RWTH AACHEN UNIVERSITY

HelpMeOut

Crowdsourcing suggestions to programming problems for dynamic, interpreted languages

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

by
Manuel Kallenbach

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Bjoern Hartmann

Registration date: Jun 21st, 2010
Submission date: Jan 18th, 2011

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, January 2011
Manuel Kallenbach

Contents

List of Figures

List of Tables

Abstract

When working on a software project, developers usually encounter a lot of errors they have to fix. To find more information about how to solve them, they usually start to search the web, which is challenging for two main reasons: First, finding a good search query for several reasons is not easy. Second, someone has to have – usually manually – provided the necessary information before.
We present a tool that tries to help with both of these problems. It consists of two components: a central server running a crowdsourced database of fixes and a client program. This client program augments a testing framework for the Ruby programming language and monitors the test executions. When a failing test is encountered, a query for related fixes is automatically generated and sent to the server. Related fixes are then displayed next to the test results for the developer's examination. When a test passes that failed before, a diff of the affected files is sent to the server and becomes part of our crowdsourced database of fixes.
A preliminary evaluation between 8 developers showed that during 8 hours of programming, our tool was able to provide useful suggestions for 57% of the failing tests. During this time 161 new fixes were generated.

Überblick

Während der Entwicklung von Softwareprojekten werden Programmierer häufig mit Fehlern konfrontiert, für die sie eine Lösung finden müssen. Um weitere Informationen zu ihrem aktuellen Problem zu erhalten, verwenden sie häufig Websuche. Dies bringt zwei Probleme: Eine gute Such-Anfrage zu formulieren ist aus verschiedenen Gründen nicht einfach und damit Informationen gefunden werden können, müssen sie vorher von einer anderen Person – meist händisch – zur Verfügung gestellt worden sein.
Wir präsentieren eine mögliche Lösung für diese Probleme. Unser Ansatz besteht aus einer zentralen Datenbank von Lösungen und einem Client-Pogramm. Das Client-Programm überwacht die Testläufe eines Testing-Frameworks für die Ruby Programmiersprache. Wenn Tests fehlschlagen, wird automatisch eine Anfrage nach verwandten Lösungen generiert und an den Server gesendet. Diese Lösungen werden dann in der Ergebnisansicht des Testing Tools präsentiert und dienen den Entwicklern als Anhaltspunkte für ihre eigene Lösung. Wenn unser Tool einen erfolgreichen Test bemerkt, der vorher fehlschlug, werden die Unterschiede der betroffenen Dateien zum Server gesendet und zu einem Teil der Lösungs-Datenbank.
Eine vorläufige Studie unter 8 Entwicklern hat gezeigt, dass ein Prototyp innerhalb von 8 Stunden Programmierung nützliche Vorschläge für 57% der fehlschlagenden Tests machen konnte. Während dieser Zeit wurden weiterhin 161 neue Lösungen generiert.

Acknowledgements

Without the people and companies acting as user testers and allowing us to conduct interviews and contextual inquiries this thesis would not have been possible. I am very grateful for the support we received in this way.

I would also like to thank Prof. Dr. Jan Borchers, Prof. Dr. Björn Hartmann and Leonhard Lichtschlag for providing me with lots of advice and corrections of this thesis.

Valuable contributions came from Brandon Liu and Dhawal Mujumdar, who worked together with me on this project. Thank 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.

```
myClass
```

The whole thesis is written in American English.

Chapter 1

Introduction

Correcting errors is a large part of software development. It starts with the beginning of the implementation phase of a software project, when developers are faced with bugs during programming. After releasing a software, more bugs are usually encountered by end users and are to be corrected for the next version of the software.

Developers are required to figure out what caused the error and find a solution. When errors result in an exception, its message is often cryptic and hard to understand. Relating it to a possible solution is often impossible for novice programmers. When the error manifests not in an exception, but simply in unexpected program behaviour, the task of fixing the bug becomes even harder. Developers now require at least some understanding of the program's internal structure to find the source of the bug.

Hartmann et al. [2010] introduced HelpMeOut to suggest fixes for programming errors in the Processing and Arduino environment. These fixes are collected from users of the HelpMeOut tool during development. Whenever they change their source code in a way that eliminates an exception, HelpMeOut sends these changes to its central server. When developers later encounter a related problem, these fixes are presented as examples to the developers. Their studies showed high potential for such recommender systems.

In dynamic, interpreted languages there is no compiler to do static analysis and catch errors before the program is executed. Our initial interviews showed, that most errors in dynamic languages do not result in a runtime exception, but simply in wrong program behaviour. This makes directly applying the original HelpMeOut technology, that relied on compile-time errors and runtime exceptions, to dynamic languages impossible.

Because there is no compiler to check for errors, test-driven development is widely used together with dynamic languages. The principle of not writing any implementation code unless there is a failing test results in many failing tests during development and gives us a good way to suggest and collect fixes.

This thesis contributes an application of the HelpMeOut concept for dynamic, interpreted languages. It demonstrates how to leverage test-driven development practices to enable crowdsourced bug fix suggestions.

A preliminary evaluation of a prototype implementation showed promising results. Eight developers, each working for one hour on a simple test-driven development task, generated 161 new fixes. For 120 (57%) of the 211 times tests failed, the prototype suggested useful fixes.

1.1 Chapter Overview

The remainder of this work is structured as follows:

Chapter 2 explains terms and concepts important for the understanding of the rest of this work.

Chapter 3 summarizes previous approaches aiding the process of software debugging and presents Hartmann et al. [2010]'s HelpMeOut, which our tool is heavily inspired by.

Chapter 4 then describes our initial design studies and their results. It also reasons about the decisions we took in implementing our prototype.

Chapter 5 presents our prototype, its architecture and technical design.

Chapter 6 describes the studies we evaluated our prototype in and explains their results.

Chapter 7 summarizes this thesis and gives some ideas for future work.

Chapter 2

Theory

This chapter explains terms and concepts that play a fundamental role for the rest of this work.
First we talk about software bugs, their commonness and impact. Next, we show what differentiates dynamic from static languages and describe the Ruby language in particular. Finally, the concept of software testing is explained.

2.1 Software Bugs

SOFTWARE BUG:
A software bug is an error in a computer program that leads to a wrong result.

Definition: *software bug*

This definition implies that bugs do not need to manifest in program crashes. They can also result in wrong program output. These kinds of bugs are less obvious and because of that much harder to detect. In general, bugs are caused by mistakes in the programs source code and are very widespread.

There is probably no completely error free larger software project. According to its bug tracker[1] , the Linux kernel at

Bugs are very common.

[1] http://bit.ly/9cpIRm

the time of this writing (October 2010) contains 7628 bugs that are not yet corrected. One can assume that there are more yet to be discovered.

The industry average is at 15-20 errors per 1000 lines of code.

Reasons for bugs being so common can be seen in the high complexity and the huge source code volume of large programs. The current Linux kernel version 2.6.36 consists of about 13 million lines of code (H-Online[2]). This makes a defect rate of about 0.6 errors per 1000 lines of code, which is in comparison to industry projects very good. According to McConnell [2004] the industry average is at 15-20 errors per 1000 lines of code.

Novices often have problems locating and correcting errors.

Of course, bugs are not only prominent on large, established software projects, but also in programming novices code. As they are not yet familiar with programming in general or the language they use, they are very likely to make a lot of mistakes. When confronted with the results of their errors, they are also more likely to have problems relating them to a possible solution. They lack the experience to quickly relate error messages to the changes that need to be done to resolve them. [McCauley et al., 2008]

Approaches to increase software quality

Great efforts are taken to reduce the number of bugs in software projects. New programming languages are developed that hide *dangerous* features like pointers and assure that programmers only use them in a safe way. Compilers are constantly improved to detect more errors. Lots of static analysis tools (e.g. Coverity[3] , CodeSonar[4] , KlocWork[5]) are developed to check code for error patterns and methodologies like Pair Programming and test-driven development are presented that promise higher code quality, often by better testing techniques.

High economic impact of software bugs

The results of bugs can be very severe. They can present security risks that allow users to execute malicious commands or crash systems that are of a very important function. Software bugs even let to fatal accidents, like the crash of the Ariane 5 rocket. According to a study by the

[2] http://www.h-online.com/open/features/What-s-new-in-Linux-2-6-36-1103009.html?page=6
[3] http://coverity.com/
[4] http://www.grammatech.com/products/codesonar/overview.html
[5] http://klocwork.com/

US National Institute of Standards and Technology[6] , software bugs cost the US economy about $59 billion per year. [Tassey, 2002]

2.2 Dynamic languages

A programming language is said to be dynamic, if it alters its structure and behavior at runtime in a way, that static programming languages can only do at compile time. These alterations can include adding methods to objects, changing the class of an object, changing the implementation of an object or many other things. [Paulson, 2007]

Most dynamic languages are also dynamically typed.

Most, if not all, of these dynamic languages are also dynamically typed. This means, that most of the type checking is done at runtime. Variables are, in contrast to statically typed languages, not bound to a specific type and can hold values of any type. Because of the huge flexibility of dynamic languages it would be substantially harder to enforce type rules at compile time. The types of objects a method can accept as parameters could even depend on user input and therefore be impossibly determined before runtime.

There are advantages as well as disadvantages associated with dynamic typing:

Advantages of dynamic typing

Flexibility As variables do not need to be set to an explicit type, code can be much more flexible. One can easily define methods to accept all kinds of parameters and handle their differences appropriately.

Productiveness Flexibility and missing type declarations result in less code to write and thus can lead to a higher productiveness simply by decreasing the typing work of the developer. [Church et al., 2008]

Ease of learning The missing type related syntax leads to less learning effort. No commands related to type

[6] http://www.nist.gov

declaration or casting need to be remembered. [Warren, 2004]

Intuitiveness Often dynamic typing seems more intuitive. If a method depends on its parameter values to define some other method, one can simply assume they do. In most statically typed languages this would require special techniques. Java, for example, has the concept of *interfaces* for this, which results in another layer of complexity. [Warren, 2004]

Disadvantages of dynamic typing

Type safety In dynamic languages, type errors are not detected by a compiler. This can result in less stable programs. According to Tratt and Wuyts [2007] however, type related errors are rare in production code for dynamically typed languages and most errors — i.e. division by zero, off-by-one — cannot be caught by a type system anyway.

Interfaces are less clear In statically typed languages, explicit type declaration can also provide a form of documentation. The signature of a method defines what parameter types it accepts. As there are no parameter types in dynamically typed languages, the developer has to find out what values are suitable parameters differently. Descriptive identifiers are of great help here.

IDE Support Because many aspects of a program are not determined before it is run, static analysis of the source code results in less information when using dynamic languages. Therefore, it is much harder for IDEs to assist their users in refactoring tasks. Renaming a method in a Java program can be done almost automatically with an IDE like Eclipse, that replaces all references to the methods old name with its new name. In a dynamic language, the parser usually does not know enough about the programs structure to allow features like this.

Most dynamic languages are interpreted.

Most dynamic languages are interpreted and not directly

compiled to machine code. This is a little ambiguous, because in the end they result in machine code too, of course — otherwise they could not be executed. The difference to *compiled languages* is, however, that there is no explicit compilation period. Program code instead is analyzed and run by an interpreter every time the program is executed.

Dynamic languages rise in popularity.

While dynamic languages were often seen as amateurish, prototyping languages in the past [Paulson, 2007], today many big companies make use of them. Google heavily uses and promotes Python, Twitter is built on Ruby on Rails, a web-framework written in Ruby, and SAP is creating an own Ruby interpreter to run on their ABAP stack – the environment which SAP business applications are built around. The use of dynamic languages for large web projects also increased their acceptance in enterprise environments.

2.2.1 Ruby

Ruby[7] is a dynamic programming language. It was developed by Yukihiro "matz" Matsumoto and published in 1995. Since then it has gained a lot of popularity, especially since the release of the Ruby on Rails[8] framework in 2003. It builds on some principles:

Everything is an object In Ruby, every piece of information is an object. This makes it possible, for example, to add methods to numbers, which in many other languages like Java are primitives. This consistency adds to the simplicity of Ruby and makes very readable code possible. The code

```
3.times { puts 'hello' }
```

does exactly what it reads: it puts the string "hello" 3 times to the screen.

Flexibility As developers can alter almost everything, Ruby is very flexible. One can reopen classes and

[7] http://www.ruby-lang.org
[8] http://www.rubyonrails.org

alter their behavior. The following example adds a method `scream` to the core class `String`, that prints the string followed by an exclamation mark:

```
class String
  def scream
    puts self + "!"
  end
end
```

The existing behavior is not affected by this alteration. All other methods remain in the class.

Principle of least surprise Ruby is designed to be very consistent and intuitive. The focus for designing the language was not on maximizing execution speed but programmers productivity and joy.

Duck Typing In Ruby, there is no static type checking. The type-requirements of objects are not explicitly expressed by specific type names, but implicitly by the attributes they are expected to have. A method that expects a parameter can be called with a parameter of any type, as long as it supports all operations the method will perform on it.

This is often described by the phrase *"If it walks like a duck and quacks like a duck, it must be a duck."*, leading to the name *Duck Typing*.

Because of the flexibility to alter almost all behavior and the very dynamic type system, static analysis of Ruby source code is very hard. This is why test-driven development is widely used among Ruby developers.

2.3 Software Testing

Definition: *software testing*

SOFTWARE TESTING:
According to Myers and Sandler [2004], "Testing is the process of executing a program with the intent of finding errors."

A software tests purpose is to reveal errors.

As this definition implies, a test is considered successful, if

it reveals an error in the program. The overall goal of testing is to raise the programs quality and decrease the probability that it contains errors.

Testing usually makes up a big part of the software development process. According to Desai et al. [2009], typically 50% or more of a programming projects resources are spent on testing.

Test-driven development integrates testing into the whole development cycle.

In traditional development models like the Waterfall model testing is done after the implementation by a group that is potentially independent from the development team. In contrast, newer methods like Agile or Extreme Programming promote *test-driven development (TDD)*, where testing and implementation are done in parallel by the application developers. The typical TDD cycle is illustrated in Figure 2.1 and consists of the following steps:

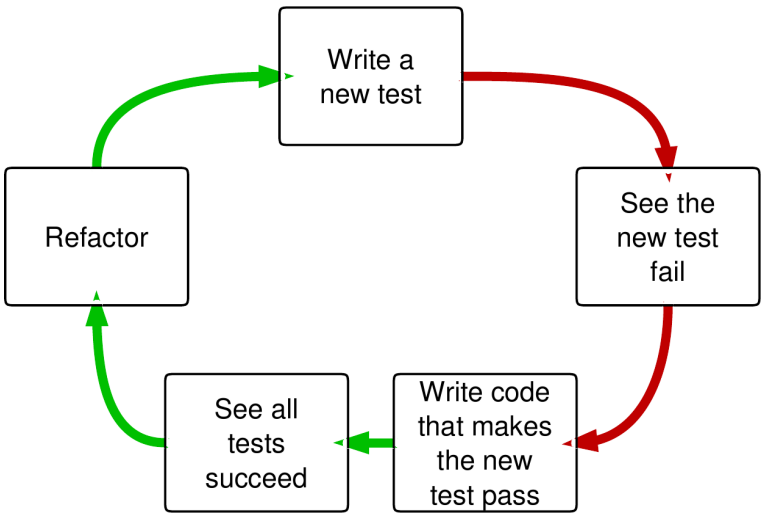

Figure 2.1: Test-driven development cycle

1. Before adding any implementation code, write a failing test. This makes sure that the new functionality does not exist yet. The test should be minimal and only test the new functionality, so that it clearly defines, what needs to be changed to make it pass.

2. Run the tests and see the new one fail. If it does not, either there is a mistake in the test or the functionality

already exists.

3. Implement code that makes the new test pass. This code should be as minimal as possible, to ensure a high test coverage. If the new test expects a function to return `0` for a given input, it is perfectly fine to implement the function to always return `0` first, before further tests really make a computation of the return value necessary.
4. Run the tests again and see them succeed.
5. If necessary, refactor the code to remove duplication. The existence of a passing test suite ensures that this step does not change the behavior of the implementation.

This means, that in TDD tests are written before functions are implemented and *implementation* code is only written when there are failing tests. This mostly results in a large test suite. Often there is even more test code than implementation code.

2.3.1 RSpec

RSpec[9] is a framework for test-driven development in Ruby. It allows developers to easily create and run tests. It was initially developed by Steven Baker in 2005 but soon handed over to David Chelimsky, who still maintains it today.

In TDD tests are executable specification.

The philosophy behind RSpec is called *Behaviour-Driven* or *Example-Driven Development* to emphasize that in TDD tests actually are executable specifications or examples of the programs intended behaviour. Test cases are usually referred to as examples and what other frameworks call "assertions" is termed "expectations" in RSpec. This is also visible in RSpecs syntax, which aims to be close to specifications in natural English language:

RSpecs syntax encourages writing of very readable examples.

A `describe` block groups specifications for one subject.

[9] http://rspec.info

```
1 describe 'GET new' do
2   it 'should render the new template' do
3     get :new
4     response.should render_template(:new)
5   end
6 end
```

Listing 2.1: RSpec syntax

Line 1 in listing 2.1 begins an example group for the subject `GET new`. Line 2 then begins one example or specification for this subjects behaviour. The name of this example — `should render the new template` — clearly expresses how the subject should behave. Lines 3 and 4 then perform the actual test logic. In line 3 a *GET* request to the `new` action is performed, while line 4 tests whether the response renders the expected template file. RSpec provides convenient methods with names resembling natural language for checking expectations. The expression `response.should render_template(:new)` checks, whether the template used for rendering the response is identified by the given parameter (`:new`).

RSpec is widely adopted in the Ruby community. In a survey[10] taken by more than 4000 developers, 39% of the participants stated that it is their preferred testing framework. 16% preferred Test::Unit, the framework that comes with the Ruby Standard Library and ranked second.

39% of Ruby developers prefer RSpec as testing framework.

Because of its popularity in the Ruby community, our prototype builds on RSpec.

2.3.2 Classification of Tests

Bourque and Dupuis [2005] suggest different dimensions for the classification of software tests. The one most interesting for our topic is the classification by target. Tests can be classified by the components they test:

Unit tests Unit tests are the most fine-grained tests in this classification. Their targets are the smallest pieces of

[10] http://survey.hamptoncatlin.com/survey/stats

code that can be tested separately – often methods, classes or modules. They are typically implemented by the developers with access to the code they test.

Integration tests Integration tests verify the interaction between several smaller pieces of code, that might already have been tested by unit tests. The level of abstraction is higher than for unit tests and depends on the components that are subject of the test.

System tests System tests verify the behavior of a whole software system. At this point, most failures should already have been identified by unit and integration tests.

Unit tests are best suited for our tool.

Because they are often implemented prior to all other classes of tests, our tool will be most useful with unit tests, where *simple* bugs like syntactical ones are likely to be found. And because the volume of code they cover is supposed to be minimal, similarities to other code and possible solutions to errors should be easier to discover than in integration and system tests, where much code has to be inspected to find related bug fixes.

Chapter 3

Related work

This chapter summarizes previous approaches to aid the debugging process. We describe examples of work on automated debugging, providing useful information for the users current task and different attempts to make it easier for the user to understand the source of a bug.

We will also talk about HelpMeOut, which our tool heavily bases on.

3.1 HelpMeOut

Hartmann et al. [2010] introduced HelpMeOut, a tool that suggests crowdsourced solutions to programming errors for the Processing and Arduino environments. It consists of IDE plugins and a central server.

When a programmer using the HelpMeOut plugin encounters either a compiler error or a runtime exception, the plugin queries the central database for examples of related problems that others successfully fixed. These examples then are presented to the developer, who can apply them to his own code.

In contrast to other tools employing hard-coded strategies of solving errors, the fixes in HelpMeOut are collected from

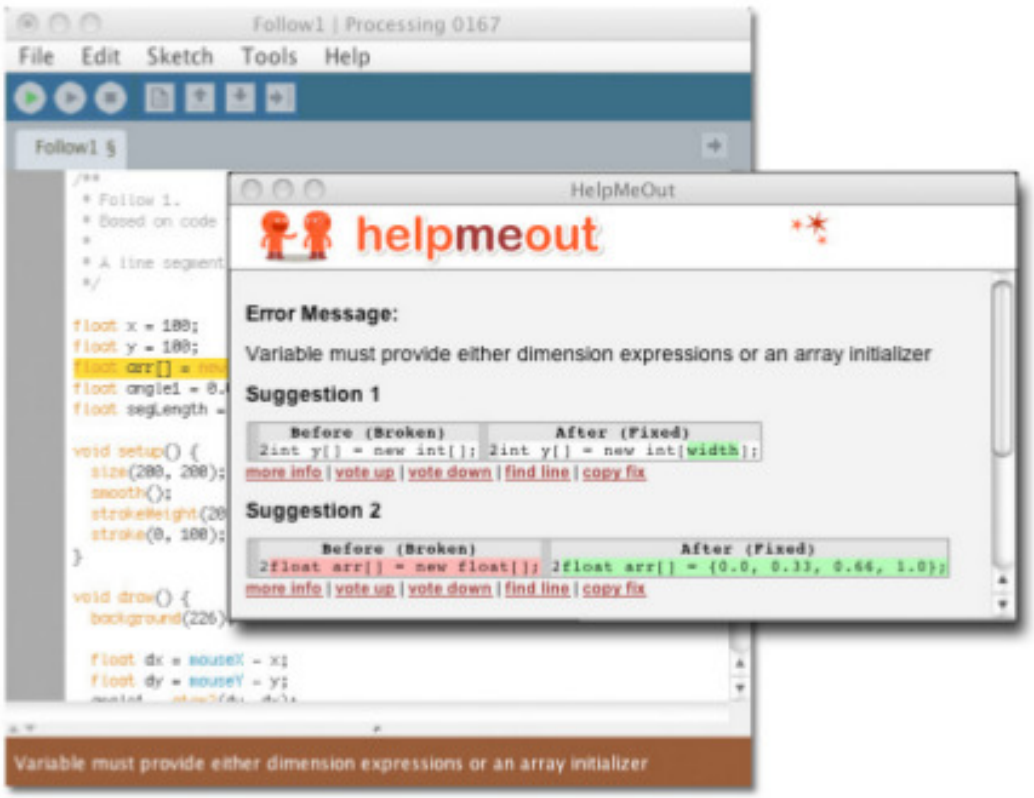

Figure 3.1: Screenshot of HelpMeOut suggesting fixes

programmers using the plugins. Whenever HelpMeOut notices an error has been solved, it sends a diff of the affected files to its database and makes it available as a possible suggestion for others.

In a study amongst novice programmers, HelpMeOut could suggest useful fixes for 47% of errors after 39 person-hours of programming.

Hartmann et al. [2010] describe the problems related to deciding whether a runtime error has been fixed. Runtime errors can depend on user input, the current time or other dynamic variables. Because of that, it is not possible to decide whether a bug is fixed by simply watching whether a given line of code executes without an exception being thrown. This is why HelpMeOut employs a progress heuristic to catch a subset of these runtime exceptions.

In dynamic languages this problem is even more prominent. There is no compiler to catch errors before the programs execution and so all errors are runtime errors. This is a reason for us to utilize a test framework. In contrast to deciding whether a bug is fixed or not, it is easily decidable whether a test passed or failed.

3.2 Automated Debugging

Automated Debugging tries to take a lot of the effort of fixing broken code from the developer by automatically providing bug fixes. Noteworthy research in this field includes:

3.2.1 ReAssert

In software projects with a large test suite, even minor changes in the implementation code can make many tests fail. Daniel et al. [2009] developed the Eclipse plugin ReAssert to fix these tests with several strategies. ReAssert for example compares expected and actual values of assertions and can change the test to expect the correct value. ReAssert is not intended to fix actual implementation bugs, but solely to change the tests and make them pass.

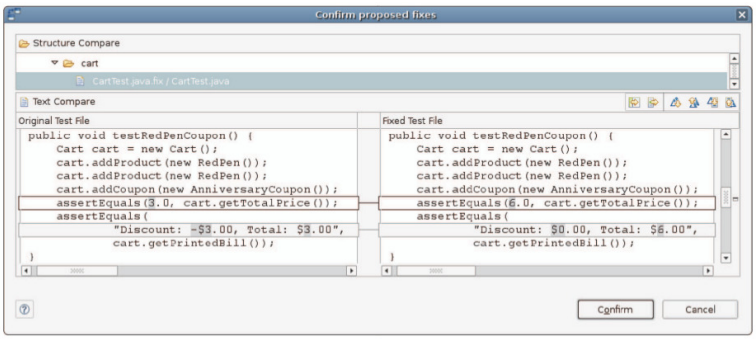

Figure 3.2: Screenshot of ReAssert suggesting a fix for a broken test

3.2.2 AutoDebug

AutoDebug is an algorithm to locate and correct erroneous statements in a programs source code. Modifications of the source code are computed by different strategies and tried until one is considered successful or none are left. It requires a test that validates the bug as well as pre- and post-conditions of the function containing it in first order predicate logic. He and Gupta [2004] implemented AutoDebug for a subset of the C language and were able to fix most bugs in their test programs.

3.2.3 BugFix

BugFix [Jeffrey et al., 2009] suggests possible solutions to programming errors from a knowledge base. Machine learning techniques are used to improve the suggestions for a given bug. Once a bug is fixed, developers are able to enter a new bug fix description into the knowledge base.

3.3 Finding related resources

Finding resources related to an error can be challenging. Searching the web is not trivial (see 4.1.2—"How bugs are solved today") and even project specific repositories can contain a huge number of artifacts like bug tickets, documentation or the actual source code. We here present some tools that identify resources related to the developers current task to assist him in getting a deeper understanding.

3.3.1 Blueprint

Brandt et al. [2010] developed a plugin for Adobe FlexBuilder that integrates a web search interface into the IDE. Queries are augmented with the current code context and results are presented in a more code centric way(Figure 3.3). If the developer decides to adopt the presented code

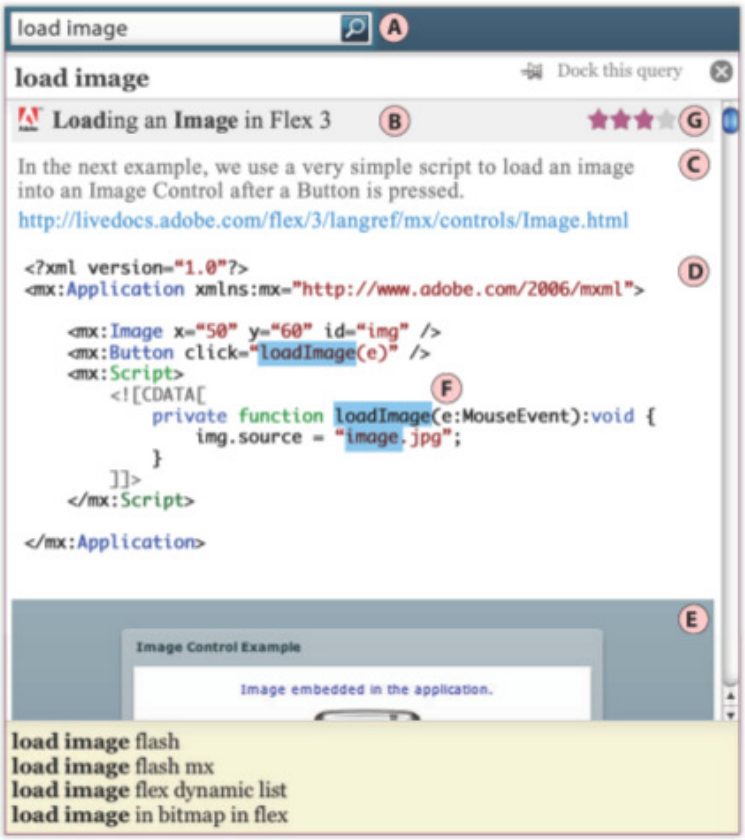

Figure 3.3: Screenshot of Blueprint showing the example-centric presentation of the search results

examples, the source code is linked to the web page containing the example. A study found significant improvements in code quality and programmer productivity when Blueprint was used.

3.3.2 DebugAdvisor

DebugAdvisor [Ashok et al., 2009] allows developers to search for information related to a bug with a *fat query* consisting of the whole context of the bug. This query can include natural language text, core dumps, debugger output etc. Results in their study at Microsoft come from all their software repositories including version control, debugger

logs, bug databases etc. As the results are linked to other related resources, navigating through the shared knowledge is easily possible. 75% of the artifacts DebugAdvisor returned during their study were considered useful by the developers.

3.3.3 Hipikat

Hipikat [Čubranić and Murphy, 2003] relates artifacts in software projects to make it easier for newcomers to get an overview of the project. The tool is implemented as an Eclipse plugin and lets users query by elements of their current workspace (a Java class for example). Hipikat then shows a list of related source code, bug or feature descriptions, mailing list messages or other project documents.

3.4 Helping understand the error

When confronted with an error, developers have to form a mental model of why it occurred. Compiler messages are often cryptic and challenging to understand. The following shows different approaches to lead developers towards a deeper understanding of the error.

3.4.1 Whyline

Whyline [Ko and Myers, 2008] helps developers answering why and why not questions about program behaviour. A trace of the program execution is generated and programmers can ask questions like *"Why did x=3?"* or *"Why didn't Frame appear?"* and view steps from the program execution related to their question. Figure 3.4 shows its interface while presenting events related to his question.

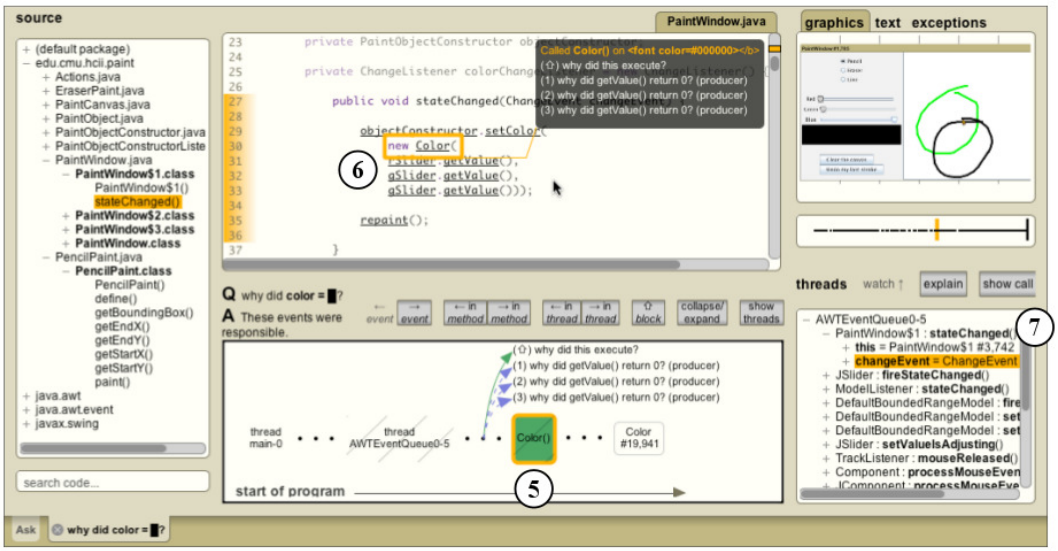

Figure 3.4: Whyline presenting the answer to a *Why* question.

3.4.2 Backstop

Murphy et al. [2008] developed Backstop, a tool to assist programming novices in fixing runtime errors in Java programs. It replaces exception output with more user friendly messages that also suggest how to avoid that error. These messages are provided by backstop and not collected during runtime.

3.5 Summary

All the approaches described above assist debugging by either automatically fixing errors or providing useful information for the developer. Our tool not only finds and presents examples of bug fixes related to the currently failing test, but also automatically collects these. It provides fixes that others applied to closely related errors in a test-driven development environment. In contrast to ReAssert, we do not try to alter only the tests to make them pass, but also the actual implementation code. What differentiates our tool from Blueprint is the step of explicitly searching for information. While Blueprint improves search results for

user-entered queries, our tool automatically queries whenever a test fails.

Chapter 4

Design

4.1 Motivation

Fixing errors makes up a big part of software development.

While writing a program, software developers usually encounter a lot of situations where they have to fix errors. Especially when TDD is employed, every new functionality first results in a failing test and as such in an error. These errors can be roughly divided into the following categories:

4.1.1 Categories of errors

Syntax Errors are invalid sequences of characters in the programming language. In compiled languages they cause the compiler to be unable to translate the source code. In interpreted languages however these errors are noticed during runtime, when the interpreter tries to parse the effected part of the source code.
An example for a syntax error in most programming languages are unbalanced opening and closing parentheses.

Semantic Errors also called logic errors, are errors that do not cause a program to be syntactically invalid, but to produce a wrong result. This could be an abnormal abortion or wrong output. While the first case can be easily detected and handled in most program-

ming languages, wrong program output often is not noticed, especially if no extensive test suite is used. An example for a semantic error that results in program abortion is the use of an invalid array index. Wrong output not resulting in an exception could be caused by rounding errors, that are not obvious from the source code.

Missing implementation is a special category of errors that occur, when test-driven development is used. When tests are written and executed prior to the implementation of the actual code, they result in an error, because the functionality they test does not exist yet. Strictly, these kinds of errors would also fit into above categories, but for our purposes, we will distinguish them from other syntax or logic errors.

TDD allows more bugs to be automatically detected.

While usually only syntax and some logic errors result in an exception and so can be easily detected, with a good test suite, errors of all the above categories can be caught programmatically. This gives us the chance to employ the HelpMeOut approach to more fields of errors.

4.1.2 How bugs are solved today

Developers use web search to find solutions.

Oftentimes a similar error that occurs to one developer has been made and solved by himself or someone else before. This is why many developers use a web search engine to find a solution for their problem if they do not immediately know how to solve it. Searching for source code however presents the developer some challenges:

First, the developer has to think of a proper search term. Obvious queries could consist of the exception message or the line of source code that represents the error. Each of these strategies has some problems:

Error messages describe symptoms, not the root of the problem.

Exception messages do not necessarily describe the root of the problem. A message like *"NoMethodError: undefined method 'to_str' for someVar:SomeClass"* could be caused by a wrong assignment to the variable someVar earlier in the

code, by a missing implementation of the method to_str, by a spelling error in "someVar" or many other reasons. Therefore, the exception message does not directly relate to the solution the developer wants to find.
As the above error message shows, it is also unclear what parts of the exception message to search for. It may contain variable or method names that are unlikely to be the same in other developers code. On the other hand, the method name may be relevant, if it is a standard method. The solution for above error could also be to cast the object to another class that implements the missing method.

The location of the error is not always known.

Searching for the line of code that caused the error implies that this location is known. This, however, is not always the case. If the error results in an exception, the stacktrace might point to a location in the source code where the exception was thrown. The real problem, however, could be for example a wrong variable assignment earlier in the code.

Search engines are not optimized to help debugging

Also, most search engines are optimized for natural language queries and cannot relate similar source code fragments. Specialized source code search engines like Google Code Search[1] usually index repositories of working code and as such are unlikely to return good results when queried with a line of broken code.

4.1.3 Participation Inequality

Most content in online communities is produced by only 1% of users.

Only a tiny minority participates in online bulletin boards by providing solutions to problems. According to Nielsen [2010], 90% of users of most online communities do not produce any content, 9% produce a little and only 1% is responsible for most of the content. Most people either only search for their problem or ask questions and wait for others to reply. Participants in our interviews noted that they do not like to ask questions at bulletin boards because they do not have the time to wait for somebody to reply.
Because of this participation inequality, a lot of knowledge is not accessible to others. While many people probably would not mind sharing their knowledge, they simply lack

[1] http://google.com/codesearch

time and motivation to do so.

With HelpMeOut a technique to collect instances of bug fixes and later suggest them to developers was introduced. We apply this technique to TDD in dynamic languages.

4.2 Design studies

Design was guided by eight interviews with users of dynamic languages.

To help us design a tool that fits todays work practice and is most helpful to developers, we conducted eight interviews with developers using dynamic languages. Four of these interviews were traditional question and answer interviews and four of them were contextual inquiries (CIs). Because they enable the designer to actually watch the work process rather than rely on peoples description of it, we assume contextual inquiries to result in more and more precise data than common interviews. Contextual inquiries are very time consuming and subjects are easier to convince to participate in a usual survey, so we conducted four contextual inquiries and augmented them with data from four additional interviews.

4.2.1 The Contextual Inquiries

Contextual inquiries were performed at the users workplaces.

The contextual inquiry method as described by Beyer and Holtzblatt [1997] has the goal of creating a shared understanding of the users task by herself and the designer. Therefore, the designer collects data at the users work environment by watching and getting explained what the user does. For this process, the apprenticeship relationship model has proven to be useful.

Apprenticeship Model

A familiar model of the relationship between designer and user gives both of them the possibility to behave in a natural way without thinking too much about the proper way to

behave in an interview. Beyer and Holtzblatt [1995] suggest the relationship between master and apprentice as such a model for the following reasons:

Users are not teachers Usually, the user has no teaching abilities. This matches the craftsman, that also does not have a teaching education but still manages to teach his apprentice. This is done by simply doing a task and explaining what he is doing.

Recalling is harder than doing Doing the work is often easier for the user than recalling how he did it in the past. Steps of his task might have become habitual and are done without thinking about them. When people have to recall their actions, these steps might get lost. Explaining what they are doing while they do it often also gives users a chance to stop and think about their work, which is not natural in the normal workflow and can result in revealing problems and ways to improve their work situation.

Recalled situations lack details When users have to recall how they work on a task, they usually abstract over all the times they did that task. This way, a lot of details get lost. They might also think that specific details are not important to the designer and leave them out of their descriptions.

Recalled situations lack divergence To design a useful product, the designer needs to find a structure in the work of the users. Therefore he abstracts over all of the situations he observed at different users. If the users recalled their behaviour and already abstracted over it themselves, important structural similarities between different users might get lost.

Artifacts serve as reminders When working on a task, the involved artifacts like a handbook, the keyboard or a spreadsheet remind the user of events related to them. *"Last time I worked on that spreadsheet, I was struggling with..."*. Recalling these events while working on something similar helps to emphasize the differences between the two times the user did that task and so provides more details.

Designer has to understand the users work structure.

Of course there are also differences to a real master and apprentice relationship that a designer has to keep in mind. The designer is not really interested in learning to do the work. The purpose of his apprenticeship is to gather data about the structure of his masters' work. It is therefore his responsibility, to guide the master to a direction that is useful for that purpose. He must also articulate his understanding of the work structure, so the user can correct him. In contrast to an apprentice, it is not sufficient for a designer to be able to copy what he sees. To find a structure in the users work, he has to really understand it. It is also a good practice to directly discuss ideas of improvement with the user. This ensures that the designers interpretation of the users work structure is adequate and leads to a first feedback and maybe also to suggestions for different ways of improving the design.

Data gathered in a CI is usually more precise, as the user does not have to remember how he solved a task in the past or even make up how he would do something fictional. Instead the interviewer watches him doing the actual task and gathers whatever data is helpful in designing the product. This also often leads to completely new ideas, because the designer can watch things he might not have thought about before at all.

4.2.2 Own Contextual Inquiries

To find subjects for our design studies we searched German online job listings for companies looking for Ruby developers. We were allowed to visit two companies to conduct contextual inquiries with their developers. A message to the universities mailing list gave us two more subjects to interview.

This way, we had two subjects working professionally with Ruby at companies that specialized in that field as well as two subjects that were hobby developers and described themselves as beginners. Experience with Ruby was between 4 and 5 years among the professional developers and less then 1 year among the beginners.

During the contextual inquiries we followed the apprenticeship model. Subjects were instructed to work as usual but explain what they are doing while working. During these sessions of about 2 hours each, one of us watched the subjects and took notes about their work structure. If anything needed more explanation, we interrupted to ask questions.

Relevant findings of the CIs were:

Test-driven development While professional developers used TDD, beginners did not. They knew about it and had an idea how it would be applied, but simply did not have a project large and complex enough to feel the need for it. Both beginners stated that they were curious about TDD and would like to try it some day.

While TDD is common amongst professionals, the beginners we interviewed did not use it.

Amongst professional Ruby developers TDD seems to be standard, though. Besides both companies we visited using TDD, discussions on Ruby web pages, conference topics and job listings very frequently mention testing frameworks or TDD practices and imply that this is highly adopted amongst professional Ruby developers. In a 2010 survey[2] amongst more than 4000 Ruby developers, more than 85% of the participants stated that automated testing was either "required" or they at least "do it often". Only about 15% answered "don't do it".

As a consequence of the large adoption, we decided to facilitate a TDD framework to collect bug fixes. It is not clear how to decide when an error is solved without a test or compiler indicating so. There are many publications proposing TDD to be included in early programming education ([Desai et al., 2009], [Schaub, 2009], [Spacco and Pugh, 2006]), which could benefit from such a tool. While we limit the collection of bug fixes to a testing framework, suggesting these outside of this context could be easily implemented in a future project.

Exceptions vs. "silent errors" Most of the errors developers encountered during our interviews were logical

Silent failures are very common.

[2] http://survey.hamptoncatlin.com/survey/stats

ones that did not result in an exception but in wrong program output . One of the professional developers, for example, was working on a JavaScript application that received time entries from a server and displayed them in a calendar. The beginning and end of a time entry were supposed to have a different color than the rest of it, which was not the case. Errors like this do not result in an exception but can be caught by tests.

This shows that without a testing framework, we would not be able to detect most errors in dynamic languages. Tests give us the possibility to turn silent failures into exceptions that we can catch and react on.

Common strategy of bug fixing

Sequence of actions When developers encountered an error, depending on its kind they all followed a similar sequence of actions.
First they tried to find out what place in the source code the error originated at. If an exception was thrown, they used the backtrace to get the file and line number. If there was no exception, they thought about what methods were related to the misbehaviour and inspected those.
When TDD was used and the error was in existing code, a test was written that would catch the bug.
When they identified the source of the bug, they either immediately knew how to solve it or took some time to think about it and tried some variations of their existing code.
If this did not lead to a result, they began looking for help. They either asked their colleagues, looked at some documentation or used Google to search for related errors.
For all the errors we encountered, this sequence eventually led to a solution.

Our tool will assist in the "looking for help" step. By automatically suggesting fixes, we free the developer from formulating a search query. If used internally in a company, we can also suggest a colleague that experienced a related mistake before and might be able to help.

Workplace situation All the beginner developers we interviewed work on their own. Professional program-

mers worked in small teams of 3-5 developers together in an office. Both companies we visited also employed remote coworkers that telecommuted via Skype. Especially at one of the companies, the lead developer spent a lot of his time answering questions of his remote colleagues via Skype, some of which were also about programming problems.

4.2.3 Traditional interviews

To back up the data we gathered during the contextual inquiries, we conducted four more interviews in a traditional question and answer style. Subjects were referrals from students at Berkeley using different dynamic programming languages and their experience ranged from a few month to 6 years.

Bug fixing strategy confirmed in traditional interviews.

These interviews confirmed our findings from the contextual inquiries. Especially the action sequence described above was approved here, too. No matter which programming language was used, if developers did not immediately find the solution to their problem, they mostly started a web search.

Subjects also noted, that they seldom actively generate content that might be helpful for others. One subject stated, that he uses Google and hopes that the search results reference a page from Stackoverflow[3] . The same person, however, said that he would not ask or answer questions himself, because it takes too much time. When he is looking for information, he wants to find it immediately and not wait for others to reply.

Client side interpreters provide less information about errors than server side interpreters.

The interviews also revealed that errors are harder to locate on the client side, as the information browsers give when an error occurs is typically less than the information interpreters for server side languages give and especially often misses the location in the source code where the error originated. There seems to be no widely adopted TDD framework in client-side programming. This suggests that

[3] http://www.stackoverflow.com

a server-side language like Ruby might be more promising for our tool.

4.2.4 Design Decisions

Programming Language

As web browsers give less information about the source of errors, the HelpMeOut approach at this point does not seem to fit client-side languages. Programmers usually use their semantic understanding and knowledge of what specific methods do to locate bugs in their source code. Without semantic understanding, there seems to be no way for HelpMeOut to know which part of a program to compare to collected fixes to find a solution for the current problem. Semantic equivalence among code pieces however is an undecidable problem, so we considered it best to implement HelpMeOut for a server-side language.

Ruby as underlying programming language

As we had some knowledge about and experience with Ruby, we chose this language. It is easily extensible and test-driven development is highly adopted. Most large open source Ruby projects require each patch contributed to contain unit tests (see Rails[4] , Gemcutter[5] , Sinatra[6] for examples).

Interface

No standard IDE for Ruby developers

In contrast to other languages like Java, where IDEs like Eclipse or Netbeans are used by most developers, there is a very wide choice of editors among Ruby programmers. This is probably caused by the dynamic nature of Ruby, that makes automatic refactoring — one of the main benefits of such IDEs — very hard.

Results presented in web browser

For this reason, implementing HelpMeOut as a plugin to some editor seems not to be reasonable, because this way

[4] http://edgeguides.rubyonrails.org/contributing_to_rails.html
[5] https://github.com/rubygems/gemcutter/wiki/contribution-

we would strongly limit the number of potential users. As a web browser should be installed on most developers computers, we chose to implement the HelpMeOut interface as a web page.

Testing Framework

According to above mentioned survey[7] , RSpec seems to be the most used testing framework for Ruby. The mailing lists, websites and the source code of Ruby projects at Github[8] we used while working on this prototype also very frequently mentioned or employed RSpec.

As RSpec also provides a mechanism to write custom formatters for its output, we chose HelpMeOut to use RSpec as its underlying testing framework. This gives us a good handle on the test results while at the same time providing an interface to easily add to the test runners output.

guidelines

[6] http://www.sinatrarb.com/contributing

[7] http://survey.hamptoncatlin.com/survey/stats

[8] http://www.github.com

Chapter 5

Prototype

5.1 Requirements

To aid test-driven development, there are several requirements we defined for our tool. We divided these by whether they help us collecting fixes or whether their purpose lies in suggesting these fixes to the user.

5.1.1 Collecting fixes

To collect fixes for failing tests, our tool first has to notice when a test fails. It then should be able to identify this test and recognize when it is fixed.

When a formerly broken test is fixed, our tool should identify what changes lead to the test passing. These changes in the source code should then be stored at a central server, together with data identifying the error.

5.1.2 Suggesting fixes

When a test fails, our tool should query the database of collected fixes and return the most relevant ones.

Presentation of the suggested fixes should be in a way that allows the programmer to easily spot the affected pieces of source code.

5.2 Overview

5.2.1 Walkthrough

This section will provide a quick example of how HelpMeOut could be used in practice. In the following scenario, consider two developers working on two unrelated Ruby on Rails projects in different physical locations.

Developer A implements a blogging application and currently works on the interface for creating new blog posts. According to the Rails conventions, he decides that whenever a *GET* request to the action called `new` is issued, the template named `new` should be rendered. Because he works in a test-driven way, the first thing he does is to write the test from Listing 5.1.

```
1 describe 'GET new' do
2   it 'should render the new template' do
3     get :new
4     response.should render_template(:new)
5   end
6 end
```

Listing 5.1: Developer As RSpec test

As the functionality is not implemented yet, this test case fails with an `ActionController::UnknownAction` exception.

He then implements the missing functionality. Because Rails automatically renders a template with the same name as the action, the solution in this case is to simply create a method named `new`:

```
1   def new
2   end
```

Listing 5.2: Solution to the test case from Listing 5.1

He runs his test again and sees it passing. HelpMeOut realizes that this test failed before and generates a new fix for its global database, consisting of the lines in Listing 5.2.

Developer B now works on a Ruby on Rails application to manage his DVD collection. He wants a page to list all his DVDs and names the method to generate it `index`, again according to Ruby on Rails conventions. He writes the RSpec test case in Listing 5.3 to ensure that this action assigns an instance variable `@dvds` for use in his template file.

```
1 describe 'GET index' do
2   it 'should assign all my dvds' do
3     get :index
4     assigns[:dvds].should == @all_dvds
5   end
6 end
```

Listing 5.3: Developer Bs RSpec test case

As the action `index` is not defined yet, this test will also fail with an `ActionController::UnknownAction` exception. HelpMeOut queries its central database for fixes and finds the one from Developer A. Developer B sees, that others have added a new method to solve a similar problem, so he tries that. He knows that his method needs a different name, because he is not currently working on a `new` action, and creates a method named `index`.

Because his test also checks whether the variable `@dvds` gets correctly assigned, his test will still fail, this time with a `Spec::Expectations::ExpectationNotMetError`. He continues implementing functionality and alters his code to Listing 5.4.

```
1 def index
2   @dvds = Dvd.all
3 end
```

Listing 5.4: Solution to Developer Bs RSpec test case

When his test passes now, HelpMeOut again realizes that it failed before and stores a new fix, consisting of the addition of line 2 in Listing 5.4.

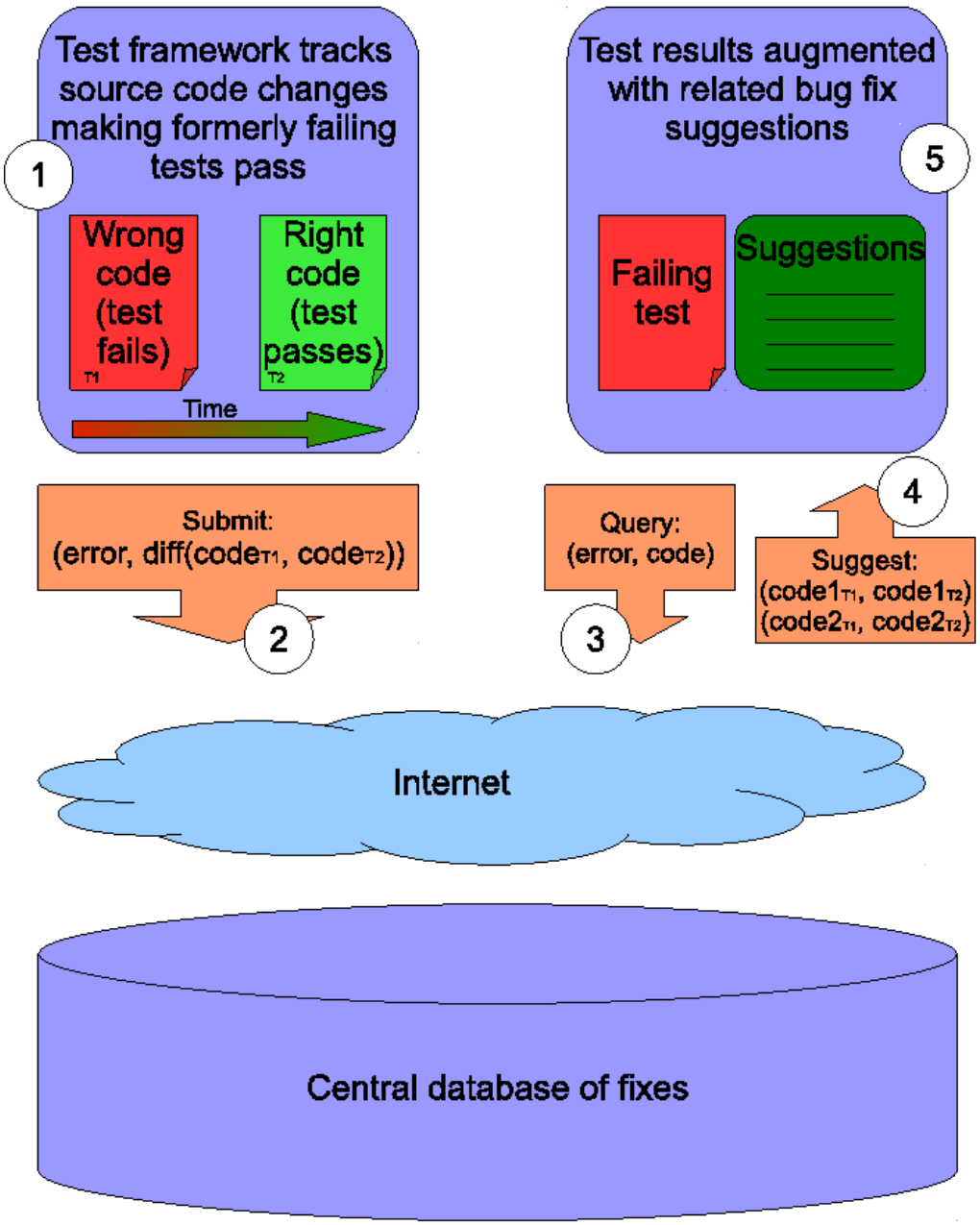

Figure 5.1: HelpMeOut leverages a test framework to collect and suggest bug fixes: ① Test executions are monitored for changes that lead to a formerly failing test passing. ② When such changes are noticed, a new bug fix description is generated and submitted to the central server. ③ When a test fails, a query consisting of the error data and source code fragments is sent to the server. ④ The server replies with suggestions of what others did to solve similar problems. ⑤ These suggestions are then displayed inside the test frameworks output.

5.2.2 Architecture

HelpMeOut is designed according to the client-server model.

CLIENT-SERVER MODEL:
The client-server model describes a software architecture in which many clients request services from one central server.

Definition:
client-server model

The client comes in the form of a Ruby Gem and integrates into the RSpec testing framework. It is responsible for collecting and presenting the fixes. When a fix is made at the client-side, it is send to the server. When a failing test is noticed, the server is queried for matching fixes, which then would be presented to the developer.

RUBYGEMS:
RubyGems is a package manager for Ruby libraries. These library packages are called Gems and are easily installable via the `gem install` command.

Definition:
RubyGems

The servers task then is to find fixes related to the given query. Fixes are retrieved by exact matches of the exceptions classname and a processed backtrace. This can result in many fixes, so they are ranked by string distance of the exception message and processed source code, before the 5 highest rated ones are send back to the client.

From the users perspective, nothing changes besides of the augmented output of the test results. He still runs his tests with the `rake` command, waits for them to complete and reviews the output. If there are failing tests, HelpMeOut tries to find related fixes and adds them to the test runners output (see Figure 5.2).

5.3 Client

RSpec provides an interface to register custom formatters.

Implemented as
RSpec formatter

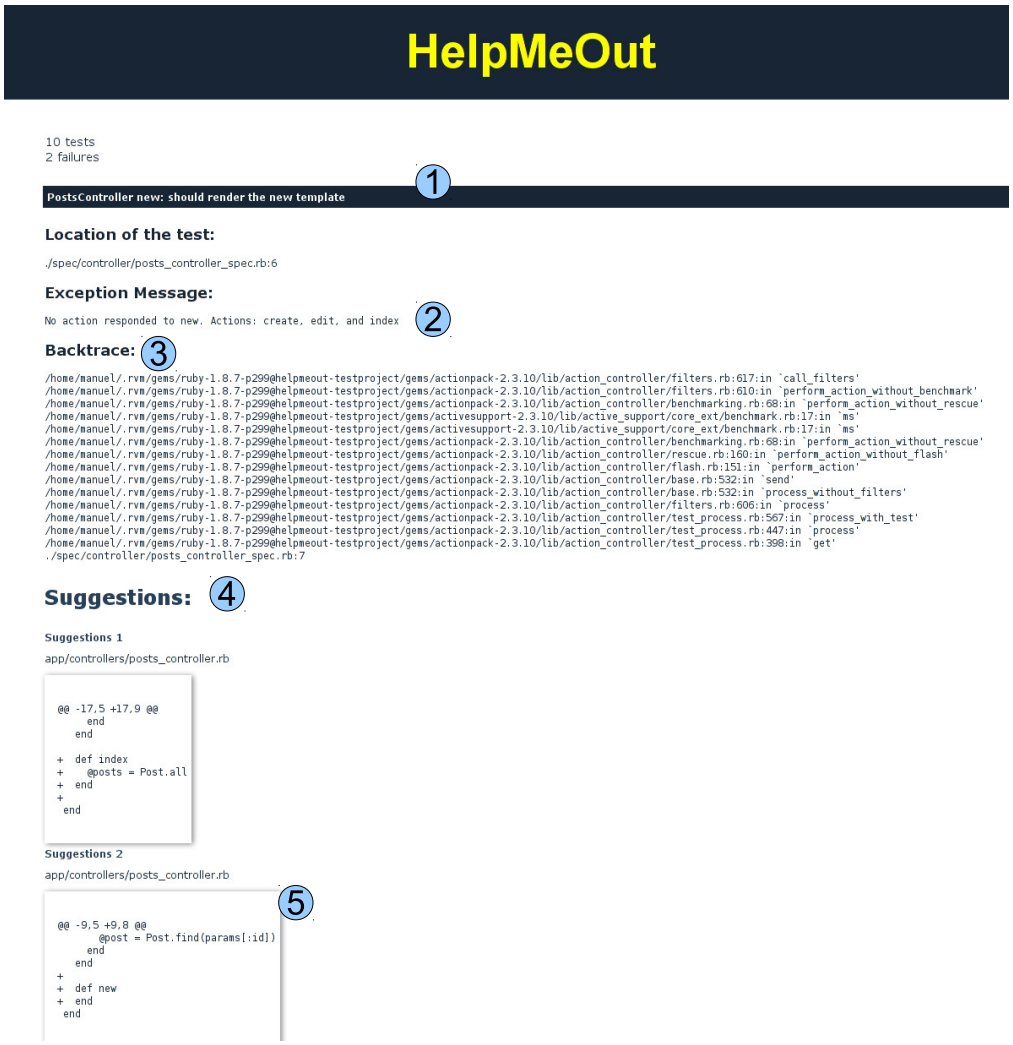

Figure 5.2: The HelpMeOut interface suggesting fixes to a broken test. ① shows the name of the failing test, ② and ③ show the exception message and backtrace to help understanding the error. HelpMeOuts suggestions start at ④ , containing one related fix and the exact fix in second position, at ⑤ .

These are usually used to present the output of the test runs in different ways. The standard formatter outputs the test results to the console, but there are others that generate a web page or integrate into an editor. Whenever a test fails or succeeds, a method on the formatter is called with arguments containing the name and location (file and line number in the source code) of the test and, if the test failed, the exception and the backtrace.

As this gives us access to the information we need about the test runs and lets us present the test results together with our fixes, we implemented the HelpMeOut client as a custom RSpec formatter. It serves two purposes:

5.3.1 Collecting fixes

Failing tests are stored in a local database.

To collect fixes, HelpMeOut watches the test execution for failing tests. Whenever a failing test is noticed, a new local database entry containing the name of the test, the exception that was thrown and the backtrace is created.

When the HelpMeOut client notices a passing test, it queries its local database of formerly failing tests for an entry matching the passing tests name. If it finds one, this suggests that the test failed before and is fixed now. We then need to find out what files changed since the test failed.

Git is used to track changes to files.

For this the HelpMeOut client uses the Git version control system. After every run of the whole test suite, we commit all the changes to a local git repository. When we encounter a fixed test, we ask git for the files that changed since the last commit. We then add the latest committed version as well as the version on the file system of these files to the data that is sent to the server.

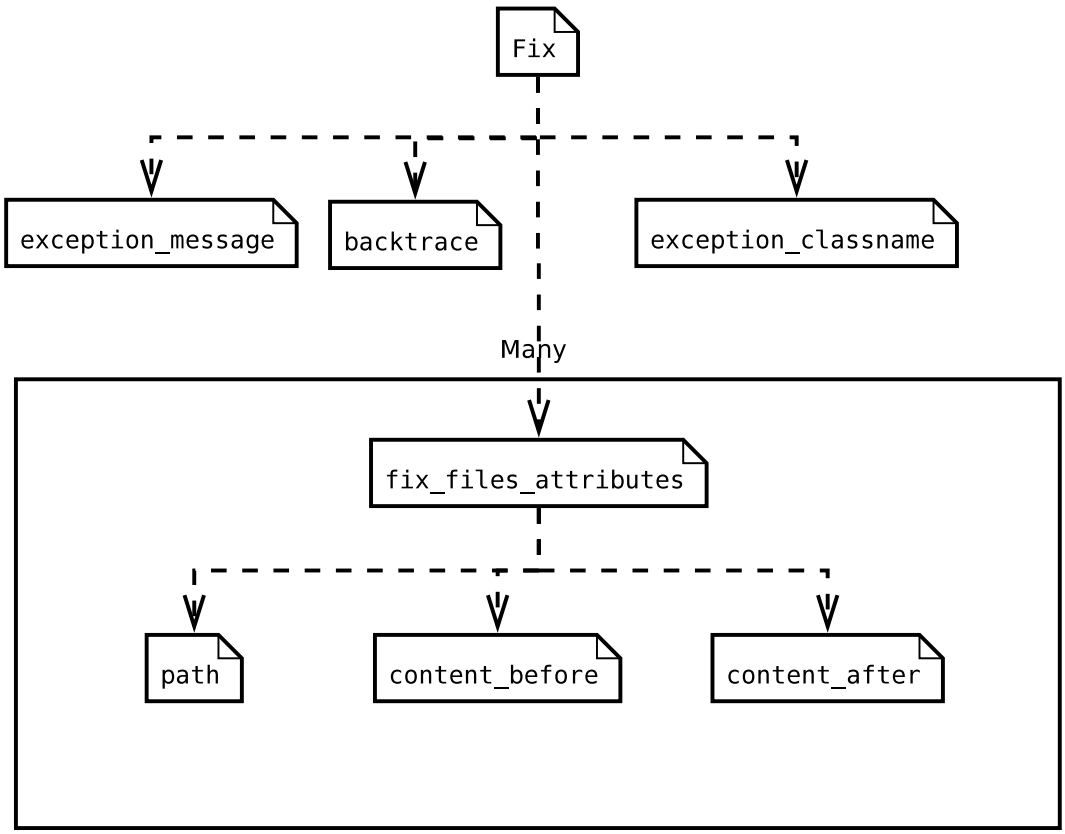

Figure 5.3: Structure of the XML data sent to the server as a new fix

Definition: *Git*

GIT:
Git is a distributed version control system. It was initially designed for Linux kernel development by Linus Thorvalds. As every working copy contains the full version history and has full revision tracking capabilities, it is not dependent on network access or a central server.

To later match this fix with other programmers problems, further data is needed. We also send the exceptions message and classname as well as the generalized backtrace. The structure of the sent XML data is shown in 5.3

User-specific differences are removed from the backtrace.

Generalization of the backtrace is necessary, because it usually contains absolute paths to locations of installed libraries, which can vary between different users. We also need to ensure that it does not contain paths to project specific files, which are unlikely to be the same in other

projects. This is done in three steps:

```
1  /home/manuel/test_project_2/app/controllers/posts_controller.rb:16:in 'show'
2  /usr/lib/ruby/gems/actionpack-2.3.10/lib/action_controller/base.rb:1333:in 'send'
3  /usr/lib/ruby/gems/actionpack-2.3.10/lib/action_controller/base.rb:1333:in 'perform
       _action_without_filters'
4  /usr/lib/ruby/gems/actionpack-2.3.10/lib/action_controller/filters.rb:617:in 'call
       _filter'
5  /usr/lib/ruby/gems/actionpack-2.3.10/lib/action_controller/filters.rb:610:in '
       perform_action_without_benchmark'
6  /usr/lib/ruby/gems/actionpack-2.3.10/lib/action_controller/benchmarking.rb:68:in '
       perform_action_without_rescue'
7  /usr/lib/ruby/gems/activesupport-2.3.10/lib/active_support/core_ext/benchmark.rb
       :17:in 'ms'
8  /usr/lib/ruby/gems/activesupport-2.3.10/lib/active_support/core_ext/benchmark.rb
       :17:in 'ms'
9  /usr/lib/ruby/gems/actionpack-2.3.10/lib/action_controller/benchmarking.rb:68:in '
       perform_action_without_rescue'
10 /usr/lib/ruby/gems/actionpack-2.3.10/lib/action_controller/rescue.rb:160:in '
       perform_action_without_flash'
11 ./spec/controllers/posts_controller_spec.rb:10
```

Listing 5.5: Example of an unprocessed backtrace — user- and project-specific parts in orange

- First, all relative paths are expanded. This makes comparison in the next steps easier. In 5.5 this would affect line 11, which would be `/home/manuel/test_project_2/spec/controllers/posts_controller_spec.rb:10` afterwards.

- The next step ensures that differences in the location of installed libraries are ignored. For that, the HelpMeOut client provides the configuration option `exclude_paths`, which for the example backtrace in 5.5 should point to `/usr/lib/ruby`. Every occurrence of an exclude_path at the beginning of a line in the backtrace is then replaced with the string "EXCLUDE". For line 2 in 5.5 this would result in `EXCLUDE/gems/actionpack-2.3.10/lib/action_controller/base.rb:1333:in 'send'`.

- Finally, all lines referring to project specific files are removed from the backtrace. These are determined by a configuration option `project_root`, which should

point to the projects root directory. Paths to files in this directory are unlikely to be the same between different users. In the example above, this would affect lines 1 and 11, which would not be part of the processed backtrace anymore.

The generalized backtrace of 5.5 is shown in 5.6.

```
1 EXCLUDE/gems/actionpack-2.3.10/lib/action_controller/base.
      rb:1333:in 'send'
2 EXCLUDE/gems/actionpack-2.3.10/lib/action_controller/base.
      rb:1333:in 'perform_action_without_filters'
3 EXCLUDE/gems/actionpack-2.3.10/lib/action_controller/
      filters.rb:617:in 'call_filters'
4 EXCLUDE/gems/actionpack-2.3.10/lib/action_controller/
      filters.rb:610:in 'perform_action_without_benchmark'
5 EXCLUDE/gems/actionpack-2.3.10/lib/action_controller/
      benchmarking.rb:68:in 'perform_action_without_rescue'
6 EXCLUDE/gems/activesupport-2.3.10/lib/active_support/
      core_ext/benchmark.rb:17:in 'ms'
7 EXCLUDE/gems/activesupport-2.3.10/lib/active_support/
      core_ext/benchmark.rb:17:in 'ms'
8 EXCLUDE/gems/actionpack-2.3.10/lib/action_controller/
      benchmarking.rb:68:in 'perform_action_without_rescue'
9 EXCLUDE/gems/actionpack-2.3.10/lib/action_controller/rescue
      .rb:160:in 'perform_action_without_flash'
```

Listing 5.6: Processed backtrace from 5.5

5.3.2 Suggesting fixes

The server is queried for suggestions for every failing test.

When the HelpMeOut client notices a failing test, it generates a query consisting of the line of code where the error is assumed to originate, the class name and message of the exception and the backtrace, processed as described above. The assumed error location is generated from the first line in the backtrace that refers to a file in the projects directory.

It then sends this query to the server and retrieves related fixes. These are rendered in an html file that is presented to the user after all tests are completed.

5.4 Server

The servers purpose is to store fixes and match them to incoming queries from clients. It is build as a Ruby on Rails application.

5.4.1 Finding relevant fixes

Query fixes by exception class and message, backtrace and line of code

A query to the server for relevant fixes consists of the exceptions classname and message, the backtrace and the line of code where the errors origin is assumed. When the server retrieves such a query, it has to find relevant fixes in its database. This is done in two steps:

First, we retrieve fixes by exact matches of the processed backtrace and exception classname from the database. As the processed backtrace only contains lines referring to installed libraries and the number of exception classes usually is not very high, this is a very liberal step and usually returns a lot of candidates.

Rate fixes and return five best

To decide which of the fixes from the first step to present to the user, we rate these and return the five highest rated ones to the client. Rating is based on a weighted sum of normalized Levenshtein string distance of the exception messages and the given line of code to the source code of the broken files in the fix. The source code rating of one fix is the highest rating of any line in any one of the files that belongs to this fix. Some tests showed, that a weight of 2 for the exception message and a weight of 1 for the source code gives good results.

Definition:
Levenshtein distance

LEVENSHTEIN DISTANCE:
The Levenshtein distance between two strings is defined as the minimum number of insertions, deletions or substitutions of a single character needed to transform one string into the other one.
Normalization is done by dividing this number through the number of characters of the longer string, resulting in a number between 0 and 1.

Tokenization of source code

Identifiers are replaced for comparison in the source code.

Because different programmers will use different identifiers, we replace these in both the code line from the query and the source code in our database before computing the ratings. We replace all custom variable, class and method names as well as number, string and symbol values. To match on the usage of Ruby or Rails specific methods or classes, we extracted method and class names from the Ruby and Ruby on Rails documentation and check if the current identifier is in this list before replacing it. This list currently consists of 711 class and 2520 method names.

This generalization of the source code is done by a custom lexical analyzer. Table 5.1 shows the name of the tokens we replace and the string we replace them with.

Token	Replacement string	Example
VarNameToken	INST_VAR / CONST / VAR	@foo → INST_VAR Foo → CONST foo → VAR
MethNameToken	METHOD	def hello → def METHOD
StringToken	STRING	“hello” → STRING
SymbolToken	SYMBOL	:hello → SYMBOL
NumberToken	NUMBER	42 → NUMBER

Table 5.1: Tokens replaced by the lexical analyzer

```
class Post <
    ActiveRecord::Base

  def pluralize_title
    @title = @title.pluralize
  end
end
```

```
class CONST <
    ActiveRecord::Base

  def METHOD
    INST_VAR = INST_VAR.pluralize
  end
end
```

Table 5.2: Source code before and after transformation by the lexical analyzer

Table 5.2 gives an example of source code before and after transformation. The class name `Post` is replaced by a general token `CONST`, the method `pluralize_titles` name is replaced with the token `METHOD` and the instance variable `@title` is replaced by `INST_VAR`. The identifiers

`ActiveRecord::Base` and `pluralize`, that both are part of the Ruby on Rails framework, stay part of the transformed source code.

Once the rating of the matching fixes is computed, XML data is generated from the 5 highest rated ones and sent to the client.

Chapter 6

Evaluation

To evaluate the utility of our prototype, we had 8 users perform a development task with HelpMeOut.

6.1 Questions

We concentrated our observations on the amount and utility of the presented suggestions as well as the number of newly generated fixes. We were also interested in general feedback subjects could provide about our tool.

A comparative study to evaluate the possible increase in productivity and software quality HelpMeOut could provide remains future work.

6.2 Participants

Participants had varying programming experience, but knowledge of RSpec.

Participants were found through the mailing lists of a Ruby on Rails class at Berkeley and the East Bay Ruby User Group. We also contacted the Ruby consultancy from Germany again, resulting in 3 more subjects, two of which already took part in our contextual inquiries. Like in our initial design interviews, the subjects experience with Ruby

on Rails varied. Some just finished their first introductory programming class while others had worked professionally for several years on large software projects. By the nature of our tool however, subjects were required to at least have some knowledge of the RSpec testing framework.

6.3 Method

To answer above questions, we wanted our subjects to program using the HelpMeOut tool. We would then watch them and take notes about the provided bug fix suggestions and receive general feedback about the tool.

To not have our subjects spend a lot of time thinking about what to work on, we had to design a task for them, which yielded some problems. The task description had to be fine-grained enough so developers know what to do but open enough to allow realistic results. An initial pilot test showed, that too specific task descriptions lead to fixes that are directly applicable to other subjects problems, which would seldom happen outside of our study and thus lead to invalid results.

6.4 Pilot tests

Evaluation set-up tested in pilot tests

Before the actual tests, we pilot tested with two more subjects to ensure that our evaluation set-up will provide useful results.

6.4.1 First run

Very specific tasks

The task for our first pilot tests participant was very specific. We set up a Ruby on Rails "Quiz" application consisting of a *Question* and an *Answer* model. The subject then was given very specific instructions on what he should do like
"Write the method Answer#correct?. It should return true if the

given answer matches the questions solution."
or
"Adjust Answer#correct? to be case-insensitive and ignore leading and trailing white-spaces in the given answer as well as in the questions solution.".

To be able to present some suggestions right from the start, we did the tasks twice before the participant and seeded the database with some fixes.

Too specific tasks not suitable for multiple subjects

We quickly realized that this approach is too specific and will not scale to multiple subjects. Even with only our initial data seeds, the suggestions for the first problem our participant ran into contained the complete solution, often even for future tasks. When the first test for the `correct?` method failed, our suggestions contained the fix in Figure 6.1, for example, which would have made this test pass, but is also the solution for the last task involving this method. This way all he had to do to solve his task was copy and paste the suggestion to the correct file.

Fix 3

app/models/answer.rb

```
@@ -5,7 +5,9 @@


   def correct?
-    text.upcase.strip == question.solution.upcase.strip
+    question.solution.split(',').any? do |solution|
+      text.upcase.strip == solution.upcase.strip
+    end
   end
 end
```

Figure 6.1: Suggestion containing solution to future tasks

Subject regarded suggestions as changes to his own code

Another problem this test revealed was the perception of suggested fixes not as examples of what others did, but as exact changes that need to be done to the users own code. The closeness of the examples we provided might also be responsible for that, but we also changed the wording of the suggestion interface to talk about *Suggestions* instead of

Fixes and explained what our tool did more in detail before subjects started programming for our next tests.

We then realized that only a more open task would be suitable to evaluate the utility of our tool. The most realistic approach would be to let developers work on whatever they want. This, however, is problematic, as many participants would probably not immediately have an idea of a web application they could develop when we invite them to our test. Extremely spontaneous trial and error programming also seems hard to do in a test-driven way.

6.4.2 Second Run: Blog application

More open task: creation of a blog application

For the second pilot test, we let the subject create a blog application. It should allow creating, listing and showing blog posts. Most Rails developers should already be familiar with that task, as it is a very common theme of learning tutorials. A video tutorial labeled "Creating a weblog in 15 minutes" was featured on the official Ruby on Rails website (www.rubyonrails.org[1]) for a long time.

This is a more open task, as functionality can be implemented in many different ways and there is no given order of the steps involved. For the subjects less experienced with test-driven development however, we provided a list of tests that should exists when they are done (see A— "Description of the evaluation programming task").

No use of code generation, strict test-driven development

We also instructed our participants not to use any code generators except for database migrations. All code had to be written by hand. Another requirement was to strictly develop test-driven. No implementation code should be written unless there was a failing test. Otherwise our tool could not show its full potential. Failing tests are required to as well collect as suggest fixes.

Before this pilot test we again worked through the tasks ourselves and generated some fixes. Even after this phase it was visible that there are multiple ways to solve the task.

[1] http://www.rubyonrails.org

This time, after the pilot test, our set-up seemed more promising. The subject understood the interface and realized that the suggestions were examples of others solutions. The task was open enough for suggestions to only provide hints and not complete solutions.

Suggestions did not provide complete solutions

6.5 Results

During the study, we recorded the screen and the users voice. We then counted the number of useful suggestions and noted suggestions and comments a participant might have about our tool.

We considered a suggestion useful, if its application would lead to the affected test case passing. As suggestions were only examples of other peoples code, the changes they presented had to be mapped to the current users code. Because of that, we considered suggestions useful, if they present the next *abstract* step to take. If the current problem is a missing method `edit`, we also considered a suggestion showing the definition of a method with a different name as useful.

After 8 hours of programming, our subjects generated 161 fixes. Of the 211 times they ran into problems, HelpMeOut suggested useful fixes in 120 cases (57%). 38 times (18%) suggestions were of no help and for 30 problems (14%) there were no fixes suggested. 23 times (11%) the users tests contained errors that prevented execution to reach the point where HelpMeOut gets loaded.

161 fixes were generated and 57% of suggestions were useful.

Between the three subjects working for the same company, HelpMeOuts suggestions were useful in 63% of the cases versus 53% for the rest of the subjects. This difference can be explained by the common style of programming companies often try to establish between their employees, but the sample of course is much too small to ensure significance.

More suggestions were useful if subjects followed a similar programming style.

After the small modification suggested by our pilot testers, people generally liked the HelpMeOut interface. Only little improvements were requested like still showing the re-

Only small improvements were suggested.

sults of the last test runs or presenting positive test runs in green. One subject suggested showing more than five suggestions per failing test and others did not like being unable to copy and paste from our interface due to the + and - line prefixes from the diff output. Copy and paste being hard might however not be too bad, because implementing the examples we show by hand encourages developers to think about them, which the HelpMeOut model requires. Not all solutions we suggest do really work when simply copied.

The three subjects working for the programming company noted they could imagine using a tool like HelpMeOut to train new employees to follow their programming conventions and style. If the new employees would see related examples prior to implementing functionality, this could not only help them to solve errors, but also remind them of company conventions like proper formatting or naming of variables and methods.

6.6 Discussion

While our study did not test the effect of the suggestions on programmer's productivity and software quality, we demonstrated that the concept of the original HelpMeOut tool can be applied to dynamic languages with similar results.

For the collection of bug fixes we added the requirement of using test-driven development, which might be an additional hurdle for programming beginners in contrast to the original HelpMeOut. Because of the large adoption of software testing amongst Ruby developers, we do not consider this to be a really big problem, however.

The difference in the rate of successful suggestions between the subjects with the same employer and the others suggests, that following common programming guidelines might improve HelpMeOuts utility. This effect could also be true for the other direction, as subjects imagined using HelpMeOut to train new coworkers to follow their guidelines by showing related examples during programming.

One remaining problem is that users might not want their code to be publicly available. As they currently have no control about what parts of their source code get submitted as a fix, it might contain private data and even passwords. If deployed inside one company, this might not be a problem at all, but future work could also implement mechanisms to filter certain parts of the source code before transmission.

Chapter 7

Summary and future work

7.1 Summary and contributions

Fixing bugs is a major part of software development. Starting with the first implementation steps, where developers often try different variations of possible solutions to simply see whether they will work, and certainly not ending with the release of the software to customers, resolving errors is an always present task.

Especially in dynamic languages, where there is no compiler and no type system to ensure certain constraints are satisfied and eliminate classes of errors, software testing is vital for a projects quality. In the Ruby community, test-driven development is very popular, resulting in very large test suites for many projects – often consisting of more code than the actual implementation.

This thesis presents an approach to automatically collect and suggest fixes for broken unit tests. A tool monitors the tests execution and queries a central server for fixes, if tests fail. If these tests later successfully pass, the differences of the affected files are sent to the server to become a new possible suggestion. Because of mentioned popularity of test-driven development in the Ruby community, this approach

gives us a very good handle on problems developers encounter.

To evaluate the idea we implemented a tool for the RSpec testing framework for Ruby programs. A user study of 8 hours of programming showed good potential for as well collecting as suggesting fixes. During this time, 161 new fixes were collected and 57% of the suggestions were useful.

7.2 Future work

7.2.1 Detection of duplicate fixes

Multiple similar fixes are presented for one failing test.

One problem with the current implementation is that there is no detection of duplicate fixes. When we select the fixes to be presented to the user, we simply take the 5 highest rated ones. The more our database is filled, these highest rated fixes are likely to be very similar though. There currently is no detection of whether one of our suggested fixes was applied, resulting in a new submission of the same fix the next time the affected test is run and passes successfully.

Implementations circumventing this duplication in the suggested fixes have to apply some way to detect them first. This could be achieved by either asking the user to state which fix he will apply in the HelpMeOut interface and then not submitting it again or by some similarity detection at the server. Both of these strategies have problems. Asking the user in the interface presumes that he knows whether a fix will work before it is applied and similar fixes at the server are also likely to not be exactly the same. Here again some heuristical way has to be evaluated in future work.

7.2.2 Suggesting fixes outside of tests

Use testing framework for collection, suggest fixes whenever exceptions occur

While we think it is a good way to limit the collection of fixes to a testing environment, suggestions could also happen outside of it. This would also allow users that do not

use RSpec to benefit from our tool, while for the collection of fixes we would still have the very clear indication of whether a bug is resolved and a test passes. Most people doing their first steps in Ruby probably do not use a testing framework and would probably be happy to see some suggestions when they run into exceptions.

An example implementation of this could be done by surrounding the request processing in the Ruby on Rails framework with a HelpMeOut exception handler. Instead of showing an error page in the browser in the case of an exception, an interface suggesting possible fixes could be presented.

7.2.3 Improve matching and rating of fixes

There is still a lot of room to improve fetching of fixes for a given error. Possibilities include improvements of the current approach by changing the text-based comparison of source codes to a comparison based on the programs structure, like a similarity matching of the generated syntax trees.

Other possibilities may include new features like allowing users to rate fixes. This way, we would not only crowd source the fixes itself, but also their relevance for specific problem scenarios.

7.2.4 Usability improvements

As some subjects noted, an option to allow copy and pasting or even automatic application of presented fixes might be helpful. As suggestions are most often not directly applicable to the users code and more a source of inspiration, more research has to be done on how to improve the way users can adopt the suggested fixes.

Another way to improve the users perception of the suggestions as helpful information and not code changes they need to apply might be to enrich their presentation with

more textual elements, moving the focus a bit away from the source code. These textual information could be user provided explanations of the suggested fixes. Future work could also prove a combination of HelpMeOut with the Blueprint concept (see 3.3.1—"Blueprint") by integrating results from web search into HelpMeOuts suggestion interface useful.

Appendix A

Description of the evaluation programming task

Create a blog application. Strictly follow the Test-Driven Development principles: Only write implementation code if there is a failing test. Do not use any rails generators except for migrations.

At least the following tests should exist:

Post:
- A post without a title is invalid
- A post without a body is invalid

PostsController:
- A get request to the index action renders the index view
- A get request to the index action assigns all posts as @posts

- A get request to the edit action renders the edit view
- A get request to the edit action assigns the requested post as @post

- A get request to the new action renders the new template
- A get request to the new action assigns a new post as @post

- A post request to the create action renders the new template, if creation was not successful
- A post request to the create action assigns the post as @post, if creation was not successful
- A post request to the create action redirects to the new post, if creation was successful

- A put request to the update action redirects to the post, if it was updated successfully
- A put request renders the edit action, if the update was not successful
- A put request assigns the post as @post, if the update was not successful

- A get request to the show action renders the show template
- A get request to the show action assigns the requested post as @post

Figure A.1: Description of the evaluation task

Bibliography

B. Ashok, Joseph Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa, and Vipindeep Vangala. Debugadvisor: a recommender system for debugging. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 373–382, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-001-2. doi: http://doi.acm.org/10.1145/1595696.1595766.

Hugh Beyer and Karen Holtzblatt. *Contextual Design: Defining Customer-Centered Systems (Interactive Technologies)*. Morgan Kaufmann, 1st edition, September 1997. ISBN 1558604111.

Hugh R. Beyer and Karen Holtzblatt. Apprenticing with the customer. *Commun. ACM*, 38(5):45–52, 1995. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/203356.203365.

Pierre Bourque and Robert Dupuis, editors. *Guide to the Software Engineering Body of Knowledge: 2004 Version – SWEBOK*. IEEE Computer Society Press, Los Alamitos, CA, 2005. ISBN 0-7695-2330-7. URL `http://www2.computer.org/portal/web/swebok/2004guide`.

Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 513–522, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-929-9. doi: http://doi.acm.org/10.1145/

1753326.1753402. URL http://doi.acm.org/10.1145/1753326.1753402.

Andrew Church, Lawrence Tratt, Roel Wuryts, Berend de Boer, Hong-Lok Li, George Brooke, James E. Hewson, and David T. Britt. Dynamically typed languages. *Software, IEEE*, 25(2):7 –10, 2008. ISSN 0740-7459. doi: 10.1109/MS.2008.35.

Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. ReAssert: Suggesting repairs for broken unit tests. pages 433–444, November 2009. http://mir.cs.illinois.edu/reassert/.

Chetan Desai, David S. Janzen, and John Clements. Implications of integrating test-driven development into cs1/cs2 curricula. In *SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education*, pages 148–152, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-183-5. doi: http://doi.acm.org/10.1145/1508865.1508921.

Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*, pages 1019–1028, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-929-9. doi: http://doi.acm.org/10.1145/1753326.1753478.

Haifeng He and Neelam Gupta. Automated debugging using path-based weakest preconditions. In Michel Wermelinger and Tiziana Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering*, volume 2984 of *Lecture Notes in Computer Science*, pages 267–280. Springer Berlin / Heidelberg, 2004. URL http://dx.doi.org/10.1007/978-3-540-24721-0_20. 10.1007/978-3-540-24721-0_20.

D. Jeffrey, Min Feng, N. Gupta, and R. Gupta. Bugfix: A learning-based tool to assist developers in fixing bugs. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 70 –79, May 2009. doi: 10.1109/ICPC.2009.5090029.

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

R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander. Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18:67–92, June 2008. doi: 10.1080/08993400802114581.

Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004. ISBN 0735619670.

Christian Murphy, Eunhee Kim, Gail Kaiser, and Adam Cannon. Backstop: a tool for debugging runtime errors. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 173–177, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-799-5. doi: http://doi.acm.org/10.1145/1352135.1352193.

Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. ISBN 0471469122.

Jakob Nielsen. Participation inequality: Encouraging more users to contribute, November 2010. URL `http://www.useit.com/alertbox/participation_inequality.html`.

Linda Dailey Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2):12 –15, 2007. ISSN 0018-9162. doi: 10.1109/MC.2007.53.

Stephen Schaub. Teaching cs1 with web applications and test-driven development. *SIGCSE Bull.*, 41(2):113–117, 2009. ISSN 0097-8418. doi: http://doi.acm.org/10.1145/1595453.1595487.

Jaime Spacco and William Pugh. Helping students appreciate test-driven development (tdd). In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*,

pages 907–913, New York, NY, USA, 2006. ACM. ISBN 1-59593-491-X. doi: http://doi.acm.org/10.1145/1176617.1176743.

G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.

Laurence Tratt and Roel Wuyts. Guest editors' introduction: Dynamically typed languages. *Software, IEEE*, 24 (5):28 –30, sept.-oct. 2007. ISSN 0740-7459. doi: 10.1109/MS.2007.140.

Davor Čubranić and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X. URL `http://portal.acm.org/citation.cfm?id=776816.776866`.

Peter Warren. Learning to program: spreadsheets, scripting and hci. In *Proceedings of the sixth conference on Australasian computing education - Volume 30*, ACE '04, pages 327–333, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc. URL `http://portal.acm.org/citation.cfm?id=979968.980012`.

Index

Typeset January 13, 2011