An empirical investigation into the usage of a live debugger

Master Thesis

Roger Stebler
from
Balsthal, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

31. Mai 2017

Prof. Dr. Oscar Nierstrasz
Dr. Andrei Chiș

Software Composition Group
Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland

Abstract

Reasoning about the run-time behavior of software applications is a challenging task. Live debuggers are essential tools that developers use in combination with other tools from an IDE to find and fix problematic behavior. Nevertheless, if developers use the debugger the wrong way or the debugger does not provide adequate features, it will complicate the debugging process. To explore these issues and help improve the debugging process, we perform an empirical investigation into how developers fix a given bug in an unknown piece of code. We focus our investigation on how developers use the debugger, and on how domain-specific information helps developers during the debugging process. Towards this goal, we design the experiment as a between-group study with 10 participants.

We collected and analyzed 6 hours of recordings. By analyzing them we observed that: *(i)* developers use different strategies to find the cause of a bug; those who successfully solved the given bug observed the live behavior of the application while stepping in the debugger; *(ii)* domain-specific information helps developers if they are able to find it, especially when dealing with unfamiliar libraries, and *(iii)* finding and using domain-specific information is not always straightforward unless the information is shown by default. Based on these observations we propose several improvements to the debugger, such as visually highlighting domain-specific information in the debugger, and automatically executing previous debugging actions to avoid their repetition.

Contents

1 Introduction

In software development, integrated software environments (IDEs) help programmers to create and maintain software.

1.1 Debugging software

Software developers spend a large part of their time debugging software. This process involves various activities like: reproducing bugs, finding and understanding their causes, and finally repairing them. Current IDEs provide multiple tools and features that support developers in performing these tasks. One example is static analysis tools. They can analyze the program without executing it and detect a wide range of issues, like type mismatches, arithmetic overflows, null exceptions, *etc.* A different category of tools are those that actually allow developers to interact with a running system, instead of just looking at the static code. One crucial development tool from this category that is nowadays present in most IDEs is the live debugger.

The live debugger[1] supports developers in understanding the run-time behavior of a running application. It does this by allowing developers to control the execution of the program. First, developers can add breakpoints to pause the program execution when an interesting point is reached. Second, developers can inspect and change the state of the interrupted program through embedded inspectors. Third, developers can step through the execution of the program, one or more instructions at a time. In some languages, live debuggers also enable developers to edit the code of a running method, save it and resume the execution, without stopping the program.

An alternative to a live debugger is a post-mortem debugger. This debugging technique allows the developer to debug a process that already finished executing or has crashed by analyzing a memory dump file that was created during run time of the process.

[1] Unless otherwise mentioned, when we use the word debugger to refer to a live debugger.

1.2 Research Questions

Given that the debugger is a crucial, but also complex tool, in this thesis we focus on investigating ways in which we can improve it. To achieve this, we design and perform an empirical study with software developers. The experiment consists in fixing a bug in an unknown piece of code. By focusing on a user study instead of a survey, we can directly see how developers interact with the debugger during a bug-fixing scenario. We can further observe how they use the debugger in combination with other tools from the IDE.

We selected to design the user study starting from the following three research questions:

RQ1: What debugging strategies are effective in solving the given bug?

RQ2: Does domain-specific information improve program comprehension during debugging?

RQ3: How do participants notice and use domain-specific information during debugging?

There are many possible ways of approaching a debugging task, depending on the actual bug, the developer's experience and the available tools. Through *RQ1* we want to investigate strategies taken by developers who were successful in solving the given bug. In particular we are interested in how developers navigate through code, what tools they use, and for what tasks they rely on the debugger.

It is a common statement that domain-specific information helps developers during debugging [2, 3, 9]. For example, debugging a parser using breakpoints at the level of the parser's grammar should be better then debugging the same parser using only breakpoints at the level of the programming language. Through *RQ2* we want to investigate this aspect, as well as what domain-specific information can help developers solve the given bug.

Last but not least, a developer can only take advantage of domain-specific information or domain-specific actions in the debugger if she is able to find and understand them. Through *RQ3* we are interested in observing how developers notice and discover such information.

As a concrete debugger for answering these research questions we selected the debugger from the Pharo IDE[2]. We made this choice as Pharo comes with a debugging framework where it is easy to add domain-specific information to the debugger [2], as well as to create domain-specific extensions. The Pharo IDE also comes with a generic debugger, and two custom extensions, one for debugging bytecode and one for debugging failed tests.

1.3 Experiment Setup and Main Findings

Gaining insight into the three research questions requires first and foremost a relevant bug. To select it we surveyed the bug tracker for the Pharo IDE and discussed with Pharo developers. We detail this process in Section 3.2. Second, to be able to answer our research questions we selected a between-group design. We designed two experimental setups and assigned participants to one of the two groups at random. In the first setup, participants had to use the standard debugger from Pharo. For the second group, we extended the debugger and object-inspector with several pieces of domain-specific information, relevant for the domain of the selected application.

We performed the experiment with 10 participants (PhD students and software developers); we assigned 5 participants to each group. During the experiments we collected information about the usage of the debugger and the interaction with other tools in the IDE using two recorders

[2] `http://pharo.org`

for the Pharo IDE. We also recorded the screen and asked participants to think aloud during the task.

To answer the three research questions we used a qualitative analysis with additional quantitative support. We first transcribed the interviews and, for each research question, extracted relevant information. We also analyzed the recorded data to infer navigation between tools, tool usage, and time spent in various methods. We can make the following observations about each research question:

RQ1: Only two participants solved the bug. We observed that they used a strategy in which they based their decisions on observing the live behavior of the application containing the bug. Several other participants relied on dynamic data, however, by not observing the behavior of the application they were unable to find the bug. A participant did not rely on dynamic data and made very little progress;

RQ2: We noticed that indeed domain-specific information improved several subtasks, as long as participants were able to find the relevant information in the debugger. Domain-specific information was especially useful for participants not familiar with the libraries used in the selected application; developers who encountered those libraries before already knew what to look for.

RQ3: We observed that participants could not really find relevant extensions unless they were active by default. Even if participants were told where to find them, during the task they focused mainly on tool features that they already knew and felt comfortable with.

Based on this analysis we propose several improvement opportunities in the debugger. One improvement opportunity is to emphasize in a visual way domain-specific information and actions in the debugger, the first time a developer encounters them. This could ensure that developers are made aware that they exist. We show a possible approach for highlighting the domain-specific extensions used by the second group in Section 5.1.

Another improvement opportunity discussed in Section 5.2, consists in embedding the visual representation of a graphical component directly in the debugger, when dealing with graphical components. This change is directly related to the domain of the selected application, and in our experiment this would have allowed participants to better find the code containing the bug.

We further noticed a significant number of repetitive actions in the debugger for navigating through the execution of a program. For example, developers started a debugger, performed several actions like *step into* and *step over* to reach a certain point, closed the debugger in order to make a change, and then redid the navigation in the debugger to get back to the previous point. To address this we propose in Section 5.4 a solution that records the actions done in the debugger, allowing a developer to restore previous navigations through the execution. An alternative solution consists in inserting conditional breakpoints that would stop in the same place. We chose to record and replay debugging actions as for core libraries of a system inserting conditional breakpoints can slow down the execution.

Outline

The remainder of this thesis is structured as follows:

Chapter 2 discusses several related studies that analyze developer actions within IDEs.

Chapter 3 describes the setup of the experiment. It presents the task that the participants were asked to solve, and the differences between the tools used by the two groups.

Chapter 4 presents the participants' various debugging strategies and analyzes usage data for several tools from the Pharo IDE. It also describes the participants' difficulties and discusses the three research questions.

Chapter 5 provides different opportunities to address problems found during the experiment and presents possible implementations.

Chapter 6 summarizes and concludes the thesis.

2 Related Work

In this chapter we look at several other studies that aim to improve software development by analyzing how developers interact with an IDE. We look at studies involving both the Pharo IDE, as well as other languages and IDEs.

2.1 Understanding Developer Questions

IDEs are not only used for writing new code. They are also used to navigate through the code, understand it and build a mental model. In their study, Kersten and Murphy [4] claim that programmers spend more time navigating through code than working with it. Kersten and Gail developed a plugin called Mylar (also known as Mylyn) for Eclipse that monitors the activity of the software developer. Based on the collected data the plugin then generates a degree-of-interest (DOI) model for each program element based on the historical selection or modification of the element. Every time an element is selected or modified its DOI is increased and decreases over time. The element is then highlighted depending on its DOI. This helps the developer to find the elements that are relevant for the task she is currently working on. Kersten and Murphy tested their tool in an experiment. They informed their participants about Mylar and collected information about the usage of the tool. This data was used to calculate an "edit ratio" – the number of keystrokes in the editor over the number of selections made in the IDE. Their results show that the edit ratio increased between their baseline usage data and the Mylar usage data on average by 15%. However, Mylar is not optimized to support debugging activities and can lead to overpopulation *e.g.,* single-stepping through the code results in too many irrelevant elements to be marked as interesting. Additionally some participants disliked the intensity of the colors added to the views.

Not only the direct interaction with the IDE is important. Another useful approach to improve the IDE is to know what the developers think during their development task. Sillito *et al.* identified four categories of questions being asked during a programming session in statically typed languages [14]. These categories are:

- Finding initial focus points – *e.g.,* "Which type represents this domain concept?"

- Expanding focus points – *e.g.,* "Which type is this type part of?"
- Understanding a subgraph – *e.g.,* "What is the behavior these types provide together?"
- Questions over groups of subgraphs – *e.g.,* "What is the mapping between these UI types and model types?"

Perscheid *et al.* published a field study [12] about the advancement in debugging practice of professional software developers. They observed eight software engineers while debugging and interviewed them about their experiences. Additionally they did an online survey to get more data. Their results show that the most difficult bugs nowadays are related to erroneous program design and parallel behavior. They also observed that different tools and debugging strategies are useful for different types of bugs. An important skill for efficient debugging is to be able to decide which tool to use for what kind of bug. For their participants the most important properties for new debugging tools are useful features, the ease of use and availability of documentation.

In a study of expert Java programmers using Eclipse [5], Ko *et al.* identified three fundamental activities performed by developers during maintenance: (1) collecting a *working set* – a group of task-relevant code fragments, (2) navigating dependencies between code fragments (such as callers, callees, declarations) and (3) repairing or creating the necessary code. They suggest six design requirements for maintenance-oriented tools based on the findings in their study: (1) A working set interface for adding/removing task-relevant code fragments, (2) distinct tools for navigating and representing working sets, (3) automatically adding dependencies to the working set, (4) highlighting unchanged references of copied code, (5) helping to find indirect dependencies of copied code and (6) automatically building a working set for *why* and *how* questions about the program output. They claim that using these opportunities for new tools could save up to 35% of the programmer's time.

Three years later Ko *et al.* presented a tool called Whyline [6] that tries to implement the idea about automatically answering developer questions. The tool derives *why did* and *why didn't* questions from the program's code and execution about the object and its properties and fields. These questions are answered using the call graph, as well as static and dynamic slicing techniques. The authors claim that novice programmers using their tool were twice as fast as expert programmers without it.

In a paper presenting three studies [8] LaToza *et al.* showed that *reachability questions* – searches across feasible paths through a program for target statements matching search criteria – are common and consume a lot of the developers' time. In their first study they found that developers often understood facts about the code incorrectly and based on these false assumptions they implemented buggy changes. The authors could relate half of those defective changes to a reachability question. The goal of the second study was to understand the frequency and difficulty of reachability questions. Their results show that on average the developers asked 9 reachability questions every day; 82% of them were rated as hard to answer. The third study showed that 90% of the longest debugging sessions were associated with reachability questions. The authors conclude that developers would be able to perform coding tasks more quickly with tools that support them in answering reachability questions.

The study that we are proposing in this thesis builds on the ideas of these papers, however, proposes to have a closer look at how developers use the debugger during a bug-fixing task.

2.2 Studies involving the Pharo IDE

Minelli and Lanza proposed a tool called DFlow [10] that records and creates web-based visualizations of the interactions between developers and the Pharo IDE. Using this tool they collected

information about how developers spend their time during programming sessions, concluding that they spend around 70% of their time performing program comprehension tasks and 14% of their time in fiddling with the UI of the IDE. The time used to actually edit and navigate through code covers only about 5% respectively 4% of the time [11]. Their results show that a simple to understand and user-friendly IDE is important to improve the efficiency in software development.

Kubelka *et al.* replicated Sillito's thesis in Pharo, an environment for the dynamically typed language Smalltalk [7]. Additionally to Sillito's findings they added more questions to the four categories. On debugging sessions with an unfamiliar codebase they found similar results as Sillito *et al.* But on sessions with a familiar codebase the number of questions from the category *Expanding focus points* was higher and the number of questions from the category *Questions over groups of subgraphs* was lower than in Sillito's paper. Kubelka *et al.* assume that this is caused by different development strategies due to missing type information and test driven development. The authors also noticed various techniques to get static information in Pharo. To find the declaration or definition of a method the participants often used the *senders* (callers) and *implementors* (callees) tools. However, if the method name is commonly used this technique can be inconvenient and, if the object's type is known, navigating to the class and then to the implementation of the method would be more efficient. Kubelka *et al.* also noticed that some developers made code changes based on assumptions about the object's type or the method names that a particular class should understand, and only in the case of a failure did they investigate the code in more detail.

2.3 Summary

Studies show that the maintenance and debugging process is time consuming. Developers spend a lot of time for code comprehension and answering reachability questions, to find the code fragments that are related to the bug they are trying to resolve. For different types of problems there are different tools and debugging strategies that are most effective to find and solve this problem. Developers often ask *how* and *why* questions to get a better understanding about the code. These questions can vary depending on the type and the domain of the application. In this thesis we are interested in improving debugging by better understanding how developers use a specific tool, the debugger, and by investigating how domain information can help developers during a bug fix.

3
Study Design

In this chapter we describe the design and setup of the study. We also describe the selected bug and the best strategies for solving it.

3.1 The Pharo IDE

Before going into details about the experiment, we present a brief overview of Pharo, the environment that we are using for the experiment. Pharo is an open-source, Smalltalk development environment. As in any Smalltalk implementation there is a very close integration between the language and the IDE. The Pharo IDE itself consists of multiple development tools (Figure 3.1). A central tool is the code browser *Nautilus* that allows developers to navigate through code, create new classes and methods, and refactor code. Another important tool is the *debugger*. It is a live debugger that allows developers to go through the code step by step and inspect the state of the program and its variables. Two other tools for static code analysis are the *senders* (callers) and *implementors* (callees) browsers. The *playground* is a tool that provides an interface for scripting and live programming. It is often used by developers to work with snippets of code not yet belonging to a class. Last but not least, there is the *inspector* that allows developer to explore the state of run-time objects.

Unlike other IDEs, both the inspector and the debugger from Pharo are *moldable*: they allow developers to create new domain-specific extensions with low effort [1]. For example, there are inspector extensions for displaying widgets in a visual way and debugging extensions for parsers. We rely on this in our experiment, as for one group we introduce several extensions to the IDE.

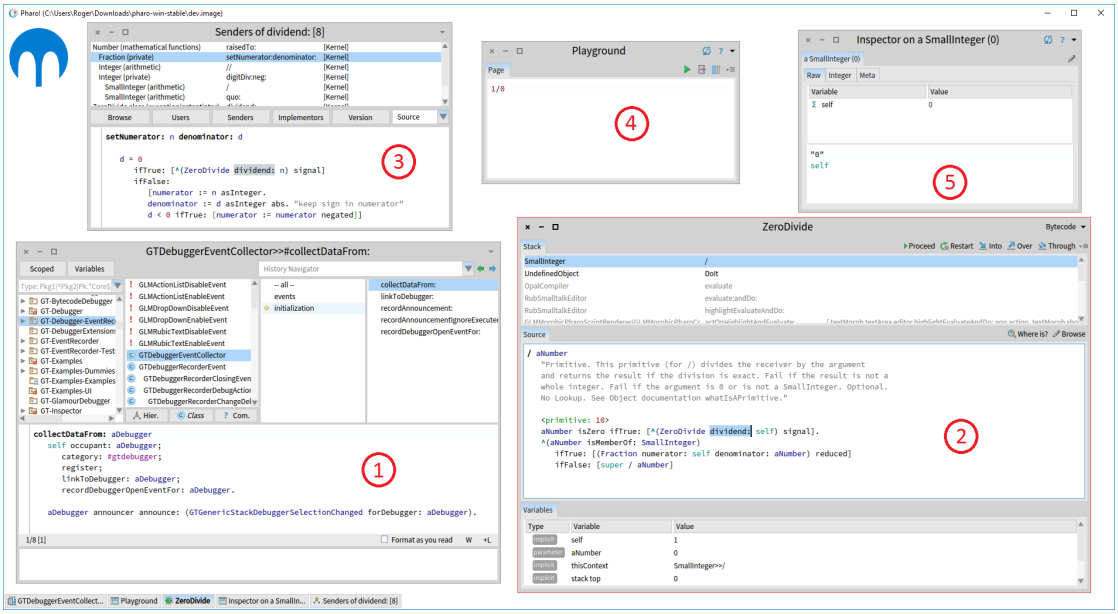

Figure 3.1: The main tools from the Pharo IDE: the code browser Nautilus (1), the debugger (2), the browsers for senders and implementors (3), the playground (4) and the inspector (5).

3.2 Task

Next we describe the selected bug in detail together with its fix.

3.2.1 Finding a suitable bug

Gaining insight into the three research questions mentioned in Chapter 1 requires first and foremost a relevant bug. Hence, we decided to look for an existing bug instead of creating a bug by ourself. This helps us to reason about how developers solve a real-world bug. Another requirement that we had for the bug was its difficulty. To get enough data about tool usage in Pharo we needed a bug that should not be found and fixed with ease.

To find a suitable bug in a Pharo application we surveyed FogBugz[1]. This bug tracker contains bugs related the Pharo IDE itself, as well as libraries and frameworks used by the IDE. We looked for bugs where the discussion indicated a difficult bug and that was not closed for several months. We also asked Pharo IDE developers for possible candidates. This search produced six potential candidates[2]. We then selected *"Case 15345: GT workspace/inspector evaluates source wrong when it ends with $", treating it as nil instead"* as the bug for the experiment. This bug was reported on April 16 2015 and was fixed in Pharo build 50662 on March 26 2016. The bug affects

[1] `https://pharo.fogbugz.com`

[2] `https://pharo.fogbugz.com/f/cases/7357`, `https://pharo.fogbugz.com/f/cases/13049`, `https://pharo.fogbugz.com/f/cases/13316`, `https://pharo.fogbugz.com/f/cases/15286`, `https://pharo.fogbugz.com/f/cases/15345`, `https://pharo.fogbugz.com/f/cases/17330`

Rubric, a text editor from Pharo used by several development tools and it covers our needs well, as it is not straightforward to locate the code containing the bug.

3.2.2 Understanding the problematic behavior

Like any code editor, the code editor from Pharo allows developers to simply select code and perform various contextual actions on it. For example, in the debugger and playground developers can evaluate, debug, or inspect the selected code. In case no code is selected when a contextual action is invoked (Figure 3.2) the code editor will automatically select the current line. However, due to the presence of the selected bug, if the line ends with a code comment there will be an error during the invocation of the action (Figure 3.3).

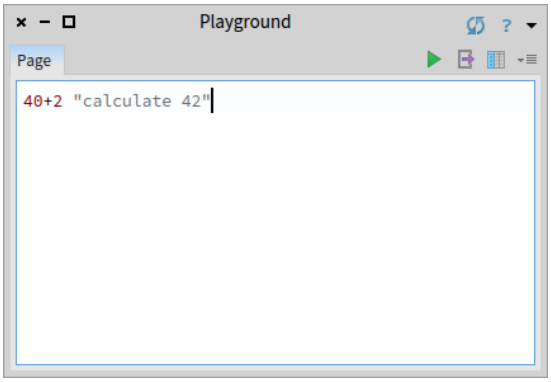

Figure 3.2: Pharo Playground with a snippet of code that triggers the bug.

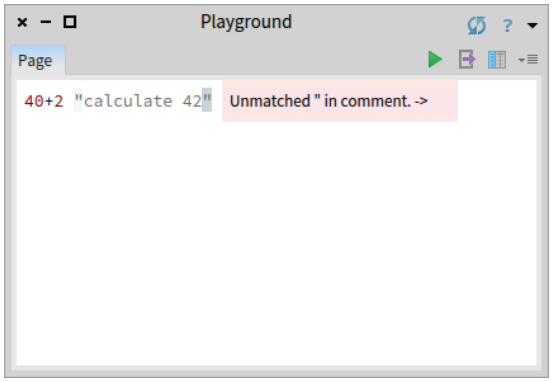

Figure 3.3: Error message after executing the line in case no explicit selection was already made.

This bug appears due to a feature of Rubric that allows developers to execute a line of code that is commented out. When a developer comments out a line of code[3] and later still wants to execute this line, for example to inspect the returned value, she does not need to remove the quotes denoting the comment or select the code inside the comment manually. She can simply place the cursor somewhere on the line and perform the desired action. The editor then selects the code by removing the double quotes at the beginning and at the end of the line, before passing it to the compiler. If the Playground contains on a line the code (*e.g.,*`"40+2"`), executing this code without any selection will select and execute `40+2`.

The method `#computeSelectionIntervalForCurrentLine` from the class `RubTextEditor` contains this behavior. Rubric uses this method to compute the selection of the current line. Nonetheless, the bug that leads to the error described above can be found in this method and is shown in Code 1.

On line 4 the interval for the complete line is calculated. In lines 7-8 the left side of the selection of the line is trimmed by removing quotes and separators *i.e.,* whitespaces, tabs, line breaks and similar characters. In lines 9-10 the same is done for the right side of the selected line.

What the creator of this method did not consider is the case when a developer does not comment out the whole line but just adds a comment at the end (or at the beginning) of the line. In this case the method above just removes the last quote of the comment and the rest of

[3]Pharo uses double quotes to define comments

```
1  computeSelectionIntervalForCurrentLine
2     | i left right s |
3     s := self string.
4     i := s encompassParagraph: self selectionInterval.
5     left := i first.
6     right := i last.
7     [ left <= right and: [ (s at: left) = $" or: [(s at: left) isSeparator] ] ]
8        whileTrue: [ left := left + 1 ].
9     [ left <= right and: [ (s at: right) = $"  or: [(s at: right) isSeparator] ] ]
10       whileTrue: [ right := right - 1 ].
11    ^ left to: right
```

Code 1: The faulty method that calculates the selection interval when no explicit selection is present.

the line is still selected. The same happens for comments at the beginning of the line where only the first quote is trimmed.

Consider a line having the code `40+2 "compute 42"`. The size of this string is 17 characters, so the variable `left` will have the value 1 at the beginning of the trim process and the variable `right` will have the value 17. On the left side of the code there are no quotes or separators. Thus in lines 7-8 nothing is trimmed and the variable `left` stays unchanged. On the right side however we have a comment. The last character is a quote so the variable `right` is decreased from 17 to 16 in lines 9-10. The next character is the number `2` which is neither a separator nor a quote. So nothing is trimmed anymore on the right side. The resulting selection of the line will be from character 1 to 16, *i.e.,* `42+2 "compute 42`, resulting in the parsing error `Unmatched " in comment` due to the missing quote character at the end.

3.2.3 Addressing the bug

The developers of Pharo fixed this bug by ignoring any quotes at the beginning and the end of the line. The solution integrated in Pharo also removes the original feature, that of allowing developers to execute lines that are completely commented out. Since the Pharo developers removed this feature we also accept this solution in our experiment. Participants are allowed to just remove the complete code that handles and removes the quote characters. After asking the developers who were involved in fixing this bug we found out that they were actually unaware of the previous behavior, never used it, and consider the fix to be correct.

3.3 Experiment Setup

To ensure that all participants started from the same buggy code, we prepared a set of unit tests reproducing the bug. One of them is presented in Code 2.

This test opens a playground, using Rubric as a text editor, and executes the line `40+2 "compute 42"` without selecting it first. Then it checks if the result was computed successfully. The test suite also included other tests to exclude possible buggy solutions that while fixing this bug were breaking other parts of the text selection logic in Rubric.

For the experiment we created two groups and we randomly assigned participants to one of them:

- *Group 1*: The participants in *Group 1* got a default Pharo image in which we reintroduced the bug described above. They were allowed to use any tool available in the Pharo IDE.

```
testExecutionWithComment
   | page obtainedResult |

   page := GTPlayPage new
      saveContent: '40+2  "compute 42"';
      yourself.

   window := playground openOn: page.

   playground codePresentation evaluateSelectionAndDo: [ :aResult |
      obtainedResult := aResult ].

   self assert: obtainedResult equals: 42.
```

Code 2: One of the tests used in our experiments

- *Group 2*: The participants in *Group 2* got a Pharo image in which we reintroduced the bug described above, and were also allowed to use any tool available in the Pharo IDE. For this group, however, we introduced in the IDE several domain-specific extensions for the debugger and the inspector.

Every participant got 50 minutes to find and solve the bug. However, we did not enforce this limit. We allowed participants to stop if they solved the bug earlier and also if they felt that they could not make any progress towards solving the bug. Also after 50 minutes we informed participants of the time, but if they wanted they could choose to spend more time on fixing the bug. We asked participants to think aloud as they were solving the bug and used screen recording during the experiment. Additionally we employed two tools to collect usage data during the task:

- DFlow[4] – also known as DevFlow – is a general profiling tool for the Pharo IDE. DFlow collects various information like user clicks in the IDE, selected code entities, used tools and code modifications.
- GT Debugger Event Recorder – To record more fine-grained data about the usage of the debugger we created a custom recorder. This recorder is based on GT Event Recorder from the Glamorous Toolkit, the framework on which the Pharo debugger is constructed. GT Debugger Event Recorder collects information about what debugging actions were executed in which method and which extensions were used. This data is then used to reconstruct the debugging session and analyze the usage of the debugger and its extensions.

3.3.1 Domain-specific extensions

The extensions available for participants in *Group 2* include two extensions for the debugger. First we added an extension for *SUnit*, the testing framework from Pharo, available while debugging unit tests. This extension shows the returned and the expected result of the failed assertion and the `setUp` and `tearDown` methods associated with a test. Second we introduced an extension for debugging *announcements* that is available when the stack frame contains an announcement. This extension shows all the subscribers of the current announcement and allows developers to directly jump to the method that handles the announcement. We decided to introduce the extensions related to announcement as they are often used in Rubric. Also to reach the method

[4] `http://dflow.inf.usi.ch/`

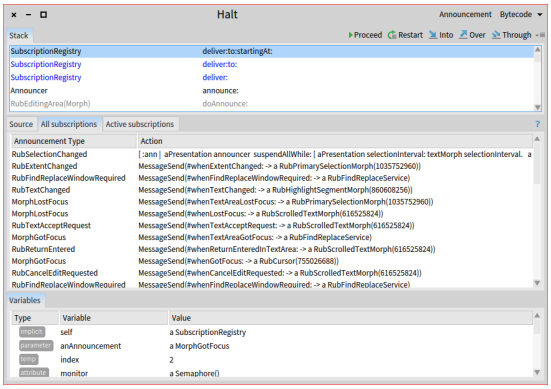

Figure 3.4: The tab "All subscriptions" shows all the subscriptions of the current announcer for *any* announcement.

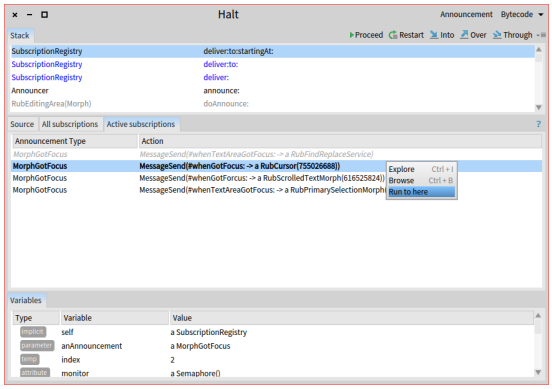

Figure 3.5: The tab "Active subscriptions" shows only the subscriptions that are interested in the announcement that is currently being delivered. It allows the user to directly jump to the code that handles the announcement.

containing the bug developers have to navigate through the propagation of an announcement, which is often considered a difficult task.

To solve this bug, developers also have to deal with `Stream` objects: after computing the selection interval, Rubric creates a stream object that gives access to the selected text and passes the stream to the compiler. A stream in Pharo holds a `String` object, and gives access to only a portion of this string controlled using a starting position and a read limit. When inspecting a stream object developers can see these three attributes, and need to know that the actual content of the stream depends on its starting position and read limit.

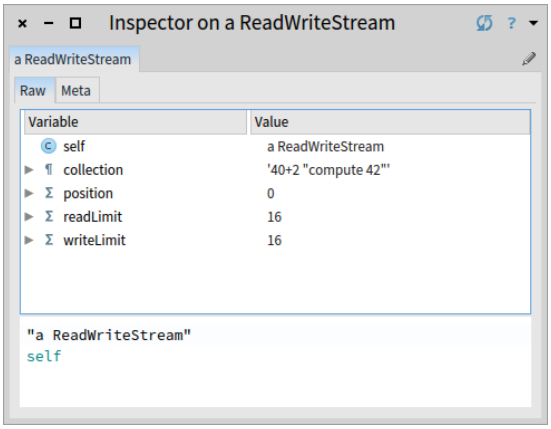

Figure 3.6: The *Raw* view of a stream in the inspector. It shows the complete stream including attributes like `position`, `readLimit` and `writeLimit`.

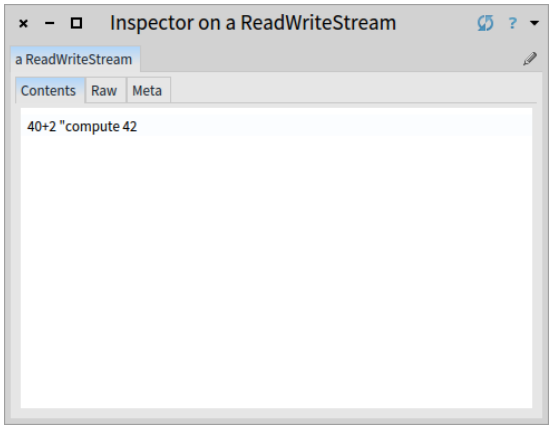

Figure 3.7: Our *Contents* extension additionally displays the actual content of the stream.

For the selected bug we noticed that it is difficult to observe that the stream does not give

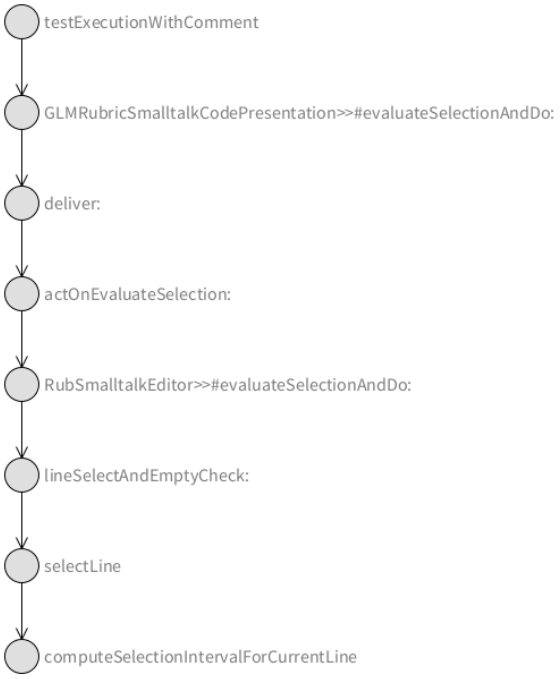

Figure 3.8: Navigation through the code from the test to the buggy method `computeSelectionIntervalForCurrentLine`

access to the entire string, as it is caused by a off-by-one error. Hence, for *Group 2* we added a custom view in the inspector (Figure 3.7) only showing the actual content of the stream.

When describing the experiment to the participants in *Group 2* we did not mention the extensions for `Stream` objects and announcements. We only described the SUnit extensions and made them aware that during the task they may encounter other extensions within the debugger and the inspector. We made this choice to not directly influence developer's activity, and to see what factors makes them decide to use other extensions.

3.3.2 Main navigation path

There are several navigation paths through the code that can lead to the source of the bug, *i.e.,* the method `computeSelectionIntervalForCurrentLine`. Figure 3.8 shows the simplest one. We will use it to analyze the navigation of participants through the code.

This path starts with a test method. After creating and opening the playground window the test adds the code `40+2 "compute 42"` to the playground's text editor and calls the method `GLMRubricSmalltalkCodeRepresenation>>#evaluateSelectionAndDo:`. In this method an announcement of type `GLMEvaluateSelection` is raised. This announcement is then handled by the text editor's renderer `GLMMorphicPharoCodeRenderer`. The renderer tells then the text editor to select and evaluate the code of the current line. The code is selected in the method `#lineSelectAndEmptyCheck:` of the class `RubTextEditor`. This method checks if the code was already selected manually. If there is no selection it will calculate the selection interval using the method `#computeSelectionIntervalForCurrentLine`.

3.4 Participants

We did the experiment with 7 PhD students from different universities and 3 software developers who work with Pharo. The general programming experience of the 10 participants ranges from 5 to 14 years and the programming experience with Pharo ranges from 0.75 to 8 years.

	Group	Industry	University	Pharo
P1	1	8	2	4
P2	1	2	3	3
P3	2	4	7	0.75
P4	2	10		5
P5	2	7	5	5
P6	1	5	4	6
P7	1	10		8
P8	1	8		4
P9	2	14		5
P10	2	11	2	4
Average G1		8.4		5
Average G2		12		3.95
Average		10.2		4.48

Table 3.1: Participant's group, their total programming experience (in years), and their programming experience with Pharo. For the total programming experience we distinguish between academic and industrial experience.

Additionally we asked participants about their experience with several libraries from Pharo directly relevant for this experiment: Glamour, the announcement framework and the Pharo compiler. We used a Likert scale with five possible answers: "Not at all", " Slightly knowledgeable", "Somewhat knowledgeable", "Moderately knowledgeable" and "Extremely knowledgeable". We observed that most participants had at least some knowledge of these libraries (Table 3.2).

	Glamour	Announcements	Compiler
P1	Not at all	Not at all	Not at all
P2	Moderately knowledgeable	Extremely knowledgeable	Somewhat knowledgeable
P3	Slightly knowledgeable	Slightly knowledgeable	Slightly knowledgeable
P4	Somewhat knowledgeable	Not at all	Moderately knowledgeable
P5	Moderately knowledgeable	Extremely knowledgeable	Moderately knowledgeable
P6	Slightly knowledgeable	Slightly knowledgeable	Extremely knowledgeable
P7	Not at all	Extremely knowledgeable	Somewhat knowledgeable
P8	Somewhat knowledgeable	Moderately knowledgeable	Extremely knowledgeable
P9	Slightly knowledgeable	Somewhat knowledgeable	Slightly knowledgeable
P10	Slightly knowledgeable	Somewhat knowledgeable	Slightly knowledgeable

Table 3.2: Participant's experience with different components in Pharo: Glamour, the announcement framework and the Pharo compiler.

Last but not least, we asked participants about their experience with other programming

languages and IDEs. All of them are familiar with Pharo. On average the participants have worked with four programming languages and three IDEs. The most used programming language is Java and the most used IDE was Eclipse. Table 3.3 summarizes these results.

	Programming Languages	IDEs
P1	C/C++ (4y) Java (5y)	Eclipse (6y) NetBeans (1y) Visual Studio (1y)
P2	Java (1y) Ruby (1y)	Eclipse (0.5y) IntelliJ (1.5y) RubyMine (1.5y)
P3	Java C Eiffel	Eclipse IntelliJ Pharo
P4	Java Pascal Delphi PHP	Eclipse Pharo
P5	Gupta (2y) Java (3y) Python (1y) C# (1y) Bash (7y) PHP (1y) Lisp (2y)	Gupta (2y) Eclipse (3y) Emacs (7y) VisualWorks (5y) Pharo (5y)
P6	Java (3y) C, Assembly (3y)	Eclipse (3y) Pharo (6y) Xcode (0.5y)
P7	Java (10y) C# (10y) Pascal (2y) Delphi (2y)	Eclipse (10y) Visual Studio (10y) Dolphin Smalltalk (1y) Visual Work (2y) Delphi (2y) Pharo
P8	Java PHP (2y) COBOL (3y) C# (2y)	Visual Studio Eclipse Pharo
P9	Java PHP Python	Eclipse (6y) Pharo
P10	VisualWorks Smalltalk (0.5y) Python (1y) Java (6y) C++ (5y) Javascript (6y)	Pharo (4y) Eclipse (5y) KDevelop (1y)

Table 3.3: Participant's experience with programming languages and IDEs.

4
Study Results

In this chapter we present the results of our experiment. We discuss how participants used the debugger, discuss their strategies, and look at how domain-specific information helped them during the task.

4.1 Data analysis

During the actual experiments we used DFlow to collect data about the participant's interaction with the IDE. DFlow however only records basic events in the IDE. To analyze these events we created the tool *DevFlow Analyzer*. This tool reconstructs the debugging session by grouping events together based on the tool that was focused at the time the event was generated.

We also used GT Event Recorder to collect information during experiments. This is an infrastructure for recording IDE interactions available in the Pharo IDE. For this experiment we extended it with a recorder capturing fine-grained interactions in the debugger. We then integrated this data into the debugging sessions obtained using *DevFlow Analyzer*.

4.2 Completion times and correctness

We observed sessions of different lengths, ranging from 20 minutes up to 53 minutes. On average participants spent 38 minutes on the task. We analyzed each session and gave it a correctness score (Table 4.1) using the following scores:

A. The participant found the cause of the bug and solved it correctly. She reached the method `computeSelectionIntervalForCurrentLine` and changed it so that all provided tests passed;

B. The participant discovered that there was a problem in computing the selection. She reached the method `evaluateSelectionAndDo:` that was accessing the selection but was unable to determine why the selection was wrong;

C. The participant successfully navigated the propagation of the `GLMEvaluateSelection` announcement, but made no other progress;

F. The participant made no real progress towards solving the bug and was unable to navigate the propagation of the `GLMEvaluateSelection` announcement and reach the method `actOnEvaluateSelection:`.

After the experiment we also let participants rate both how realistic and how difficult the given task was using a 5-point Likert scale. For indicating if the task was realistic the scale goes from 1: "Strongly disagree" to 5: "Strongly agree"; for indicating difficulty it goes from 1: "Very difficult" to 5: "Very easy". All participants found the task realistic. Six participants found the task difficult, with only two finding it easy. Even if participant *P2* solved the task correctly she did not find it easy.

	Is the task realistic?	Difficulty	Duration	Score
P1	agree	difficult	40:00	C
P2	strongly agree	neutral	23:20	A
P3	agree	difficult	40:54	F
P4	agree	difficult	45:18	B
P5	strongly agree	easy	30:36	A
P6	agree	easy	34:34	B
P7	agree	difficult	39:41	B
P8	strongly agree	difficult	48:30	B
P9	agree	neutral	25:17	B
P10	strongly agree	difficult	52:44	B

Table 4.1: Participant's rating of how realistic and difficult the bug is including the time spent on the task and the correctness scores for the debugging sessions.

4.3 Tool Usage

Apart from correctness and completeness, we further analyze the time spent by participants in different tools and the navigation between tools. We exclude from this analysis participant *P4*, as the participant broke the Pharo image several times without saving it; because of this we could not collect reliable usage data. In this analysis we include the tools introduced in Section 3.1.

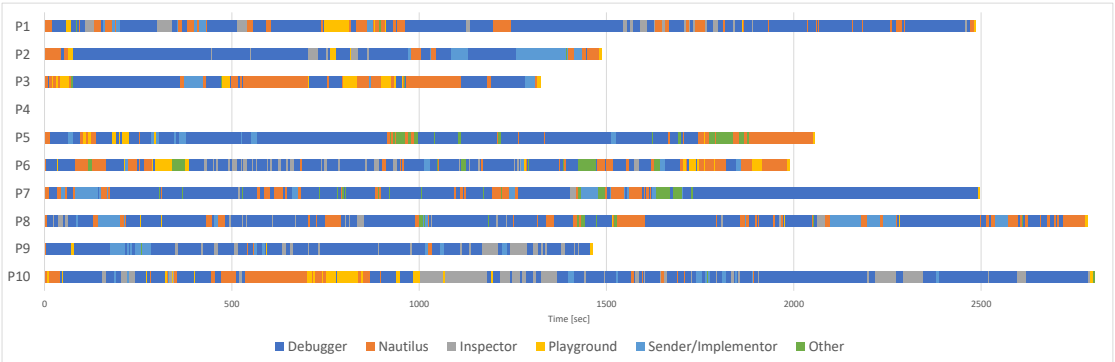

Figure 4.1: Timeline of tool usage during the debugging sessions.

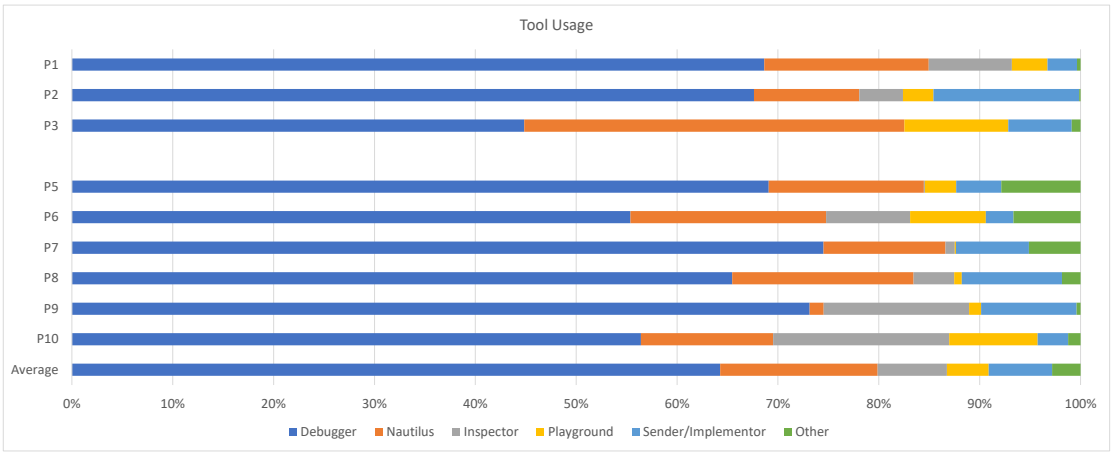

Figure 4.2: Distribution of tool usage during the debugging sessions.

Figure 4.1 shows the tool usage over the whole debugging session for all the participants, while Figure 4.2 shows the the percentage of time spend in each tool. We observe (Figure 4.2) that on average participants spent 66% of the time in the debugger, with participant *P3* spending the least amount of time (44%). This indicates that the selected bug was relevant for investigating how developers use the debugger. Regarding the remaining tools, 16% of the time is spent in the code browser Nautilus. In the inspector and the playground participants spent 5% respectively 3% of the time. They spent 7% for browsing the senders and implementors of methods. And 3% of the time is spent in other tools.

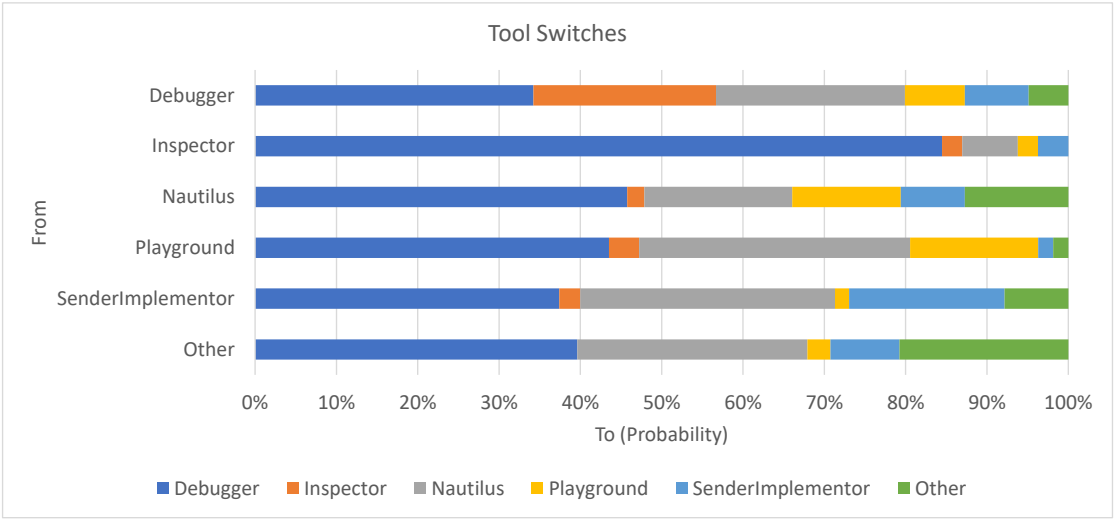

Figure 4.3: Switches between the different tools during the debugging session

We also look into how participants switch between tools (Figure 4.3). We observe that a participant who is currently in a debugger switches to another debugger window with a probability of 34% and to an inspector window with a probability of 22%. When already in an inspector,

with an 84% probability, a participant goes back to a debugger. This indicates a strong link between the debugger and the inspector. Currently, however, when in a debugger or inspector participants mainly opened a new debugger or inspector in a new window. While support exists in the debugger to open an inspector in the same window, participants were not aware of this feature. Providing better integration between the debugger and the inspector, and further allowing developers to spawn new debuggers in the same window using a pager-like interface (as in the inspector) could reduce the window problem [13] often associated with systems using a window based IDE.

When participants were in the Playground they navigated next to a Nautilus window with a probability of 33%. Most of the times when doing this navigation participants first opened a Playground, typed the name of a class and then used the 'Browse' shortcut to open Nautilus on that class. They did this even if they can first open Nautilus and then search for the desired class, or use Spotter, the main search tool from Pharo. A better integration of search features into the code editor could further reduce the number of windows opened during a debugging session.

Last but not least, we also wanted to get a clearer picture of the intensity with which participants interacted with the IDE. To achieve this we use as a proxy the number of clicks (Figure 4.4) and windows opened in 5 minute intervals (Figure 4.5). We observe that there are significant differences between participants. Participant *P2* for example, who successfully solved the task had a low numbered of interactions, not opening more than 10 windows in 5 minutes. Participant *P7* on the other hand opened 20 windows in 5 minutes. This indicates very different styles between participants in interacting with the IDE. Some participants really take their time in determining their next move, while others go much quicker in the hope of gathering more insight. We did not see a correlation between this and the successful completion of the task.

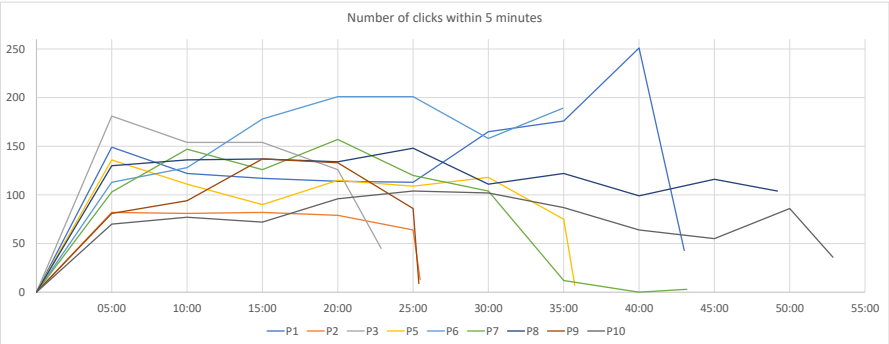

Figure 4.4: Number of clicks measured at 5 minute intervals.

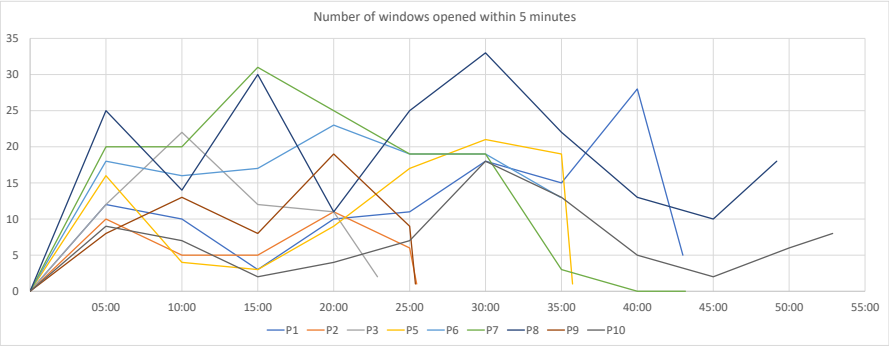

Figure 4.5: Number of opened windows measured at 5 minute intervals.

4.4 Code navigation

In this section we analyze more closely how participants navigate through those parts of the code relevant for the given task. To support this analysis we look at what methods developers visited during the task and observe several relevant paths through the code, highlighted in Figure 4.6.

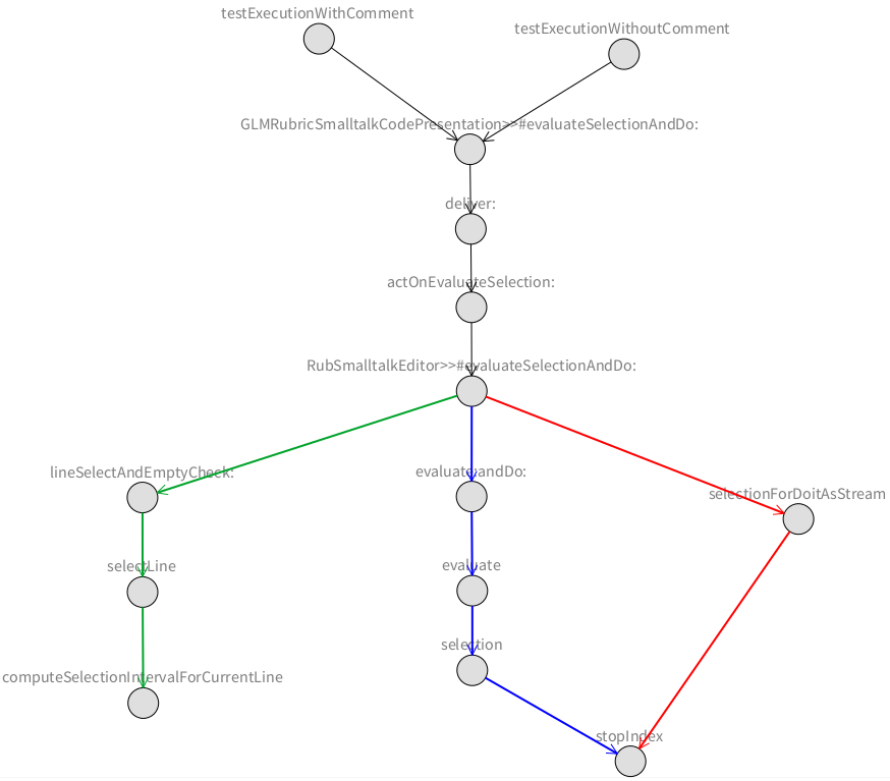

Figure 4.6: Relevant navigation paths through the code.

Initially participants start with *Path 0* by running a test method and opening the debugger. They then step into the method `GLMRubricSmalltalkCodePresentation>>evaluateSelectionAndDo:` where a `GLMEvaluateSelection` announcement is raised. The next step is to find the method

`GLMMorphicPharoScriptRenderer>>#actOnEvaluateSelection:` that handles this announcement. When they reached that method (`GLMMorphicPharoScriptRenderer>>#actOnEvaluateSelection:`), either using the debugger or static code searches, they put a breakpoint there and step into the method `RubSmalltalkEditor>>#evaluateSelectionAndDo:`. At this point we observe three different ways of continuing the exploration:

- Path 1: `RubSmalltalkEditor>>#lineSelectAndEmptyCheck:`
- Path 2: `RubSmalltalkEditor>>#evaluate:andDo:`
- Path 3: `RubSmalltalkEditor>>#selectionForDoitAsStream`

The method `RubSmalltalkEditor>>#evaluateSelectionAndDo:` where the exploration splits into the 3 different paths contains the following code:

```
1 evaluateSelectionAndDo: aBlock
2    "Treat the current selection as an expression; evaluate it and invoke aBlock with the result.
3    If no selection is present select the current line."
4
5    self lineSelectAndEmptyCheck: [^ ''].
6    ^ self
7       evaluate: self selectionForDoitAsStream
8       andDo: aBlock
```

Line 5 does the selection of the code in the playground. Following this path will lead the participant to the bug. For performance reasons, the method directly returns an empty string if the line does not contain any code. In that case the execution can directly return an empty string without invoking the compiler.

In line 7 we see the code `self selectionForDoitAsStream`. This method basically extracts the text of the current line from the beginning to the end of the selection. This text is then stored in a stream and passed to the method `evaluate:andDo:`.

Lines 6-8 contain the call to the method `evaluate:andDo:`. This method takes two arguments. The first argument is a stream that contains the extracted code that we got from line 7. The second argument is the code block that is executed after the result of the selected code is computed.

All of the participants except the ones who solved the bug (*P2* and *P5*) gave little importance to line 5. *P4* and *P10* additionally checked the return value of the line but did not step further into the method. *P5* also skipped the line on the first time and visited the method in a later iteration while *P2* directly stepped into it on the first encounter and started to debug it. We present in Appendix A the time spend by each participant on the three paths.

On line 7 the method `selectionForDoitAsStream` returns a `Stream` object representing the current selection in the text editor. However the participants who are not familiar with how the `Stream` object behaves did not know that they need to explicitly check the contents of the stream to get the actual content. They just looked at the complete stream ignoring the read limit and thought this selection stream is complete. The participants who checked the stream at this position are *P7-P10*. While *P8* did not realize that the stream is not complete and the quote at the end is missing the other three noticed it by checking the content either manually (*P7*) or by our extension (*P9* and *P10*). *P1-P6* did not check the stream at this position. *P2*, *P4*, *P5* and *P6* checked it later in the method `RubSmalltalkEditor>>#evaluate:andDo:` while *P1* and *P3* did not check the stream at all.

After they noticed that the content of the stream is wrong they went back to the call `self selectionForDoitAsStream` on line 7 and tried to find the bug there. This method has the following implementation:

```
selectionForDoitAsStream
   "Answer a ReadStream on the text in the paragraph that is currently
   selected. "
   ^ ReadWriteStream
      on: self string
      from: self startIndex
      to: self stopIndex - 1
```

Here participants hypothesized that subtracting 1 from `stopIndex` is the root cause of the bug. Hence, they removed the subtraction and run the tests again. This however had no effect on the tests, as later on the compiler ignores the actual stream passed to `evaluate:andDo:`. Several participants noticed this after having a closer look at the method `OpalCompiler>>#evaluate` on *Path 3*. There they realize that the content of the stream passed to `evaluate:andDo:` is actually never used because the actual string to be evaluated is computed in the following way:

```
...
   selectedSource := ((self compilationContext requestor respondsTo: #selection)
      and: [
         (itsSelection := self compilationContext requestor selection) notNil
            and: [ (itsSelectionString := itsSelection asString) isEmptyOrNil not ] ])
      ifTrue: [ itsSelectionString ]
      ifFalse: [ source ].
...
```

This code checks if there is a selection set in the playground. And only if there is no selection will it use the content of the stream that participants just tried to fix. However, the line `self lineSelectAndEmptyCheck:` from method `RubSmalltalkEditor>>#evaluateSelectionAndDo:` always sets a non-empty selection in the playground. This means the stream that was passed to this method is not used at all. So changing the stream does not fix the problem.

A similar confusion happens on line 4 in the above listing. The code `itsSelection := self compilationContext requestor selection` calls the method `RubTextEditor>>#selection` which looks like this:

```
selection
   "Answer the text in the paragraph that is currently selected."
   ^self text copyFrom: self startIndex to: self stopIndex - 1
```

This method returns the actual selection of the text editor. Here we have again the subtraction of 1 from the `stopIndex` of the stream. Because removing `- 1` in the previous method `selectionForDoitAsStream` did not fix the problem participants *P2*, *P6*, *P7* and *P9* tried to also change it in this method. Since this method affects the actual selection in the text editor it made the test pass but introduced a new problem; the character directly after the selection is now also included in the returned text – breaking selections where not the complete line is selected. This case was covered by another test which failed after this modification.

	Path 0	Path 1	Path 2	Path 3	Other
P1	16:12	00:00	08:09	00:00	01:52
P2	08:01	06:33	03:57	00:26	01:49
P3	17:21	00:00	00:00	00:00	00:00
P4	~21:00	00:00	~14:00	~5:00	~5:00
P5	16:39	04:58	03:30	01:49	01:40
P6	09:43	00:00	10:28	01:22	01:44
P7	08:32	00:00	23:24	00:51	01:09
P8	13:53	00:00	14:53	01:11	04:14
P9	07:09	00:00	06:44	02:24	01:51
P10	23:40	00:00	03:29	02:55	02:17
Average	13:28	05:45	09:19	01:34	02:04

Table 4.2: Participant's time spent on the different paths. We excluded the estimated values for *P4*'s broken Pharo image from the average.

Table 4.2 shows the time spent by each participant on the four paths. On average most time was spent on *Path 0* (~13 min). This is expected since participants need to visit those methods regardless of what path they will look at next. We observe that only participants that solved the bug correctly entered *Path 1* containing the solution and spent on average 6 minutes on that path. Figure A.1 to Figure A.9 show a visual representation of the time spent in the various methods on the paths for each participant.

We further observe that the rest of the participants chose to explore in detail *Path 2*, and did not discover *Path 1* at all. One explanation is that because they were dealing with unfamiliar code they did not know in advance which path is most relevant. Most participants started with Path 2 and tried to understand why the quote at the end of the stream is missing. They then went into *Path 3* by trying a possible fix that they quickly discovered was not the right one.

4.5 Addressing the Research Questions

Next we used the aforementioned data to discuss the three research questions formulated in Section 1.2.

4.5.1 Debugging Strategies (*RQ1*)

Through our first research question (*RQ1: What debugging strategies are effective in solving the given bug?*) we want to investigate effective strategies used by developers to address the given bug. By qualitatively analyzing the 10 recorded sessions and looking at what makes developers change from one path to another we observe three main debugging styles

- *S1: behavior driven exploration.* Participants switched between paths by observing the live behavior of the application;
- *S2: in-depth path exploration.* Participants focused on a single path attempting to fully understand it before moving to other paths;
- *S3: code driven exploration.* Participants made decisions by looking mainly at static code and relations between code entities.

We first notice that only two participants, *P2* and *P5*, successfully solved the bug. In doing so these two participants explored all the paths presented in Figure 4.6: they started with *Path 0* continued with *Path 2* going into the compiler, *Path 3* related to how an editor extracts the selection and finally ended up in *Path 1* containing the bug. They actual exploration through these paths is highlighted in Figure A.2 and Figure A.4.

These two participants were the only ones that followed *S1*. They both made the switch to *Path 1* when observing the behavior of the application in the debugger. For example, participant *P5* noticed that the selection of the code in the Playground visually changes while stepping over the method `lineSelectAndEmptyCheck`. Participant *P2* manually changed the selection in the Playground and observed that for those cases the selection was computed correctly when on *Path 2*; the participant then suspected that the selection was wrongly computed and investigated the method `lineSelectAndEmptyCheck`.

Most of the remaining participants used strategy *S2*. After finding the announcement handler they immersed themselves in a deep exploration of *Path 2*. When the fact that the wrong selection was not the fault of the compiler became apparent they moved to *Path 3*, however, in the given time they could not backtrack enough to reach *Path 1*.

Strategy *S3* involves the use of tools like *Senders* and *Implementors*. Especially when finding the handler for the announcement, most of the participants used this strategy. Participants *P5*, *P7* and *P9* checked the references to the announcement's class to find where it is used and what method is called when an object receives the announcement. Using the *Implementors* tool they found all the possible handlers and tried to guess the correct handler by looking at the class name or setting a breakpoint in all of them.

4.5.2 Usage of domain-specific information (*RQ2*)

Our second research question was *RQ2: Does domain-specific information improve program comprehension during a debugging session?*. To investigate it, we introduced in the experiment several domain-specific extensions that were available only for *Group 2*. They consisted of an extension showing the actual content of a stream object in the inspector and two extensions showing the list of subscriptions for an announcer directly in the debugger. We analyze next quantitative data related to how much time the two groups spent in tasks related to those extensions, and also make qualitative observations.

Regarding the inspector extension for showing the content of a stream, we observe that participants in *Group 1* took on average 1 minute to realize that the content of the stream is wrong, while participants in *Group 2* saw it instantly by using the inspector extension. In *Group 1* there were two participants more familiar with streams and they checked its contents manually. *P7* checked the contents directly after first encountering the stream and *P2* checked it after ~1.5 minutes. Nonetheless, several participants in *Group 1* (*P6* and *P8*) had difficulties in obtaining this information as they did not know that the actual content of the stream can be different from the complete string stored in a stream. This led to the wrong assumption that the stream was correct and they tried to find the bug at another location in the code. They finally realized this when they inspected the stream again after it was converted to a `String`. In *Group 2* no participant had issues identifying that that stream has the wrong content. Table 4.3 shows, for each participant, the time from the first encounter of the stream until it realized that it is not complete.

	Group	Time	Method
P1	1	-	not checked at all
P2	1	01:29	checked contents manually using method `#contents`
P3	2	-	not checked at all
P4	2	00:00	extension
P5	2	00:00	extension
P6	1	01:10	after conversion to `String`
P7	1	00:13	checking contents manually using method `#contents`
P8	1	00:51	after conversion to `String`
P9	2	00:00	extension
P10	2	00:00	extension
Average G1		00:56	
Average G2		00:00	

Table 4.3: The time it took participants to notice that the content of the stream is not complete, starting from the moment they first encountered the stream object. We also indicate what method participants used to find out the actual content of the stream.

Regarding the second extension we noticed that participants in *Group 1* found the method that handles the announcement on average after 7 minutes while the participants in *Group 2* took about 7.5 minutes. Table 4.4 shows the time it took each participant to perform this part of the task and reveals large time differences between participants. We observe that participants not familiar with this framework took a significantly longer time than participants who encountered the framework before.

Only one of the participants however (*P4*), noticed and used the extension for debugging announcements. This participant found the method handling the announcement in ~2 minutes, Additionally, he spent ~2.5 minutes more to check the action of the announcement manually to ensure that he got the correct information since he was unfamiliar with the new extension; also he added a breakpoint manually in the handler method.

Looking closer at how participants approached the task of finding the methods handling the announcement, we observed three different strategies. The most common one (used by *P1*, *P4*, *P8* and *P10*) was to inspect the announcement and check its subscriptions. Each subscription of an announcement contains information about the method that handles the announcement. *P4* noticed our extension, however, since he was unfamiliar with this new extension he decided to double check the action of the announcement manually. A second strategy was to check which classes are interested in this type of announcement (where it is being registered) and what method is executed when the announcement is received. This strategy was used by *P5*, *P7* and *P9*. *P2* found the method by stepping through the code manually using the debugger until he reached the method handling the announcement. Additionally he also checked the announcement's action. The last strategy that we observed (used only by *P6*) was to skip the search for the handler by searching for the error message showed in the Playground. This allowed him to directly jump into the method `evaluate:andDo:` on *Path 2*.

	Group	Time	Method	Experience with announcements
P1	1	16:58	Checking action of announcement	Not at all
P2	1	03:03	Stepping through the code using debugger	Extremely knowledgeable
P3	2	-	Did not find the method	Slightly knowledgeable
P4	2	04:35	Checking action of announcement & extension	Not at all
P5	2	05:09	Checking event registration	Extremely knowledgeable
P6	1	06:04	Skipped by searching for error message	Slightly knowledgeable
P7	1	01:58	Checking event registration	Extremely knowledgeable
P8	1	06:50	Checking action of announcement	Moderately knowledgeable
P9	2	03:54	Checking event registration	Somewhat knowledgeable
P10	2	16:12	Checking action of announcement	Somewhat knowledgeable
Average G1		06:59		
Average G2		07:28		

Table 4.4: Duration until the participants found the method that handles the announcement sent on path *Path 0*.

Regarding this research question, due to the small number of participants who used the provided extensions we cannot draw any strong conclusions about the improvements brought by domain-specific information. What we noticed nonetheless, is that when developers found and used the provided extensions it helped them in the task.

4.5.3 Usage of debugging extensions (*RQ3*)

Our last research question (*RQ3: How do participants notice and use domain-specific debugging information?*) looks at how developers discover domain-specific extension. To address it we rely on qualitative observations made for the participants in *Group 2*.

We first noticed that there is a significant difference between the usage of extensions in the inspector and debugger. All participants in *Group 2* who inspected a `Stream` object observed and used the *'Contents'* view. However, just one participant noticed and used the views *'All subscriptions'* and *'Active subscriptions'* from the debugger, although we showed them two other debugger extensions – the *'SetUp'* and *'TearDown'* views used in unit tests – before the experiment which works the same way as our extension. The main explanation for this is that when inspecting a `Stream` object the *'Contents'* view appears by default. However, in the debugger participants have to know about those views and switch to them explicitly. The one participant (*P4*) who discovered the views found them useful. All other participants from the experiment (from both groups), however, remarked when reaching the propagation of an announcement that they would like to have in the debugger support for seeing the actual subscriptions (*P1*: "I'm looking for the subscribers to this announcement"; *P6*: "Now I need to look which guy receives the announcement."). Participants also mentioned that it would be helpful to have a feature to automatically jump to the handling method of the announcement when it is delivered to the receiver.

Based on these results we conclude that developers need to be notified when a new extension is available. We discuss in Chapter 5 a solution for visually highlighting new extensions.

4.6 Summary

In this chapter we analyzed the data that we got from the experiment focusing on the participants' different debugging strategies and the tool usage. We observed that for the case of our task the most successful debugging strategy was to observe the application live while stepping through the debugger. We also noticed that when used extensions providing domain-specific information helped participants be faster in solving the task. However, developers often missed these extensions unless they were selected by default. Based on this analysis in the next chapter we look at several concrete proposals for improving the debugger.

5
Improvement Opportunities

In this chapter we discuss possible improvements of the debugger to address problems that arose during the experiment.

5.1 Highlight Extensions

When analyzing *RQ3* we observed that developers did not notice new extensions added to the debugger as they were using it. Hence, we propose to use visual clues to help indicate to developers that a new extension that can be used in the current context has been added to the debugger One way to achieve this and call the developer's attention to these new extensions is to highlight them, when they become available for the first time, with a color different than those of existing extensions. We implemented this idea by creating an extension for the debugger that can recognize new extensions. When this extension is active in the debugger, the very first time a new extension is added the tab representing that extension is highlighted in red.

For example, Figure 5.1 shows how the debugger looks like the very first time the extension for debugging announcements is shown in the debugger. As developers step through the debugger the color highlighting the tab is made less intense (Figure 5.2). After the extension is shown five times it is no longer highlighted. Currently, extensions are highlighted only once. As a future work, after getting developer feedback on this solution, we consider that the mechanism could be extended to also highlight extensions that were not used in a given period of time.

5.2 Graphical Components

During the experiment we noticed that most of the developers ran the test and then started to debug the playground in the debugger. The playground itself was moved to the background and they were focusing only on the debugger. So most of them did not notice that the selection in the playground changed during their debugging actions in the debugger. The result was that the developers missed important hints about which parts of the code were changing the selection.

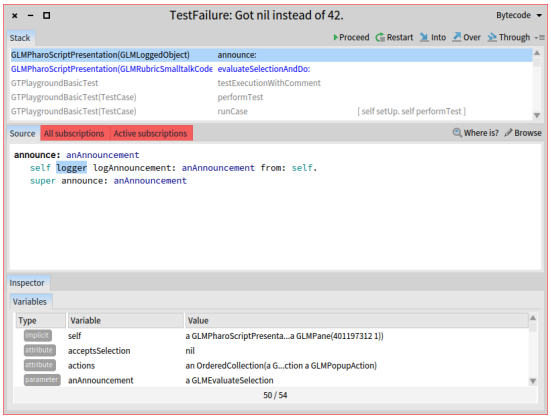

Figure 5.1: Highlighting of the extension on the first appearance.

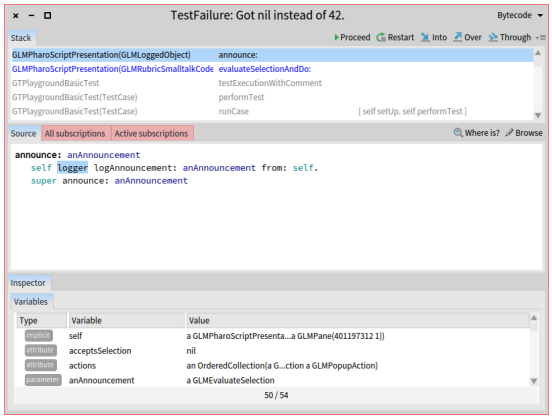

Figure 5.2: Highlighting of the extension after three debugging sessions.

To reduce this problem we created an extension that when debugging a graphical element[1] displays a visual representation of the Morph directly in the debugger (Figure 5.3). With this feature the developer can see the graphical element that she is currently debugging even if the actual window is in the background. Combined with the extension from Section 5.1 the tab showing this extension will be highlighted the very first time the extension is displayed.

5.3 Domain-specific actions

Apart from visually observing that the selection has changed, a different problem faced by participants from both groups was to find the instruction that changes the the selection in the text editor. To address this problem we created an extension (Figure 5.3) for the debugger that when debugging a text editor, displays a button allowing the user to resume execution until the selection in the text editor changes. This is done by installing a conditional breakpoint in the method `RubTextEditor>>#markIndex:pointIndex:` – a low-level method responsible for changing the interval of the selection. This particular extension is mainly useful when reasoning about the text editor. However, this extension is a particular case of halting execution when the attribute of an object changes. Improving the general support for adding these kinds of breakpoints could further improve debugging for many other types of objects.

5.4 Repetitive actions

By analyzing the data recorder during the experiment we further observed that participants repeated several debugger actions multiple times. A common reason for repetitive actions was to reach the same point again. For example participants ran the given test, got a debugger and used it to reach a certain point in the execution. Then they closed the debugger, made some changes to the code, ran the test again, and redid the previous steps to reach the same point again.

We observe many such repetitive actions especially when navigating the delivery of announcement on *Path 0*. This happened as setting a breakpoint on this path is not possible given that the

[1] In Pharo they graphical elements are called *Morphs*

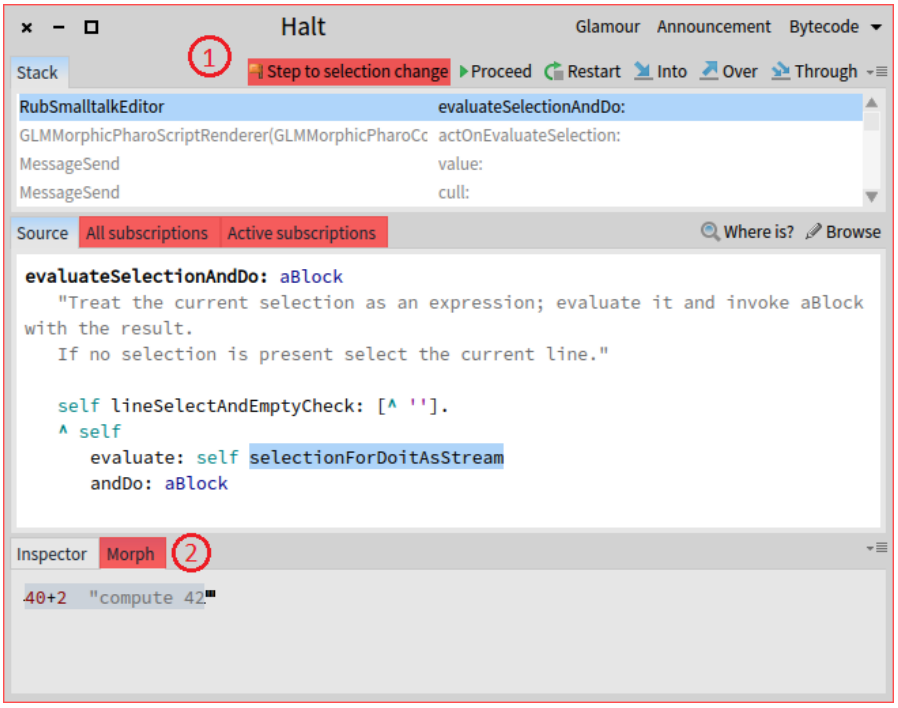

Figure 5.3: When debugging a text editor the debugger displays a button to proceed the execution until the selection in the text editor changes (1). Additionally graphical elements are directly displayed in the debugger (2).

announcements library is used in many other parts in the system. This forced the participants to set the breakpoints at the beginning of *Path 0* and after the debugger hit the breakpoint step through the code manually again until they reached the previous position in the code.

We noticed that *P1* encountered this situation six times and lost around 4 minutes due to these repetitive actions. Most of the other participants spent around 1 minute with repetitive actions caused by opening a new debugger window or using the debugger's restart option and then stepping through the code to the position where they were before. The participants set breakpoints to reduce these repetitive actions. To prevent image crashes by breakpoints in system classes *P2*, *P4* and *P6* used conditional breakpoints. However, *P3* did not use conditional breakpoints which resulted in an image crash. Also *P6* added a breakpoint in a system class leading to a high number of error messages. He was able to repair the image using Pharo's "Emergency Evaluator" but lost around 3 minutes.

To address this issue we created an extension for the debugger that is able to detect this kind of repetitive actions. Whenever the extension detects that the developer is at a position in the code where she has already been before it offers her the possibility to repeat all the debugger actions that she executed during this time. This feature also helps to restore an (accidentally) closed debugger session. With one click the tool brings the developer back to the point in the code where she closed the debugger. Figure 5.4 and Figure 5.5 show the extension that allows the developer to redo repetitive actions or restore a previously closed debugging session. If as the extension is redoing the navigation it detects that execution takes another path, for example because the developers made some code changes, it stops at that point.

Another solution to this problem consists in allowing developers to insert conditional break-

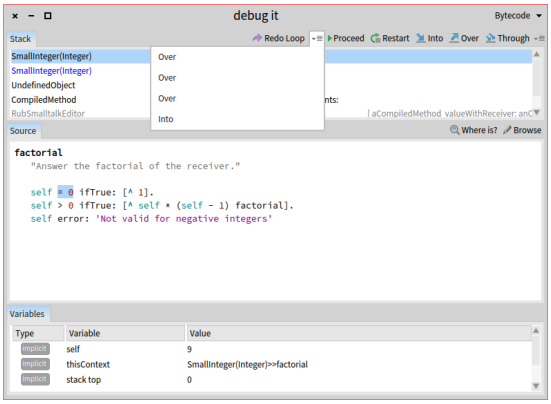

Figure 5.4: The extension allows the developer to redo repetitive actions.

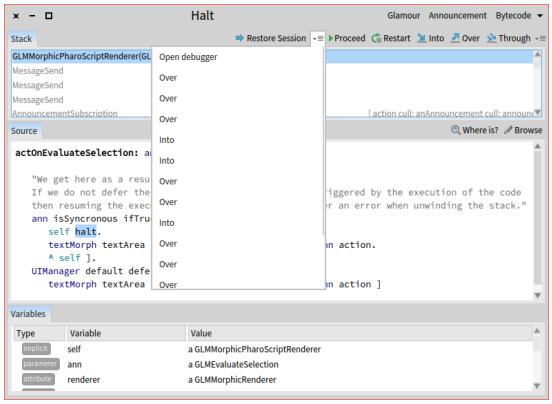

Figure 5.5: The extension allows the developer to restore a previous debugging session.

points in previously visited methods. This would also insure that the execution stops at the same point. For the given situation we chose to record and replay debugging actions, as most repetitive actions were related to the announcement framework. This is a low level framework that is used throughout the Pharo IDE and instrumenting it using conditional breakpoints slows down the execution of the entire IDE. A third alternative that could address this issues that we did not currently investigate consists in using object-centric breakpoints. For example, object-centric breakpoints would allow us to only instrument the announcement objects that is responsible to notify the Rubric editor that code needs to be executed.

5.5 Summary

In this chapter we have shown several concrete improvements that we can make to the debugger based on our experiment. Future work is needed to see how to better generalize these extensions.

6 Conclusions and Future Work

In this section we summarize and conclude our findings and present possibilities for future research and improvements to make debugging more efficient.

6.1 Conclusion

In an experiment with 10 participants we collected data about how software developers use the tools in Pharo to find and solve a bug. Our focus was to answer three research questions related to the debugger. We first observed three types of debugging strategies; *behavior driven exploration*, *in-depth path exploration* and *code driven exploration*. We then introduced several domain-specific extensions to the debugger to see if they help developers. Our analysis showed that developers often only use familiar extensions and ignore new or unknown extensions. Hence, only a few developers used the introduced extensions. However, those that used them benefited from them during the task.

We further presented three opportunities to improve the efficiency in debugging based on the problems and difficulties the participants reported during the experiment. The first one is to highlight new extensions in the debugger to bring more focus to them and encourage the developers to use and explore unfamiliar extensions. Second, graphical components can be displayed directly in the debugger so that developers notice changes in the component even if it's currently in the background. The third improvement opportunity helps to reduce repetitive actions by recording them and allowing the developer to repeat these actions.

6.2 Future work

Our analysis is based on the data collected from 10 participants. To get a more representative view of the results the number of participants can be increased by using and analyzing the data collected from the Pharo community. One way to do this is to use the data we are collecting with GT Event Recorder.

We presented different opportunities to improve the efficiency in debugging a program. The effects of these extensions should be analyzed in detail to see how they affect the developer's behavior and efficiency in her debugging process.

One of these improvement opportunities is to collect and store repetitive actions. Our current implementation allows the developer to redo the previous actions or restore the last debugging session. A more advanced manager for these debugging sessions could allow the developer to also restore older sessions. An additional feature would be to support sharing of the collected data with other developers.

Another of our improvement opportunities is the ability to break the execution of the program when the selection in a text editor changes. This is indeed, a very domain-specific extension. A more general concept for installing breakpoints for interesting events could help the developer to find relevant code more quickly. These interesting events could be defined by the original creator of a framework and be dynamically loaded and displayed in the debugger allowing the developer to install the breakpoints in an easy way.

Based on collected data about previous debugging sessions it would be interesting to see if and to what degree the debugger can be trained to learn about interesting methods using machine learning. This could reduce clicks and interaction with the debugger and improve the efficiency of the debugging process.

One final direction for future work consists in experiments that leveraging existing taxonomies of bug types would investigate differences or similarities in how developers deal with different types of bugs. This could lead to opportunities for specialized tools and techniques, as well as best practices for debugging adapted to the bug at hand.

Bibliography

[1] Andrei Chiş. *Moldable Tools*. PhD thesis, University of Bern, September 2016.

[2] Andrei Chiş, Marcus Denker, Tudor Gîrba, and Oscar Nierstrasz. Practical domain-specific debuggers using the Moldable Debugger framework. *Computer Languages, Systems & Structures*, 44, Part A:89–113, 2015. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[3] David R. Hanson and Jeffrey L. Korn. A simple and extensible graphical debugger. In *IN WINTER 1997 USENIX CONFERENCE*, pages 173–184, 1997.

[4] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for IDEs. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, New York, NY, USA, 2005. ACM Press.

[5] Andrew J. Ko, Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 125–135, 2005.

[6] 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.

[7] Juraj Kubelka, Alexandre Bergel, and Romain Robbes. Asking and answering questions during a programming change task in the Pharo language. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '14, pages 1–11, New York, NY, USA, 2014. ACM.

[8] Thomas D. LaToza and Brad A. Myers. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 185–194, New York, NY, USA, 2010. ACM.

[9] Guillaume Marceau, Gregory H. Cooper, Jonathan P. Spiro, Shriram Krishnamurthi, and Steven P. Reiss. The design and implementation of a dataflow language for scriptable debugging. *Automated Software Engg.*, 14(1):59–86, March 2007.

[10] R. Minelli and M. Lanza. Visualizing the workflow of developers. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4, September 2013.

[11] Roberto Minelli, Andrea Mocci and, and Michele Lanza. I know what you did last summer: An investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 25–35, Piscataway, NJ, USA, 2015. IEEE Press.

[12] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, pages 1–28, 2016.

[13] David Röthlisberger, Oscar Nierstrasz, and Stéphane Ducasse. Autumn leaves: Curing the window plague in IDEs. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE 2009)*, pages 237–246, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[14] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34:434–451, July 2008.

A
Navigation paths

The following figures show the navigation graphs for each participant. Due to the broken image of participant *P4* we excluded his navigation graph. The node color represents the time spent in the corresponding method. The color ranges from green to red where green means the participant spent only a few seconds in the method and red means that the participant spent at least 3 minutes in it. Methods that were not visited at all are left blank.

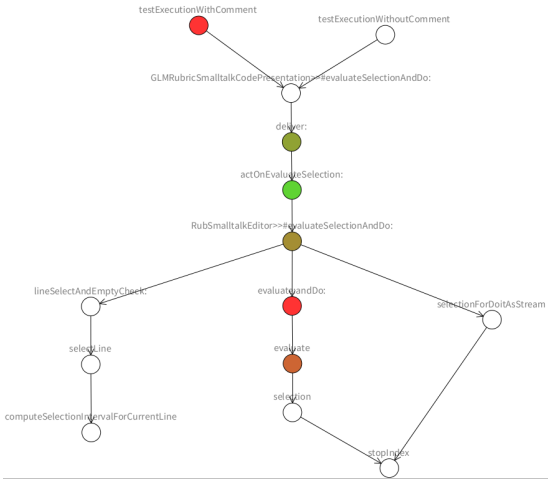

Figure A.1: *P1*'s navigation graph.

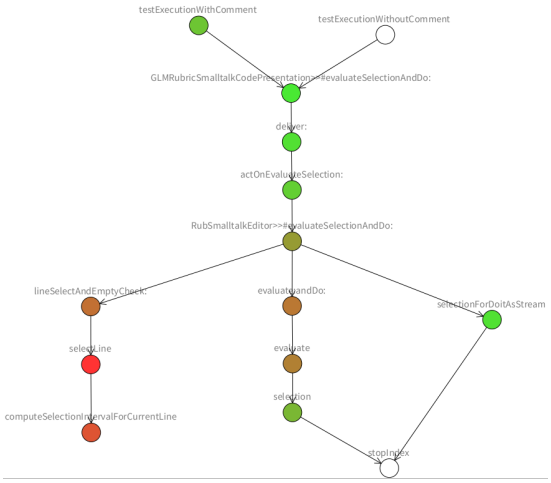

Figure A.2: *P2*'s navigation graph.

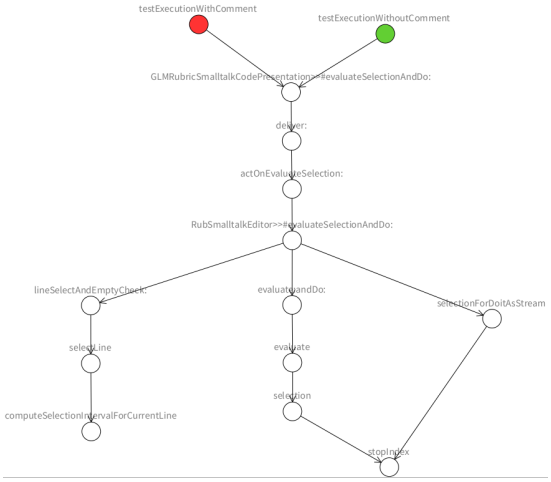

Figure A.3: *P3*'s navigation graph.

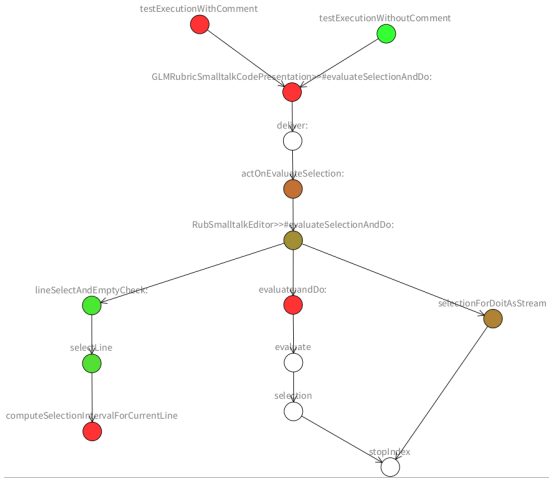

Figure A.4: *P5*'s navigation graph.

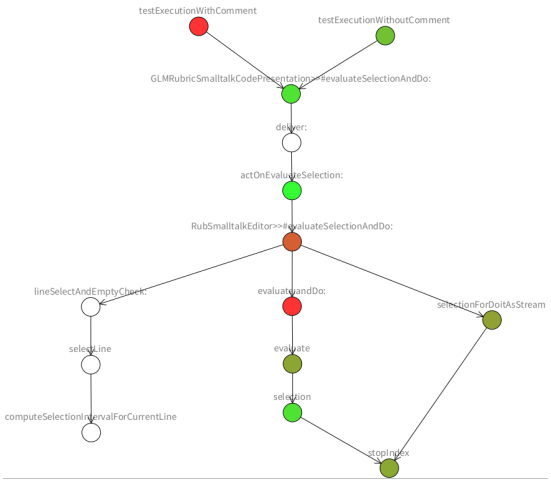

Figure A.5: *P6*'s navigation graph.

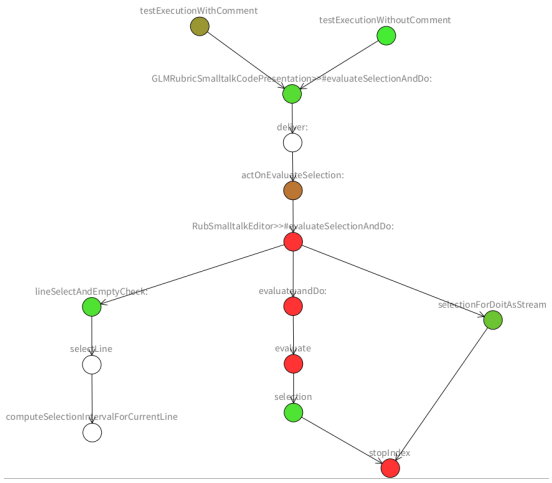

Figure A.6: *P7*'s navigation graph.

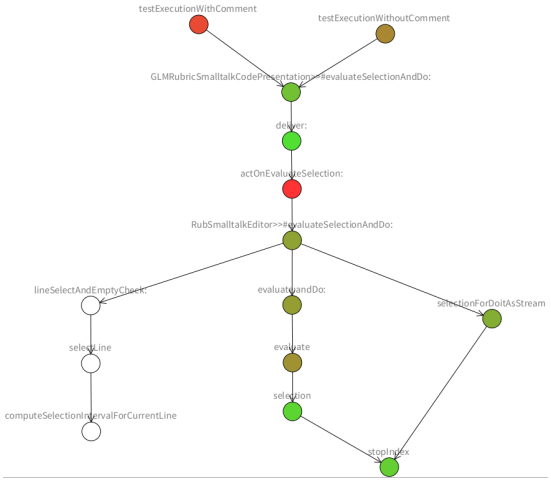

Figure A.7: *P8*'s navigation graph.

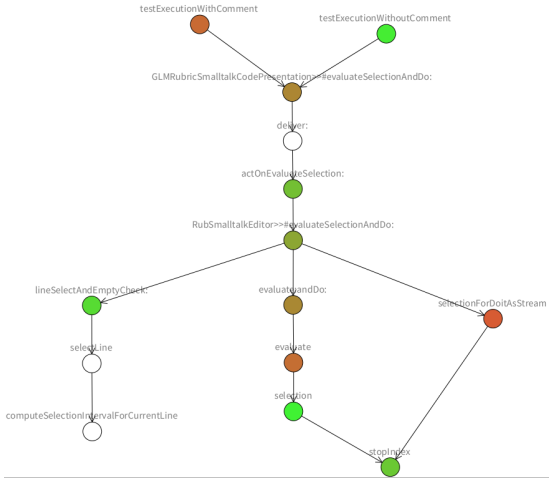

Figure A.8: *P9*'s navigation graph.

Figure A.9: *P10*'s navigation graph.