpec-Projekt mitgearbeitet?

Wie würden Sie Ihre Liquid Kenntnisse bewerten?

Profi	Fortgeschritten	Anfänger	Keine Kenntnisse

Haben Sie bereits am Liquid-Projekt mitgearbeitet?

Figure A.4: Questionnaire filled out before the user test



Gewählte Antworten bitte unterstreichen

Fragebogen Aufgabe _______

B1 Das Betrachten der Testfälle hat mehr als ___ der Gesamtzeit in Anspruch genommen

> 80%	79% - 60%	59% - 40%	39% - 20%	< 20%

B2 Die Testfälle haben mir geholfen beim Verstehen des Codes

Stimme voll zu	Stimme bedingt zu	Neutral	Stimme eher nicht zu	Garnicht

B3 Die Testfälle haben mir geholfen beim **Finden** des Fehlers

Stimme voll zu	Stimme bedingt zu	Neutral	Stimme eher nicht zu	Garnicht

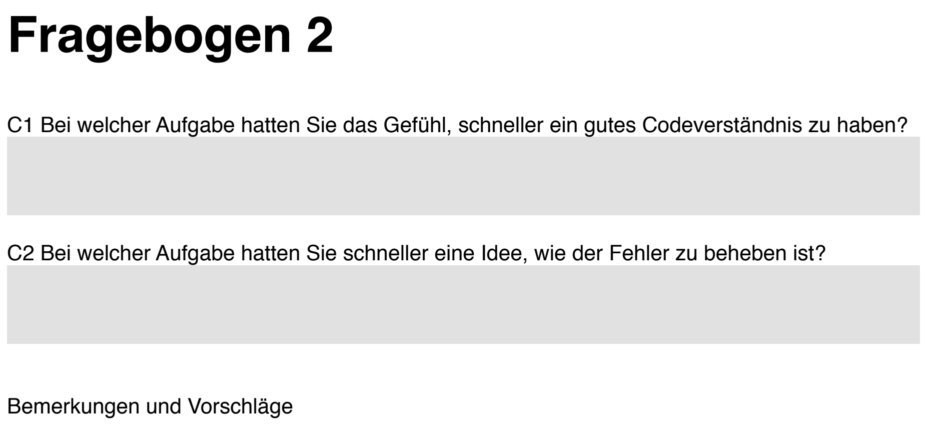
B4 Die Testfälle haben mir geholfen beim **Beheben** des Fehlers

Stimme voll zu	Stimme bedingt zu	Neutral	Stimme eher nicht zu	Garnicht

B5 Ich würde mir wünschen, bei realen Fehlersuchen immer auf Testfälle zurückgreifen zu können (auch wenn der eigentliche Fehler nicht durch einen Testfall entdeckt wurde)

Stimme voll zu	Stimme bedingt zu	Neutral	Stimme eher nicht zu	Garnicht

Figure A.5: Questionnaire filled out after each task



Fragebogen 2

C1 Bei welcher Aufgabe hatten Sie das Gefühl, schneller ein gutes Codeverständnis zu haben?

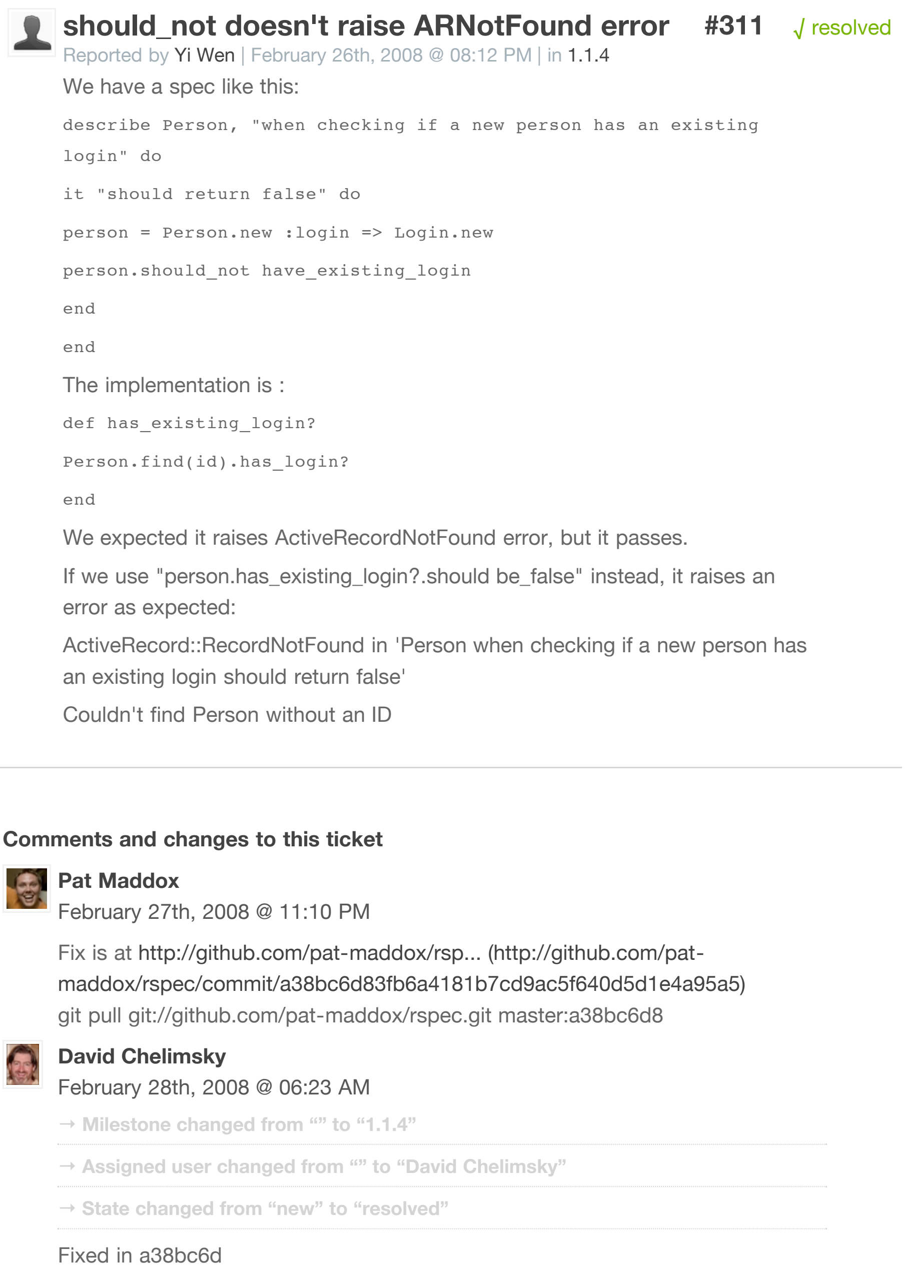
C2 Bei welcher Aufgabe hatten Sie schneller eine Idee, wie der Fehler zu beheben ist?

Bemerkungen und Vorschläge

Figure A.6: Questionnaire filled out after the user test

Appendix B

Bug Tickets



should_not doesn't raise ARNotFound error #311 √ resolved

Reported by Yi Wen | February 26th, 2008 @ 08:12 PM | in 1.1.4

We have a spec like this:

```
describe Person, "when checking if a new person has an existing
login" do

it "should return false" do

person = Person.new :login => Login.new

person.should_not have_existing_login

end

end
```

The implementation is :

```
def has_existing_login?

Person.find(id).has_login?

end
```

We expected it raises ActiveRecordNotFound error, but it passes.

If we use "person.has_existing_login?.should be_false" instead, it raises an error as expected:

ActiveRecord::RecordNotFound in 'Person when checking if a new person has an existing login should return false'

Couldn't find Person without an ID

Comments and changes to this ticket

Pat Maddox
February 27th, 2008 @ 11:10 PM

Fix is at http://github.com/pat-maddox/rsp... (http://github.com/pat-maddox/rspec/commit/a38bc6d83fb6a4181b7cd9ac5f640d5d1e4a95a5)
git pull git://github.com/pat-maddox/rspec.git master:a38bc6d8

David Chelimsky
February 28th, 2008 @ 06:23 AM

→ Milestone changed from “” to “1.1.4”

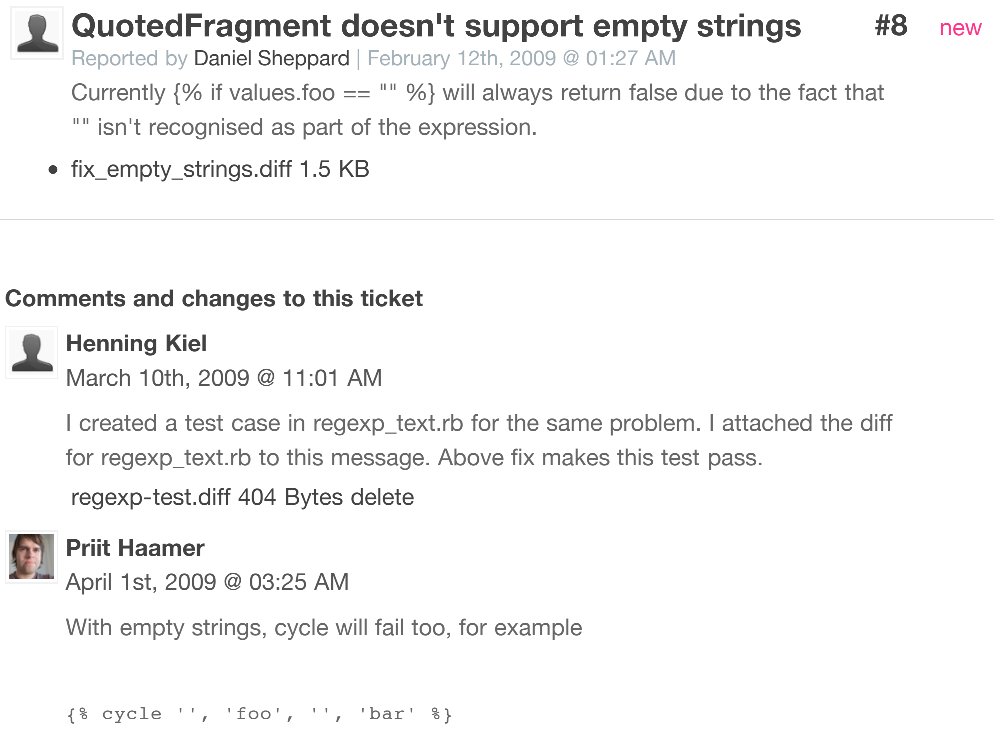
→ Assigned user changed from “” to “David Chelimsky”

→ State changed from “new” to “resolved”

Fixed in a38bc6d

Figure B.1: Original bug report for the RSpec task[a]

[a]https://rspec.lighthouseapp.com/projects/5645/tickets/311-should_not-doesn-t-raise-arnotfound-error



QuotedFragment doesn't support empty strings #8 new

Reported by Daniel Sheppard | February 12th, 2009 @ 01:27 AM

Currently {% if values.foo == "" %} will always return false due to the fact that "" isn't recognised as part of the expression.

- fix_empty_strings.diff 1.5 KB

Comments and changes to this ticket

Henning Kiel
March 10th, 2009 @ 11:01 AM

I created a test case in regexp_text.rb for the same problem. I attached the diff for regexp_text.rb to this message. Above fix makes this test pass.

regexp-test.diff 404 Bytes delete

Priit Haamer
April 1st, 2009 @ 03:25 AM

With empty strings, cycle will fail too, for example

```
{% cycle '', 'foo', '', 'bar' %}
```

Figure B.2: Original bug report for the Liquid task[a]

[a]https://jadedpixel.lighthouseapp.com/projects/11053/tickets/8-quotedfragment-doesnt-support-empty-strings

Bibliography

Lowell Jay Arthur. *Software evolution: the software maintenance challenge*. Wiley-Interscience, New York, NY, USA, 1988. ISBN 0-471-62871-9.

Dave Astels. A new look at test driven development. July 2005.

Kent Beck. *Test-Driven Development By Example*. Addison Wesley, 2002.

Thirumalesh Bhat and Nachiappan Nagappan. Evaluating the efficacy of test-driven development: industrial case studies. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 356–363, New York, NY, USA, 2006. ACM. ISBN 1-59593-218-6. doi: http://doi.acm.org/10.1145/1159733.1159787.

S. D. Conte, H. E. Dunsmore, and Y. E. Shen. *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1986. ISBN 0-8053-2162-4.

Alastair Dunsmore and Marc Roper. A comparative evaluation of program comprehension measures. *The Journal of Systems and Software*, 52(3):121–129, June 2000.

Boby George and Laurie Williams. An initial investigation of test driven development in industry. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 1135–1139, New York, NY, USA, 2003. ACM. ISBN 1-58113-624-2. doi: http://doi.acm.org/10.1145/952532.952753.

Daniel M. German, Peter C. Rigby, and Margaret-Anne Storey. Using evolutionary annotations from change logs to enhance program comprehension. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 159–162, New York, NY, USA, 2006. ACM. ISBN 1-59593-397-2. doi: http://doi.acm.org/10.1145/1137983.1138020.

Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 301–310, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1368088.1368130.

Andrew J. Ko and Brad A. Myers. Finding causes of program output with the java whyline. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 1569–1578, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-246-7. doi: http://doi.acm.org/10.1145/1518701.1518942.

Jürgen Koenemann and Scott P. Robertson. Expert problem solving strategies for program comprehension. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 125–130, New York, NY, USA, 1991. ACM. ISBN 0-89791-383-3. doi: http://doi.acm.org/10.1145/108844.108863.

Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988. ISSN 0896-8438.

P.J. Layzell and L. Macaulay. An investigation into software maintenance-perception and practices. pages 130–140, Nov 1990. doi: 10.1109/ICSM.1990.131342.

Jochen Ludewig and Horst Lichter. *Software Engineering*. dpunkt.verlag, 2007.

M.M. Muller and O. Hagner. Experiment about test-first programming. *Software, IEE Proceedings -*, 149(5):131–136, Oct 2002. ISSN 1462-5970. doi: 10.1049/ip-sen:20020540.

Glenford J. Myers. *The art of software testing*. Business data processing, a Wiley series. Wiley (New York), 1979.

Martin P. Robillard. Automatic generation of suggestions for program investigation. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 11–20, New York, NY, USA, 2005. ACM. ISBN 1-59593-014-0. doi: http://doi.acm.org/10.1145/1081706.1081711.

Martin P. Robillard, W. Coelho, and G.C. Murphy. How effective developers investigate source code: an exploratory study. *Software Engineering, IEEE Transactions on*, 30(12):889–903, Dec. 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.101.

Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 23–34, New York, NY, USA, 2006. ACM. ISBN 1-59593-468-5. doi: http://doi.acm.org/10.1145/1181775.1181779.

T. Tenny. Program readability: procedures versus comments. *Software Engineering, IEEE Transactions on*, 14(9): 1271–1279, Sep 1988. ISSN 0098-5589. doi: 10.1109/32.6171.

A. Von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, Aug 1995. ISSN 0018-9162. doi: 10.1109/2.402076.

N. Wilde and C. Casey. Early field experience with the software reconnaissance technique for program comprehension. *Software Maintenance, IEEE International Conference on*, 0:312, 1996. ISSN 1063-6773. doi: http://doi.ieeecomputersociety.org/10.1109/ICSM.1996.565034.

Norman Wilde and Michael C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, 1995. ISSN 1040-550X. doi: http://dx.doi.org/10.1002/smr.4360070105.

L. Williams, E.M. Maximilien, and M. Vouk. Test-driven development as a defect-reduction practice. pages 34–45, Nov. 2003. doi: 10.1109/ISSRE.2003.1251029.

Index

Typeset June 2, 2009