Investigating Data-flow Reachability Questions

by

James Yoo

B.Sc., The University of British Columbia, 2020

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

October 2022

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Investigating Data-flow Reachability Questions

submitted by **James Yoo** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Gail C. Murphy, Professor and Vice-President, Research & Innovation, Computer Science, UBC
Supervisor

Reid Holmes, Associate Professor, Computer Science, UBC
Supervisory Committee Member

Abstract

Software developers frequently ask and answer questions about code that involve data and its path through a program, such as "How was this value created?" or "How is this value modified?" These questions are instances of *reachability questions*, which require developers to locate points of interest within program paths. Despite how frequently developers encounter reachability questions, existing tools place the burden on the developer to translate the question of interest into a low-level analysis, or to examine analysis results out of context of the reachability question, or both. In this thesis, we introduce the *ReachHover* tool to investigate whether direct user interface support for asking and answering reachability questions makes it easier for developers to answer these questions accurately. We focused ReachHover's support on data-flow reachability questions after conducting a formative study of 72 practicing developers about the type and frequency of reachability questions they encounter in their work. We evaluated ReachHover through a controlled user study with 20 practicing developers, finding that participants who used ReachHover answered data-flow reachability questions involving multiple files more correctly than those who used standard tooling, and that those developers better maintained context while determining their answers.

Lay Summary

Software developers frequently ask and answer questions about the flow of data through a program. To investigate these questions, developers often use tools that require them to translate their high-level questions into a form that is compatible with the tools, while the results provided by the tools often take developers out of their working contexts. We conducted a survey of 72 software developers to understand the type and frequency of difficult questions they ask during their work. Using the results from the survey, we developed a tool that directly supports certain types of questions that developers might ask, and presents information within the context of the code under investigation. To evaluate the efficacy of our tool, we conducted a controlled user study of 20 software developers. We found that developers were more successful in answering questions involving many files, and with less context-shifting actions than with standard tooling.

Preface

All work presented in this thesis was collaboratively conducted in the Software Practices Laboratory at the University of British Columbia, Vancouver (Point Grey) campus. All project and methods were approved by the University of British Columbia Behavioural Research Ethics Board [certificates H21-03473, H22-01608].

James Yoo was the lead investigator, and was responsible for concept formulation, data collection and analysis, and manuscript composition. Gail C. Murphy was a supervisory author on this project. She was involved in all stages of the project, including concept formulation, data analysis, and manuscript composition. A version of this thesis has been submitted for review to a conference.

Table of Contents

List of Tables

List of Figures

Acknowledgments

If anyone by any small chance asks me about my time as a grad student, the first words that come to mind are: "I had help."

This thesis is the collective product of a countless number of people who have helped me along the way, even before I started grad school. For reasons related to space and brevity, I can't thank you all here, but here are just a few:

- Gail C. Murphy, thank you for being the best supervisor I could ever ask for. You always challenged me to think critically about what *really* matters to developers, and never hesitated to call me out when I didn't know what the hell I was talking about. I always left our meetings—when I didn't miss them—feeling re-energized about research.

- Giovanni Viviani, Arthur de Sousa Marques, Anna E., Marie Salomon, Alison Li, the members of my immediate "research family," thanks for listening to my rambling and worrying and for reminding me that there is, in fact, more to life than staying in the lab and doing research.

- Paulette, Jon, William & the PL folks of the Software Practices Lab (SPL), thanks for humouring me and my interest in programming languages; the best part about being in a research lab is how one can soak up knowledge like a sponge just by sitting around.

To the members of the SPL, thank you all for making grad school so fun and occasionally laughing at my *terrible* jokes. Caroline, your comments on my very first conference paper submission were invaluable. Whatever happens, I'm grateful for the time you took to help make the paper the best it could be. Alex, thanks for encouraging me to take a vacation and always being game for a "small beer" or two after work. Reid, thank you for being my second reader, and showing me that faculty members can still have fun. To my friends out there in the "real world," thanks for giving me an occasional escape from academia; I guess I'll be joining you all real soon. Thank you, everyone. I wouldn't have had it any other way.

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). Cette recherche a été financée par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG).

Dedication

To my mentors, my educators, and my teachers
Who often believed in me more than I did—thank you.

Chapter 1

Introduction

Software today is more complex than ever before. There are a plethora of tools that have evolved over the years with the singular goal of making it easier for developers to write code. Code-completion tools building on context-dependent program analysis enable developers to work in unfamiliar systems and software projects [32]. Modern Integrated Development Environments (IDE'S) are capable of generating large amounts of boilerplate code. Recent work in large language models and deep learning has given rise to tools that developers can use to generate entire method implementations [11].

However, developers engage in a wide variety of tasks; writing code is only part of their diverse workflows [29]. Developers also analyze requirements and map out broad flows of code that may be used to complete their tasks. Developers also spend a large amount of time attempting to understand code. This might include activities such as reasoning about the inputs and outputs of a method, sketching out what the call stack looks like, and even asking *why* the code is doing what it is doing [27]. A study by Ko et al. found that developers spent approximately 22% of their time reading code, while a further 16% was spent on navigating dependencies [21]. This equates to developers spending approximately 38% of their time on program comprehension tasks, in comparison to spending approximately 20% of their time on editing code [21].

In short, software developers must ask and answer many questions about code as they work on software change tasks. A specific category of questions that developers might encounter are *reachability questions* [25]. Reachability questions can generally be answered by tracing control- or data-flow through a program. Examples of data-flow reachability questions include "How was this value created?" (i.e., an "upstream" question) and "Where is this value modified in the execution of a program?" (i.e., a "downstream" question). LaToza and Myers found that developers frequently ask reachability questions, with 460 Microsoft developers reporting such questions more than nine times a day. Furthermore, 82% of these developers surveyed rated one or more of the reachability as at least somewhat hard to answer [25].

```
203 FindModel findModel = getReplacementModel(editor); in UsagePreviewPanel.highlight(List<? extends UsageInfo>, Editor, ...)
  185 @NotNull Editor editor, in UsagePreviewPanel.highlight(List<? extends UsageInfo>, Editor, ...)
    87 UsagePreviewPanel.highlight(validUsages, editor, project, false, HighlighterLayer.SELECTION); in ProblemPreviewEditorF
      80 private static void highlightProblems(EditorEx editor, Container editorContainer, List<? extends UsageInfo> usages, @
        77 highlightProblems(editor, editorContainer, usages, project); in ProblemPreviewEditorPresentation.setupFoldingsAn
          50 public static void setupFoldingsAndHighlightProblems(@NotNull EditorEx editor, @NotNull Container editorCon
            46 setupFoldingsAndHighlightProblems(editor, view, usages, view.getProject()); in ProblemPreviewEditorPreser
              33 static void setupFoldingsAndHighlightProblems(@NotNull EditorEx editor, @NotNull InspectionResultsVie
                433 ProblemPreviewEditorPresentation.setupFoldingsAndHighlightProblems(previewEditor, this); in Insp
                  412 previewEditor = panelAndEditor.getSecond(); in InspectionResultsView.showInRightPanel(RefEnt
                    74 return second; in Pair.getSecond()
    154 highlight(infos, myEditor, myProject, true, HighlighterLayer.ADDITIONAL_SYNTAX); in UsagePreviewPanel.resetEditor(L
      66 private Editor myEditor; in UsagePreviewPanel
        143 myEditor = createEditor(psiFile, document); in UsagePreviewPanel.resetEditor(List<? extends UsageInfo>) (filte
          326 return editor; in UsagePreviewPanel.createEditor(PsiFile, Document)
            320 Editor editor = EditorFactory.getInstance().createEditor(document, project, psiFile.getVirtualFile(), !n
              197 return editor; in EditorFactoryImpl.createEditor(Document, Project, ...) (filter: instanceof com.intellij.open
                195 EditorEx editor = createEditor(document, isViewer, project, kind); in EditorFactoryImpl.createEditor
          361 myEditor = null; in UsagePreviewPanel.releaseEditor()
```

Figure 1.1: An example of the tree visualization used by many development environments. This example shows the dataflow to a value named `editor`.

Despite how frequently developers ask reachability questions, little tool support exists to help developers answer them, particularly with respect to data-flow reachability questions. IDE'S, such as Eclipse [13] and IntelliJ IDEA [19], provide tools that report on control- and data-flow analyses, but these tools only indirectly support the answering of reachability questions. For example, to investigate how a value used at a program point is created, a developer using an IDE can invoke an upstream data-flow analysis, which results in a tree-view (Figure 1.1) of responses. Unfortunately, developers find it difficult to answer reachability questions with such views, as it is hard to discern feasible paths and locate targets of interest [25].

We are currently aware of only two specialized tools that have been introduced with the explicit goal of directly supporting the exploration of reachability questions. *REACHER* [26] helps developers iteratively investigate control-flow related reachability questions, but does not help answer data-flow related reachability questions. *Get Me Here* [5] focuses on how a reachability tool can be implemented, transforming reachability questions into constraint-satisfaction problems that are then dispatched to a Satisfiability Modulo Theories (SMT) solver. We are not aware of a user study that evaluated the usability of Get Me Here.

In this thesis, we investigate the provision of direct user interface support to ask and answer reachability questions in the context of the code under investigation by a developer. We provide direct support to *ask* data-flow reachability questions by presenting "How is ..." and "How was ..." questions, similar to the interrogative debugging model [22] first developed by Ko et al. We hypothesize that directly presenting reachability questions to be explored by a developer might improve the usability and discoverability of an interface. We provide direct support to *answer* data-flow reachability questions by displaying relevant analysis results within the context of the code editor and the original code under inspection. We hypothesize that anchoring the asking and

answering of reachability questions in the code of interest enables developers to more correctly and effectively ask and answer reachability questions. We introduce a tool, called *ReachHover*, to investigate these hypotheses. ReachHover automatically surfaces reachability questions at relevant locations in code. In response to selecting the question, ReachHover presents a tree-view of an upstream or downstream data-flow analysis. This tree-view is in the context of the code from which the question was asked, and for each node in the tree-view, it is possible to view the relevant code within an in-editor preview window. This window enables the developer to remain rooted in the context of the code in which the question was asked. ReachHover is implemented as an open-source plugin for the IntelliJ IDEA IDE and platform [41].

To focus ReachHover on reachability questions important to developers, we pose the following research question:

RQ1 What are the reachability questions frequently encountered in practice by developers?

We investigated this research question by conducting a formative study with 72 practicing software developers. We found that developers frequently ask questions regarding how data might be formed and modified in a program, and focused ReachHover on answering these data-flow reachability questions. We pose two further research questions to evaluate whether ReachHover helps developers answer data-flow reachability questions:

RQ2 Are developers able to answer data-flow reachability questions more correctly with ReachHover?

RQ3 Does ReachHover make it easier to answer data-flow reachability questions?

To investigate these research questions, we performed a controlled experiment in which 20 practicing software developers answered two data-flow reachability questions within a moderately-sized system that compared the use of ReachHover and the built-in data-flow analysis support in an IDE.

We found that developers answered data-flow reachability questions involving multiple files more correctly with ReachHover than with built-in IDE data-flow analysis support. We also found that developers switched between and within files less often with ReachHover, demonstrating that the context-preserving nature of ReachHover is helpful. Developers also indicated preferring ReachHover to the built-in support. This evidence supports our hypothesis that providing direct support for asking and answering data-flow reachability questions in the context of the original code under investigation enables developers to answer these questions more correctly and easily.

This thesis makes the following contributions:

- A formative study of 72 practicing software developers about the type and frequency of reachability questions they encounter in practice;

- ReachHover, an open-source plugin for the IntelliJ IDEA IDE that enables developers to more easily ask and answer data-flow reachability questions within the editor, and in the context of the code under scrutiny;
- A controlled user study on the correctness and usability of ReachHover on two program comprehension tasks in a popular open-source software project.

We begin with a discussion of related work in Chapter 2, and present the results of our formative study in Chapter 3. Using our findings from our formative study, we designed and implemented ReachHover, which we describe in Chapter 4. Our controlled user study of ReachHover is presented in Chapter 5, and we discuss how it might be extended to support a wider variety of reachability questions in Chapter 6. We conclude with a summary of the work in Chapter 7.

The thesis of this work is that directly presenting a mechanism for exploring data-flow reachability questions makes it easier for developers to *ask* these questions, and presenting the results of an analysis in the context of the code under investigation makes it easier for developers to *answer* them correctly.

Chapter 2

Related Work

Researchers have investigated the questions that developers ask as they understand and evolve their systems, and have proposed tools that support developers in answering their questions during software change tasks. Developers find it challenging to understand code in modern codebases for a number of reasons. Many codebases today are large and complex, with some meeting the definition of an *ultra-large-scale system* [14]. However, developers have been asking difficult-to-answer questions about code long before the advent of large-scale systems [36]. This suggests that the challenges developers face in understanding code are neither constrained nor precipitated by the size of the system under investigation.

2.1 Questions that Developers Ask

Developers have a large number of information needs that must be addressed during their day-to-day work. These information needs often present themselves as a variety of questions that require the integration of often-disparate sources of information [6, 15]. For example, to understand how a defect was introduced, a developer may have to read the source code of a program, find additional context within a bug report or a project tracker, or resort to speaking to other developers who might have more information.

Other questions that developers may ask are more focused, and may involve querying only one source of information, such as a single program or a collection of modules in a system. As developers work to evolve and understand specific parts of their systems, they may ask: "Where are instances of this class created?" or "What data can we access from this object?" as they try to discover entities and relationships in object-oriented systems [34]. Additional questions that developers may ask are related to the control-flow or the data-flow through a program, such as "Why isn't control reaching this point in code?" and "What parts of this data structure are being accessed in this code?" [34].

2.1.1 Control-flow, Data-flow, and Program Slicing Support

The need for tools to help software developers understand a program has long been recognized. For example, the Interlisp environment allowed programmers to consider control-flow by printing a tree structure of calls [36]. As another example, the FORTRAN FACES system provided support to understand data-flow by reporting for a particular line which variables affected values at that line, or which variables would later be affected by values at that line [31].

Recognizing the value of integrating control- and data-flow, Weiser introduced the concept of a program slice [40]. He defined a slice as a subset of a program that is guaranteed to faithfully represent the original program within the domain of the specified subset of behavior. Program slices help reduce the size of a program a developer needs to consider to reason about and change the system. However, Weiser's requirement that program slices be executable can make slices larger and more expensive to compute than is needed by a developer. A refinement to program slices for software evolution tasks are closure slices [38], which are not necessarily executable and are composed of a set of statements which might be related to a variable of interest via program dependencies. ReachHover is built on analyses in IntelliJ, which provide a form of closure slices.

2.2 Direct Support for Developer Questions

In practice, developers face difficulties finding answers for their questions during development tasks, even with the tooling available to them. These difficulties appear to be in part due to the inevitability of imperfect mappings between high-level developer questions to the low-level analysis facilities afforded by program comprehension tools [7]. In fact, a study of industrial programmers by Sillito et al. showed that participants often struggled to refine and map their questions to tools that could be used to answer them [34].

In this chapter, we discuss the usability of software development tools, as well as issues arising from their lack of discovery. We briefly discuss the *Whyline* [22] as an example tool that attempts to provide direct support for debugging tasks and mitigate the issues related to tool usability and discovery.

2.2.1 Tool Usability

The usability of software engineering tools is also be a significant factor on how developers use tools to help answer their questions. Many software tool developers rely on their own intuition for how a tool should work, eschewing established criteria and procedures for evaluating their usability [37]. Consequently, the tools available to developers may be useful, but ultimately be lacking in terms of their usability. This lack of usability may force developers to explore alternative strategies in understanding and evolving their systems. Developers may also choose to abandon the use of purpose-built program comprehension tools altogether. In a study of 28 developers from a wide

range of companies of different sizes, Roehm et al. found that 22 developers used an IDE. However, none were observed making use of the IDE's built-in program comprehension features, such as call-graph visualization and concept localization [33].

2.2.2 Tool Discovery

Despite the vast amount of tooling available to developers, it is often the case that they only use a small subset of the commands offered by modern development environments, reducing their overall development fluency [30]. The difficulty in making developers aware of the tools available in their IDE'S might be related to the general challenge of making end-users aware of features in software, independent of a specific domain. For example, a common problem faced by users is that they are not aware of a specific tool or operation which was available for use [16]. This might suggest that the lack of discoverability of software tooling and features is a barrier to their wider integration into development workflows.

Improving the discoverability of software tools and features has been the subject of a multi-pronged approach by researchers. For example, some recommendation systems such as *Spyglass* analyze common usage patterns displayed by developers in their IDE'S to recommend features and tools that may improve their efficiency [39]. Other tools may rely on a context-based approach to recommending tools and features. As an example, some IDE'S refactoring tools consider contextual information such as caret or cursor location, text selection, or background analysis of code smells to highlight instances where powerful, but perhaps less well-known, refactorings are available [10]. Communal and crowd-sourced knowledge may also be a source of information that tools leverage to improve tool discoverability. *Toolbox* is an example of a tool that maintains a database of frequently-executed commands by users of a Unix system, and notifies the user community of commands that appear to be newly-discovered [28].

2.2.3 An Example: The Whyline

The Whyline [22] is a tool that provides direct support for developer questions during deubgging tasks. It was initially developed as an interrogative debugging interface for the Alice programming language. Later, the Java Whyline [23] was developed to support the Java programming language. At a high-level, the Whyline enables developers to select questions about a program's output, and it then helps developers to work back from from the selected output to its causes [23]. These questions are phrased as "Why did" and "Why didn't" queries about program output. For example, consider a case when a variable `foo` is being erroneously set to some value `v`. A developer could choose to investigate the cause of this erroneous program behaviour by running the program, opening the Whyline window and selecting "Why did `foo` = `v`?" and a *temporal context*, which is a specified time-frame that they wish to investigate. The Whyline then presents a view of all statements of the program within the temporal context that contributed to the line where `foo` has the value `x`. The

Whyline's approach of presenting a pre-defined subset of questions about the output of programs has benefits, as it avoids speculation about the causes of a failure, and simplifies the exploration of code responsible for the output [23]. An experiment was conducted to evaluate the efficacy of the Whyline on isolating the causes of two bug reports from an open-source project [23]. It was found that the Whyline users were successful about three times as often and about twice as fast compared to the control group who used conventional debugging tools and techniques.

Although the Whyline was designed as a stand-alone tool, it presents an interface to a developer that resembles those of commonly used IDE's such as Microsoft Visual Studio [20], IntelliJ IDEA, and Eclipse. Figure 2.1 shows the Whyline interface as it is used to investigate the output of a Graphical User Interface (GUI) application. Developers are able to choose a question from the context menu that appears next to some observable program output. The Whyline then presents a view that traces the execution of the program to the specific lines that have an effect on the output under investigation. This view is presented in Figure 2.2, where an editor-like view shows two lines of code in different source files that affect program output.

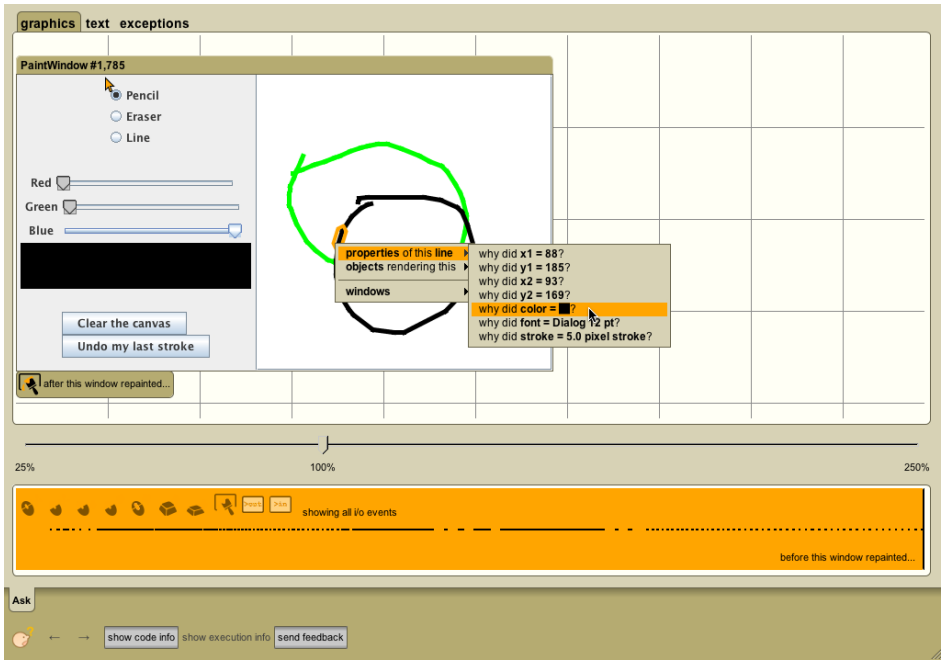


Figure 2.1: The interrogative debugging interface presented by the Whyline. Reprinted with permission from the original author, Amy J. Ko.

2.3 Reachability Questions

Researchers have conducted studies that investigate both the frequency and difficulty of answering some of the focused questions that developers ask during change tasks. LaToza et al. found that a

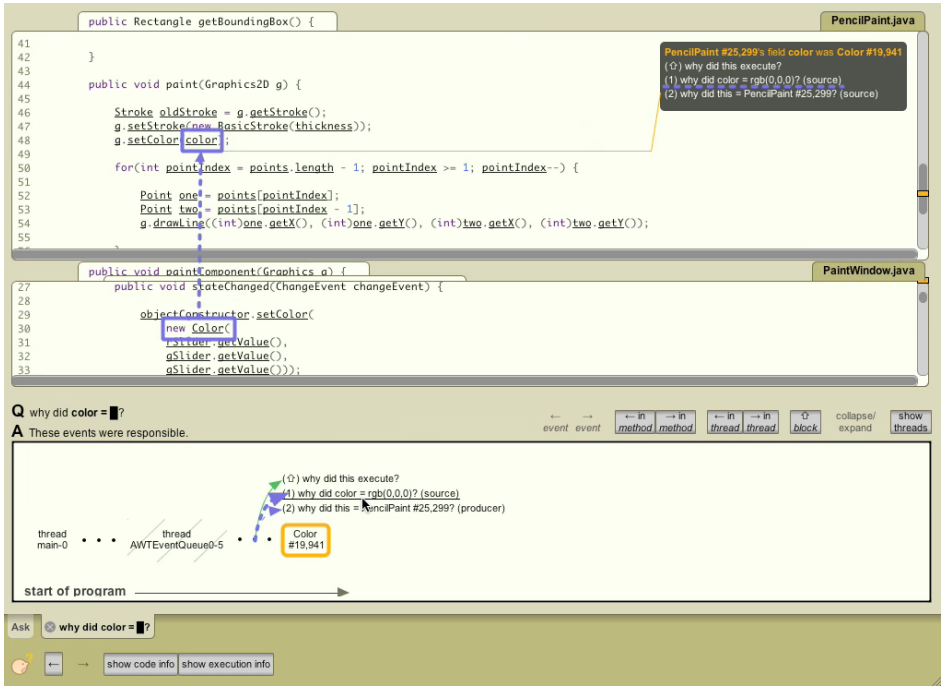


Figure 2.2: The high-level program trace presented by the Whyline. Reprinted with permission from the original author, Amy J. Ko.

significant portion of a developer's work involves answering reachability questions. A reachability question is a search across feasible paths through a program for target statements matching search criteria [25]. In a survey distributed to 2,000 developers employed at Microsoft's Redmond campus, 460 developers reported asking questions that could be phrased as reachability questions more than 9 times a day. Of the developers who responded, 82% of them rated one or more of the reachability questions as at least somewhat hard to answer [25]. Not only are reachability questions difficult to answer, but the process of investigating them is also time-consuming. A field study of 17 developers found that of the 10 longest activities undertaken by developers, 9 were associated with reachability questions [25].

LaToza et al. formalized two types of reachability questions, named *find* and *compare* [25]. A *find* question is of the form:

$$\textit{find } SC \textit{ in } TR$$

where SC and TR are variable parameters. TR represents a set of concrete program traces, where each trace tr is a list of tuples $\langle s, env \rangle$. In each tuple, s is a statement, and env maps each variable in s to a value. SC represents search criteria that acts as a function that prunes the set of

traces TR to a subset that is relevant to the reachability question being asked. As an example of a *find* question, a developer may ask: "Where does the string 'error' come up in the control-flow of this method called `openConnection`?" To formalize this into a *find* question, we define SC as $grep(\text{'}error\text{'})$, and TR as $trace(p,\ m_{start},\ m_{end},\ C)$, where p is the entire program under inspection, m_{start} and m_{end} represent the start and end of the control-flow of the method `openConnection`, and C is a filtering constraint that is left unspecified. The resulting formalization is:

$$find\ grep(\text{'}error\text{'})\ in\ trace(p,\ m_{start},\ m_{end},\ C)$$

The result of this formalization is a a set of traces that contain statements where the string 'error' occurs, which a developer may use to help answer their original question.

A *compare* question requires the analysis of two concrete traces, and is of the form:

$$compare(TR_a,\ TR_b)\text{:}\ TR_{common}, TR_1, TR_2.$$

First, an attempt is made to match each trace $tr_a \in TR_a$ to a trace $tr_b \in TR_b$. The result of this operation may be represented as a list of tuples $\langle tr_a, tr_b \rangle$. For each tuple in the resulting list, *compare* attempts to match $\langle s_a, env_a \rangle \in tr_a$ to $\langle s_b, env_b \rangle \in tr_b$. Each matching tuple is coalesced into traces tr_{common}, that form one of the outputs of the *compare* question: TR_{common}. The tuples in tr_a and tr_b for which no match is found are collected into the outputs TR_1 and TR_2, which represent tuples that were found to be unique to TR_a and TR_b, respectively. A developer may ask a *compare* question as they refactor code. "Is this code correct?" is a common question that is often asked [24]. Assuming "correct" in the context of refactoring to mean that there are no functional changes to code, the resulting formalization of the previous question might be:

$$compare(TR_{pre},\ TR_{post})$$

Where TR_{pre} is the set of traces from the program prior to the refactor, while TR_{post} is the set of traces from the program after the refactor. Using TR_{pre} as a ground truth, the developer might inspect the outputs of the formalization: T_{common}, TR_1, and TR_2, and draw conclusions regarding the correctness of the refactor.

2.4 Answering Reachability Questions

IDE'S such as IntelliJ IDEA and Visual Studio include tools that developers could use to answer reachability questions, even if they were not designed for that purpose. For example, a developer might generate a call graph for a set of methods in a class, and manually traverse it to trace or collect items of interest. Although a number of built-in tools could be adopted in helping developers answer reachability questions, it is also suggested that developers could perform development tasks more

quickly and accurately with tools that more directly support answering reachability questions [25].

2.4.1 Low-level Support for Reachability Questions

A common reachability question that developers might ask is: "Where does this value come from?" To answer this type of question, a developer could invoke a general-purpose slicing mechanism (Figure 2.3) that produces a set of data-flow slices (Figure 2.4).

Although the information produced by mechanisms such as general-purpose slicing and data-flow analysis may be complete and useful, it is difficult to determine their usability by developers. The data-flow traces produced by the data-flow analysis tool are presented in a very compact view. This view enables the presentation of a large number of results that does not require an equally large amount of visual space. However, since each row in the output represents a single line of code, it may be difficult for developers to understand it with the lack of surrounding context. For example, line 200 from Figure 2.4 contains the statement:

```
alphaField.document = alphaHexDocument
```

The only information a developer is able to obtain from this statement is that the `document` field of the `alphaField` object was assigned the value of `alphaHexDocument`. It is not possible to obtain more information about the values in the statement without additional context, which might comprise information useful to developers, such as where or how values in a method were assigned, inline comments regarding the source code, or even the general structure of the code that surrounds the statement of interest.

2.5 Tools that Support Reachability Questions

We are aware of two tools that specifically support the investigation of reachability questions. The first is REACHER, which enables developers to iteratively search along control-flow paths of a program to answer reachability questions involving control-flow [26]. The second is Get Me Here, which investigates how tools might implement direct support for reachability questions by transforming them into constraint-satisfaction problems that are then dispatched to an SMT solver [5].

2.5.1 REACHER: Visualizing Call Graphs

REACHER attempts to simplify the often error-prone and disorienting process [12] of navigating through the control-flow of a program. A developer invokes REACHER by selecting a method declaration and opening a menu that includes the options to search "upstream" or "downstream," referring to the direction of the control-flow paths to be analyzed. Depending on the type of reachability question under they are interested in, a developer selects one of the two options, which opens a REACHER Search View and a call-graph visualization showing methods and calls between

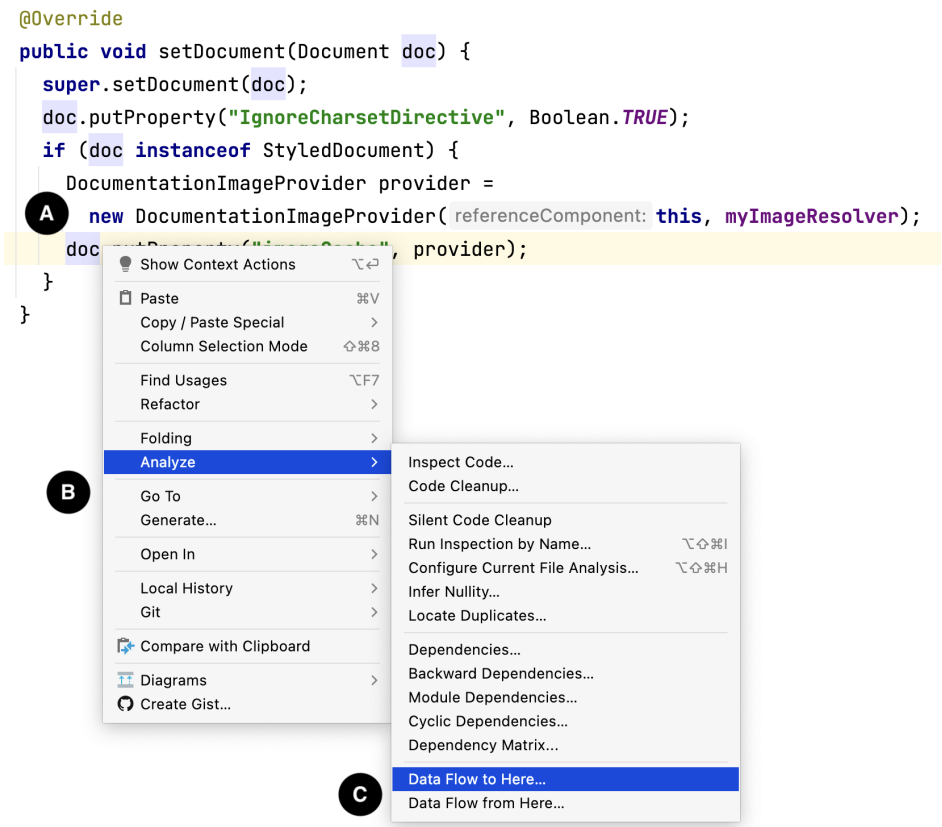


Figure 2.3: A data-flow analysis workflow in IntelliJ IDEA. (A) is the value of interest, `doc`. (B) exposes the variety of analyses that are available to developers for `doc`. (C) invokes a data-flow analysis of values that flow into `doc`.

methods. A user study of REACHER demonstrated the potential value of a tool that directly supports reachability questions. The researchers found that participants who used REACHER answered control-flow reachability questions more successfully and quicker than participants who used standard tools [26]. The same study also showed that participants still spent a significant amount of time reading and understanding code to verify their answers [26]. We seek to mitigate this issue in the design of ReachHover by providing context for the developer through the presentation of code excerpts directly alongside reachability results.

2.5.2 Get Me Here: SMT for Reachability Questions

Get Me Here enables developers to select a line of code as a point of interest and visualize the execution that reaches the given point. Figure 2.5 presents the interface through which a developer would invoke Get Me Here in order to find a relevant program execution trace. Like the Whyline,

```
Analyze Dataflow To:   parameter doc
122 doc.putProperty("imageCache", provider); in DocumentationEditorPane.setDocument(Document)
  116 public void setDocument(Document doc) { in DocumentationEditorPane.setDocument(Document)
      1103 setDocument(new PlainDocument() { in MyTextField.MyTextField() (filter: not-null)
    214 field.setDocument(doc); in ColorPicker.createColorField(boolean) (filter: not-null)
        212 NumberDocument doc = new NumberDocument(hex); in ColorPicker.createColorField(boolean)
      184 hexField.document = HexColorDocument(hexField) in ColorValuePanel
      200 alphaField.document = alphaHexDocument in ColorValuePanel.updateAlphaFormat()
      205 alphaField.document = alphaPercentageDocument in ColorValuePanel.updateAlphaFormat()
      221 colorField1.document = redDocument in ColorValuePanel.updateColorFormat()
      222 colorField2.document = greenDocument in ColorValuePanel.updateColorFormat()
      223 colorField3.document = blueDocument in ColorValuePanel.updateColorFormat()
      234 colorField1.document = hueDocument in ColorValuePanel.updateColorFormat()
      235 colorField2.document = saturationDocument in ColorValuePanel.updateColorFormat()
      236 colorField3.document = brightnessDocument in ColorValuePanel.updateColorFormat()
      166 hexField.document = HexColorDocument(hexField) in ColorValuePanel
```

Figure 2.4: The output of invoking a "Dataflow to here..." analysis on `doc` (Figure 2.3). Each row represents an assignment operation to `doc`. Information such as the line of code, and enclosing method are displayed.

Get Me Here presents a set of analysis options in a context menu that appears near the property or element under inspection.

Where Get Me Here differs, however, is in how it is able to generate an execution trace by using an SMT-based static analysis [5] without having to actually run the program under inspection. It is also presented as a query engine, where developers can specify a slice in a program, and execute queries that represent reachability questions over the given slice.

Get Me Here supports two types of queries. The first is the eponymous "Get Me Here" query, which provides a single execution trace that leads to a point of interest. The second is the "multiple-waypoint" query, which enables developers to investigate an arbitrary number of points within an execution trace. The flexibility afforded by the multiple-waypoint query enables developers to investigate queries that are are highly-parameterizable and specific to the question they have chosen to pursue in localizing a defect or understanding a program. However, with this flexibility comes a danger of enabling program comprehension or defect localization tasks that may be ultimately unfruitful. For example, a query to investigate a erroneous value assignment to a variable might have the possibility of producing results that are entirely unrelated to the defect, but are valid nonetheless with respect to the developer-written query.

Get Me Here displays an execution trace using the Microsoft Debugger Canvas [9], an industrial implementation of the earlier Code Bubbles [8] paradigm for Visual Studio [5]. Figure 2.6 shows how Get Me Here visualizes execution traces. Each method that is called within a trace is represented by a bubble, green highlighting denotes code that was executed, arrows denote the control-flow within a trace, and annotations mark the values of fields and variables.

Get Me Here was evaluated on three benchmark programs, all written in C#. The sizes of the benchmark programs ranged from 650 to 11,000 lines of code. We are not aware of any user study

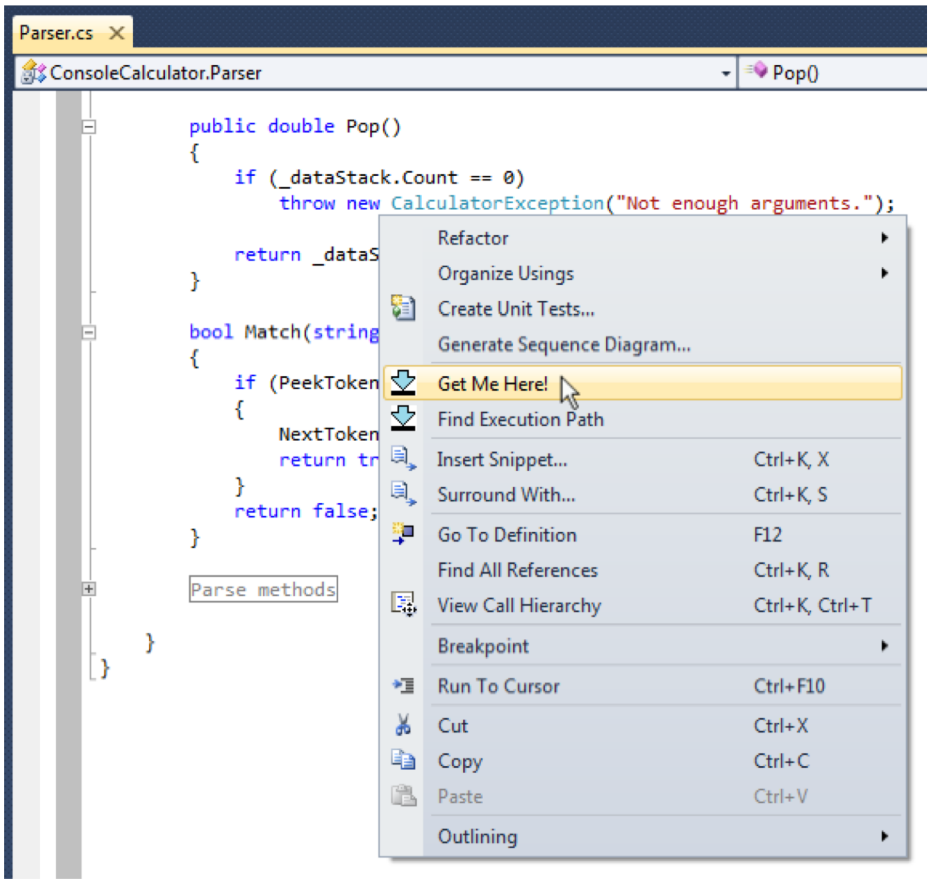


Figure 2.5: The invocation mechanism for Get Me Here in Microsoft Visual Studio. Reprinted with permission from the original author, Robert A. DeLine.

on the usability of Get Me Here.

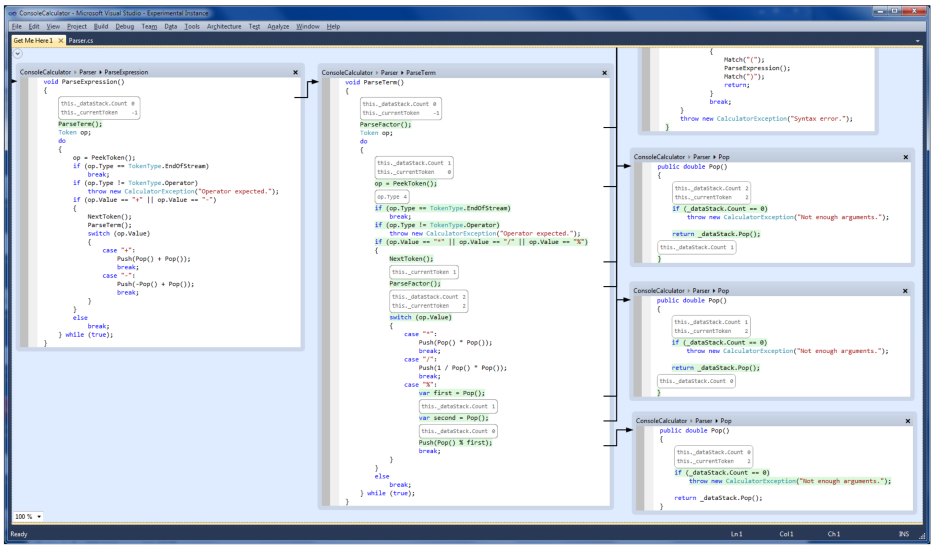


Figure 2.6: The visualization interface of a program execution trace in Get Me Here. Reprinted with permission of the original author, Robert A. DeLine.

Chapter 3

Survey

We performed a survey to investigate the type and frequency of reachability questions that developers may encounter during their day-to-day activities. This aided our investigation of our first research question:

> **RQ1** What are the reachability questions frequently encountered in practice by developers?

3.1 Method

The survey was developed with the goal of maximizing the relevance of the presented questions to actual scenarios that developers would encounter in their day-to-day work. We drew upon our previous work as software developers as well as the literature for reachability questions [25] and hard-to-answer questions about code that developers ask [24]. We drafted a set of 9 questions from these sources that we hypothesized developers might ask as they performed change tasks or explored their programs.

After piloting the survey in a laboratory setting with 7 graduate students, we finalized a survey with 9 questions with corresponding code excerpts with an expected duration of 10 minutes. The survey was deployed entirely online, and consisted of:

- 5 demographic questions;
- 9 Likert-scale questions that each included a hypothetical scenario associated with a program comprehension or reachability question.

Eight of the nine questions included a short code excerpt to contextualize the reachability question being asked. The remaining question was contextualized by a short paragraph describing a development scenario instead, as it did not lend itself well to a small code excerpt. The text of the survey that includes all code excerpts as presented to our respondents is available in Appendix A.1; the anonymized raw data is also available [42].

3.1.1 Participant Recruitment

We distributed the survey on Twitter, and via mailing lists to professional software developers. We targeted developers who primarily use a statically-typed programming language in their work in order to exploit the pre-existing sophisticated control- and data-flow tooling available for them in our tool. Individuals who reported not writing code at least once a month were excluded from our survey, as they may not be the primary audience for our tool.

3.2 Survey Results

The survey was started 108 times, and finished 72 times in total. This yielded a survey completion rate of 67%. First, we describe the demographics of the respondents to our survey, exploring their experience as software developers and the programming language(s) used in their day-to-day work. We summarize the years of development experience, the most frequently used programming language, and the range of sizes of the companies that employ our respondents in Table 3.1. We then use these results to inform our analysis of our respondents' experience with reachability questions.

3.2.1 Demographics

Most of our respondents were located in North America, including Canada (22) and the United States (22). The second-largest majority of our respondents came from Europe, including Germany (8), Russia (8), and the United Kingdom (2). The rest of our respondents came from wider range of countries in the APAC region and South America. Approximately 83% of respondents reported writing code every day; the remainder reported writing code at least twice a week, or once a month.

Many of our respondents had over seven years' worth of development experience (31%), while 31% of respondents had two to five years' worth of experience. The remainder of the respondents had zero to two, or five to seven years' worth of total experience (Table 3.1).

Our participants reported using a total of 15 unique programming languages. Java, Kotlin, Python, C and C++, and TypeScript (Table 3.1) were the languages that were reported to be used most frequently by our participants. Although other languages used by our respondents compose a large number of results (35%), they are generally composed of statically-typed languages including Swift, C#, and Objective-C.

A majority of our respondents reported being employed at companies that have more than 1,000 employees (58%). 19% of respondents reported working at a company with less than 100 employees, and the remaining 23% of respondents reported working at companies that had a range of 100 to 1,000 employees.

Table 3.1: Respondent demographics.

Demographics	
Years of Experience	
0 - 2 years	25%
2 - 5 years	31%
5 - 7 years	14%
Over 7 years	31%
Programming language used most frequently	
Java	14%
Kotlin	13%
Python	14%
C/C++	11%
TypeScript	11%
Other	35%
Number of employees in company	
Over 5000	25%
1000 - 4999	33%
500 - 999	8%
100 - 499	15%
Less than 100	19%

3.2.2 RQ1

Table 3.2 presents a view of the results of our survey in terms of the mean scores assigned to each question by our participants. We use the results to inform our investigation of RQ1:

RQ1 What are the reachability questions frequently encountered in practice by developers?

Table 3.2: Survey results ordered by mean.

Order	Question	Type	Control-flow	Data-flow	Mean ($\bar{x}$)	SD (σ)
Q2	Did I introduce any unwanted changes in the new version of this code?	*compare*	✓	✓	4.12	1.10
Q9	What does the control-flow look like between two locations in code?	*find*	✓	✗	4.08	1.23
Q1	Where does a value come from, and how is it formed?	*find*	✓	✓	4.03	1.05
Q5	Given some data, which parts of it are modified downstream?	*find*	✓	✓	3.76	1.22
Q8	Given some part of a program that depends on a value, which parts of it are executed or reachable?	*find*	✓	✓	3.59	1.33
Q7	Given two subtypes and their implementations of a common method, how do they handle data differently?	*compare*	✗	✓	3.51	1.41
Q6	Is deleting what appears to be unused code going to break anything?	*compare*	✓	✓	3.34	1.53
Q4	Given some data, which parts of it are accessed downstream?	*find*	✓	✓	3.24	1.26
Q3	How is an instance of this class created/initialized?	*find*	✗	✗	2.95	1.34

The highest-ranking question (Q2) represents situations where a developer adds a feature or refactors code, and then wants to ensure that no defects are introduced as part of the process. This reachability question is a compare-type question where developers might compare program traces before and after the addition of changes. The other two compare-type questions (i.e., Q7 and Q6) were ranked substantially lower.

The second highest-ranked question relates to the control- flow of a program, specifically between two locations in code. This type of question is supported by the existing REACHER tool.

The remaining questions are all data-flow related. Q1 (i.e., "Where does a value come from, and how is it formed?") can be formulated as a backward reachability question, where a data-flow trace is generated backward from the value of interest. Q5 (i.e., "Given some data, which parts of it are modified downstream?") can be formulated as a forward reachability question starting from a value of interest and generating data-flow traces in which the value of interest might be modified. Q8 relates to how the value of data might affect its path through a program (control-flow), or how it is affected by dynamic dispatch (which implementation of a method is called). Q4 is similar to Q5 in that it represents a "downstream" reachability question, but our participants appeared to be less interested in how data might be accessed rather than modified. Finally, like Q1, Q3 also asks about the origin of data, but constrains the possible question to how a class might be initialized.

In summary, we found that developers frequently encounter reachability questions that are related to the data-flow of a program during their work.

3.3 Threats to Validity

In this Chapter, we discuss the threats to the validity of our findings from the data collected from our formative study.

3.3.1 Internal Validity

It might have been the case that each participant had a slightly different understanding of the questions that we asked in our study. Differing interpretations of each question might have affected the responses that each participant assigned to each questions. This presents a threat to the internal validity of our findings. To mitigate this threat, we provided code excerpts and descriptions to clarify the meaning of questions. The code excerpts in our survey were static elements, meaning that the only way our participants could interact with the code was to read them in isolation. Although this furthered our design goal of making the questions encompass the core concepts behind reachability questions, it did not accurately capture how developers interact with code (e.g., IDE-based navigation, other workflows and tooling) in program comprehension tasks. To mitigate this threat, we may have had to design our survey around an IDE that our participants would be able to use. However, it is likely that this would have introduced another threat to

validity; our results may be affected by the proficiency of our participants with the IDE we select to use in our study. Consequently, we deemed the lack of interactable code excerpts to be an acceptable threat.

3.3.2 External Validity

The external validity of the results is impacted by the size of our respondent pool. Our survey was started by 108 participants and ultimately completed by 72 individuals, yielding a completion rate of 67%. All of these participants reported having professional software development experience, with 45% reporting at least 5 years of experience.

The selection of the questions in the survey also present a threat to external validity as they might not be completely representative of the reachability questions that developers encounter in practice. To mitigate this threat, we included reachability questions that were found to be asked by developers in the field [24, 25] in addition to the questions we encountered during our work as software developers.

Another threat to validity is presented by the fact that all the code excerpts included in our survey were written in Java. This does not account for the fact that our participants may have varying levels of proficiency with Java. Consequently, we may be conflating our results with the proficiency of our participants with Java. That being said, we needed to select a language for our study that would have a high likelihood of being familiar to our target audience. Java's popularity as a development language [1, 2] meant that it was a suitable choice.

Chapter 4

The ReachHover Plugin

Both of the existing tools that support a subset of reachability questions, REACHER and Get Me Here, provide direct support for asking specific reachability questions. However, both of these tools display the answers to the reachability questions in a visualization separate from the code. A user study of REACHER found that participants still had to locate code in an editor related to information about code shown in the visualization.

To mitigate this issue and explore what a tool that provides direct support for both *asking* and *answering* reachability questions might provide, we built ReachHover [41], an open-source plugin developed for the JetBrains IntelliJ Platform and IDE. Figure 4.1 presents how a developer might use ReachHover to answer a reachability question involving the creation of an object.

In designing direct support for asking reachability questions, we were inspired by the Whyline [22, 23], which augments support for debugging programs directly via "Why" and "Why not" questions about a program's observable behaviour. In designing direct support for answering reachability questions, we sought to provide answers in the context of the code in which the question is being asked. By maintaining this context, we believe the developer can be more rooted in the original context of where they began their exploration, and mitigate the disorientation developers may experience when they thrash between multiple views that display information relevant to their task [12]. We describe how a developer invokes a reachability question and how the results are displayed before briefly describing ReachHover's implementation, including the reachability questions it currently supports.

Chapter 4.1 discusses the design methodology for ReachHover, derived from the data presented in Chapter 3, and an investigation of current tools that provide information that might be used to help answer reachability questions. Chapter 4.2, describes how a user might use ReachHover in a software development scenario, with a particular focus on its mechanism of invocation and how it visually presents data. The implementation of ReachHover is detailed in Chapter 4.3.

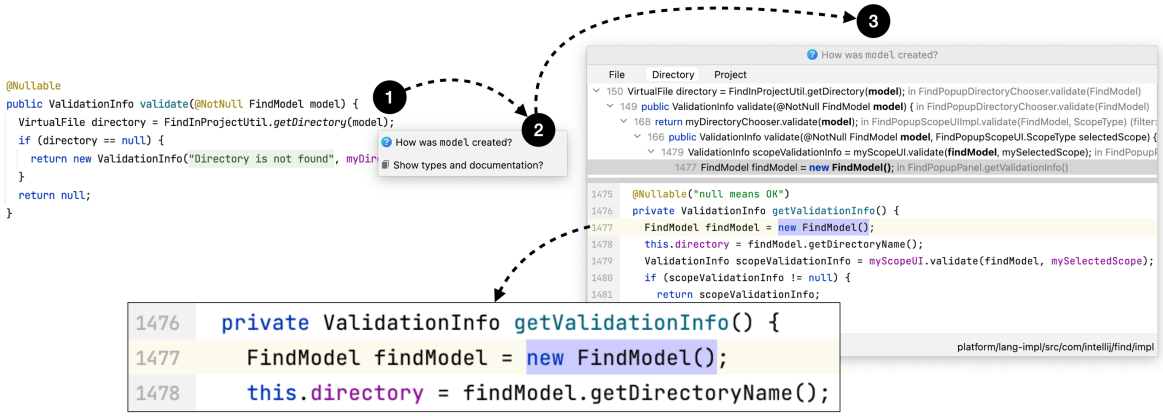


Figure 4.1: Using ReachHover to investigate an upstream reachability question. Hovering over the method argument `model` (1) results in the ReachHover tooltip (2). Clicking on the question "How was `model` created?" presents the results of a backward data-flow analysis in the context of the original code under inspection with a preview of relevant code (3).

4.1 Design Methodology

We analyzed the data from a survey (Table 3.2) that sought to identify the type and frequency of reachability questions that are asked by software developers in practice. We used the results obtained from analyzing this data to inform our design of ReachHover.

We chose to focus ReachHover on data-flow related reachability questions as existing tools have not yet considered these questions. We further chose to focus on *find*-type questions as they present more straightforward scenarios involving one data-flow or program slice rather than the *compare*-type questions, which require at least two slices. As a result, we decided to focus on the two highest-rated questions of this style from the survey (Q1 and Q5):

Survey Question 1 (Q1):
Where did a value come from, and/or how was it formed?

Survey Question 5 (Q5):
Given some data, which parts of it are modified downstream?

These questions can be answered by traversing backward and forward in a data-flow trace, respectively. For example, to answer Q1, a user could invoke a backward data-flow trace, and filter for nodes that have a control or data dependency on the value under analysis. The same analysis could be performed on a forward data-flow trace to answer Q5.

Since information provided from a data-flow trace could be used to answer the questions we attempt to support with ReachHover, we began by investigating existing data-flow analysis tools to obtain an understanding of the current landscape of analysis tooling. The IntelliJ IDEA IDE exposes forward and backward data-flow analysis capabilities to users. Figure 2.3 shows how a user might invoke a backward data-flow analysis, while the result of this analysis is shown in Figure 2.4. We found these pre-existing data-flow capabilities to be a good basis for the development of ReachHover due to its ease of extension via the JetBrains IntelliJ Platform Software Development Kit (SDK).

Next, we began to identify pain points that users might face as they invoked the data-flow analysis tool. We observed that discovering the tool itself might be difficult for users; invoking the tool is a three-click action. Additionally, the action is hidden behind a context menu that does not guide users to discover and invoke the tool from a very broad list of options (i.e., data-flow-specific analyses are accessed via a general "Analyze" menu element). In the result presented by the data-flow analysis tool, each element in the data-flow trace is presented as a node in a tree-like structure that is structurally similar to a conventional hierarchical tree. We found this to be useful for preserving the structural information relevant to our target reachability questions (e.g., method calls `a` and `b` within a method `c` are displayed as children of the node `c` in the tree).

Although structural information is easily preserved through the tree, each node displays only the single line of code within the data-flow trace, often without visual aids such as syntax highlighting. This means that users analyze each node in isolation from any surrounding context, particularly if they are not aware that double-clicking a node in the tree focuses the editor pane to the line of code under inspection. However, frequent shifts in where and what the editor is focused on might introduce the additional problem of frequent context switching and reconstruction as focus shifts between files and code blocks that might lead to disorientation for developers [12].

Investigating the current tooling available for data-flow analysis, and its associated user pain points provides a foundation for the design of ReachHover. We aim to achieve the following with the design of our tool:

Design Goal 1:
User friction should be minimized between the developer and ReachHover, especially with respect to how they might discover and invoke the tool.

Design Goal 2:
ReachHover should present the context surrounding a data-flow node in a way that minimizes the amount of context switching for a developer.

We describe the user-facing result of implementing these goals in Chapter 4.2, while a technical discussion of their implementation is discussed in Chapter 4.3.

4.2 Using ReachHover

In this section, we describe how a user can interact with ReachHover, framed within the context of a software development task. First, we describe how a user can invoke our tool to begin the process of answering a reachability question. Next, we describe the user interface of the visualization presented by ReachHover.

4.2.1 Invocation Interaction

A developer may want to use ReachHover when they are investigating how a value was created, or where it originated in a data-flow trace. Work in information search and retrieval systems [17, 18] has provided empirical evidence that cursor hovering can be a useful proxy for determining a user's focus or interest in a user interface component. We use this empirical evidence to inform our design of an invocation mechanism for ReachHover that is based on cursor attention and hovering.

The initial invocation mechanism for ReachHover is shown in Figure 4.2. In this figure, we assume a task where a developer is investigating the origin of the value `editorKit`, which is an instantiation of Q1 from our survey (Chapter 3). The hover popup displays two options; the first is to invoke ReachHover to begin a reachability analysis, and the second is to show the documentation that is usually presented by IntelliJ IDEA when a user hovers over a value. This hover popup appears only only when a user hovers over a value after a certain delay that is commensurate with other user interface component delays in IntelliJ IDEA. We do not consider our interaction mechanism to be more intrusive than the default data-flow analysis feature of the IDE; a user would have had to right-click to select the value of interest and navigate a context menu to invoke the analysis (Figure 2.3). Furthermore, we attempted to address Design Goal 1 by presenting the ability to invoke ReachHover directly on hover, rather than hidden under a number of context menus where user discoverability and friction might have increased. It is also possible for the user to not use ReachHover altogether, the default context menus provided by IntelliJ IDEA are still available. Another task where a developer might use ReachHover is in exploring how a value is modified or changed; Figure 4.3 shows how the hover popup appears in this case.

4.2.2 Result Visualization

A goal of ReachHover is to present more context surrounding the results of a data-flow analysis. We also wanted to preserve the structural information that was already provided by existing static analysis tools, such as the data-flow analysis tooling built into IntelliJ IDEA, or the call hierarchy explorer in the Eclipse IDE, which presents results using the same hierarchical tree paradigm.

Preserving the presentation of structural information serves two purposes. First, we did not aim to completely redesign an interface for viewing data-flow information; our goal was to enrich an existing interface with additional information. Introducing a completely new user interface

```
protected DocumentationEditorPane(
  @NotNull Map<KeyStroke, ActionListener> keyboardActions,
  @NotNull DocumentationImageResolver imageResolver,
  @NotNull Function<@NotNull String, @Nullable Icon> iconResolver
) {
  myKeyboardActions = keyboardActions;
  myImageResolver = imageResolver;
  if (ScreenReader.isActive()) {
    // Note: Making the caret visible is merely for convenience
    getCaret().setVisible(true);
  }
  else {
    putClientProperty("caretWidth", 0);
    UIUtil.doNotScrollToCaret( textComponent: this);
  }
  HTMLEditorKit editorKit = new HTMLEditorKitBuilder()
    .withViewFactoryExtensions(ExtendableHTMLViewFactory.Extensions.icons(iconResolver::apply),
                               DocumentationHtmlUtil.getModuleIconsExtension())
    .withFontResolver(EditorCssFontResolver.getGlobalInstance()).build();
  addDocumentationPaneDefaultCssRules(editorKit);
}
```

How was editorKit created?

Show types and documentation?

A

Figure 4.2: The ReachHover popup (A) that appears when a user hovers over a code structure of interest. In this case, ReachHover poses the question: "How was `editorKit` created?"

paradigm might introduce additional complexity in the form of developers having to adapt to a new paradigm before our tool became useful to them. Second, we wanted to preserve non-local context, i.e., information that is not directly observable from code and its surrounding context, such as how deep within a call hierarchy a location in code exists.

We present the visualization component of ReachHover in Figure 4.4. Information regarding a reachability question is presented in two main sub-components. High-level non-local context is provided in the top half of the main component, which is the same component used by the original data-flow analysis tool built into IntelliJ IDEA. Fine-grained local context is provided by the preview editor sub-component. A developer is able to investigate code from a high-level by selecting a row in the top component. This triggers a change in the preview editor to focus on the line of code under inspection, which is highlighted in its location in the editor. Additional information, such as the file and project directory in which the line of code under inspection is located, is presented in the bottom toolbar of the main component. Additionally, a user is able

```
override fun actionPerformed(e: AnActionEvent) {
  val focusManager = IdeFocusManager.findInstance()
  val focusedComponent = focusManager.focusOwner
  val ideWindow = focusManager.lastFocusedIdeWindow
  val dataContext = if (ideWindow === focusedComponent || focusedComponent === focusManager.getLastFocusedFor(
      ideWin  How is dataContext modified?   nce().getDataContext(focusedComponent)
  else Data  Show types and documentation?   ontext
                                              A

  if (!firstOpened) {
    super.actionPerformed(AnActionEvent(
      e.inputEvent, dataContext, ActionPlaces.RUN_TOOLBAR_LEFT_SIDE, templatePresentation,
      ActionManager.getInstance(), modifiers: 0))
    firstOpened = true
  }
  else {
    super.actionPerformed(AnActionEvent(
      e.inputEvent, dataContext, ActionPlaces.KEYBOARD_SHORTCUT, templatePresentation,
      ActionManager.getInstance(), modifiers: 0))
  }

}
```

Figure 4.3: The ReachHover popup (A) that appears when a user hovers over a code structure of interest. In this case, ReachHover poses the question: "How was `dataContext` modified?"

to select the scope of a reachability analysis (i.e., file, directory, and project-level) on-the-fly and without resorting to restarting the entire analysis from scratch, avoiding the need for expensive context switching or reconstruction.

4.3 Implementation

ReachHover is implemented in the Kotlin programming language, and consumes Application Programming Interfaces (APIS) implemented in Java that are exposed by the IntelliJ IDEA Platform. The implementation of ReachHover can be separated into two main components: a front-end and a back-end. The front-end detects cursor hover events and analyzes code for reachability analysis viability, while the back-end exploits existing data-flow analysis features to visualize data for reachability analysis.

4.3.1 Front-end: Detecting Values for Reachability Analysis

The ReachHover front-end (Figure 4.5) is designed as a loosely-coupled component that aims to be largely orthogonal to a specific programming language or project structure. We implement an existing event listener interface that is exposed by the IntelliJ IDEA SDK for public use in order to create a class that listens for hover events in the editor (`EditorHoverListener`). Whenever a code element is detected under a hover event, we call a utility class (`SliceDispatchService`) to

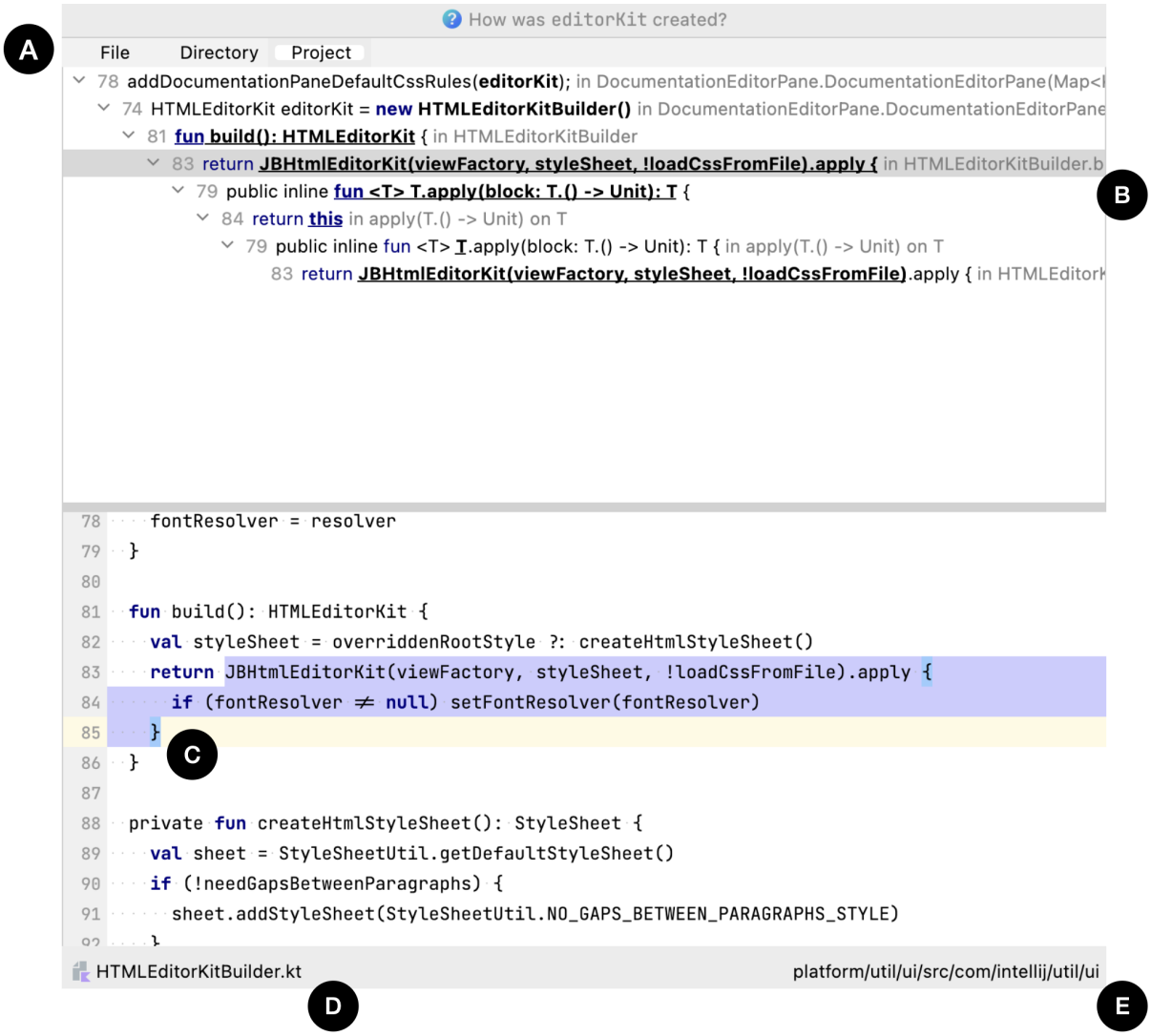


Figure 4.4: A data-flow trace presented by ReachHover. (A) enables developers to select between a file, directory, or project-level scope for the reachability analysis. (B) contains the same hierarchical text-tree view as the standard IntelliJ data-flow analysis tool. (C) demonstrates the highlighting of the code under inspection that corresponds to the selected element in (B). The file name and directory are provided by (D) and (E).

determine whether a program slicer is available for the code under inspection. If a slicer is available, we convert the raw code element into a Unified Abstract Syntax Tree (UAST) node.

The UAST data type enables us to handle program elements in a generalized manner. Consequently, the logic that determines whether an element is viable for reachability analysis does not change across different programming languages. The UAST data type has support for programs

written in Java or Kotlin. We have extended the UAST data type with two methods that enable us to determine whether a given code element is a non-literal method argument, or a local variable reference. These specific code elements are candidates for the questions "Where did a value come from, and/or how was it formed?" or "Given some data, which parts of it are modified downstream?" respectively.

Using these methods, the `EditorHoverListener` dispatches a call to the back-end of ReachHover to conduct either a forward or backward reachability analysis by sending it a context object that contains the element under inspection, its location in the editor, and other metadata.

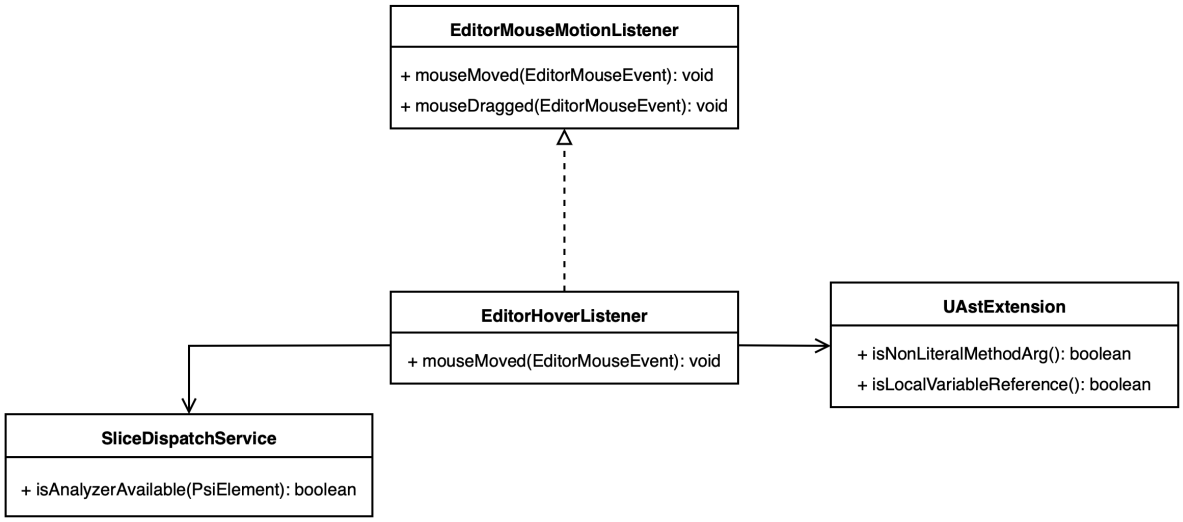


Figure 4.5: A simplified architectural diagram of the ReachHover front-end.

4.3.2 Back-end: Computing Reachability Information

The back-end of ReachHover was designed with the goal of re-using as much of the existing data-flow analysis tooling available in the IntelliJ IDEA IDE. The reasons for this were twofold. First, we wanted to save valuable development time by avoiding the re-implementation of components and tooling that we could re-use for our needs. Second, by deeply integrating our tool with the common frameworks and APIS exposed by the IntelliJ IDEA Platform, we hoped to maintain compatibility with a wider range of tools that are built on the same platform.

The `ReachabilityInfoPopupManager` class serves as the main entry point from the front-end of ReachHover. This class receives a context object from the front-end `EditorHoverListener` that contains information such as the element under inspection and the direction of the reachability analysis to be performed from the front-end `EditorHoverListener`. The popup manager first checks whether the element from the context object differs from the element it has cached from

when it was last called. If a new element is detected, the entire context object is passed down to a class (`ReachabilityPopupBuilder`) that is responsible for continuing the analysis and creating the reachability popups presented in Figures 4.2 and 4.3.

`ReachabilityPopupBuilder` consumes the context object and creates the two components of the hover popup: the reachability analysis button and the button which a user may click to show the type information and documentation related to the element under inspection. The two reachability questions supported by ReachHover are represented by two subtypes: `BackwardReachabilityButton` and `ForwardReachabilityButton` of the `ReachabilityButton` supertype. The implementations of these subtypes are largely identical, with a key difference in the type of data-flow slice they request from the `SliceDispatchService` object, which contains the program slicing facilities exposed by the IntelliJ IDEA SDK. As the name might suggest, the `BackwardReachabilityButton` requests a *backward* data-flow slice, while the `ForwardReachabilityButton` request a *forward* data-flow slice. Both these buttons execute the `ShowReachabilityElementsAction` on their event handler that registers mouse clicks.

When a reachability button is clicked by a user, the `ShowReachabilityElementsAction` is invoked, which consumes the data-flow slices that were previously computed by the button. The action instantiates the `ReachabilityPanel` (Figure 4.4), and begins decorating it with the program slice information and metadata. The hierarchical tree representation of the slice is generated by the `SliceTreeBuilder` class, which traverses the slice and assigns each element in the slice a corresponding data-flow node that appears in the tree.

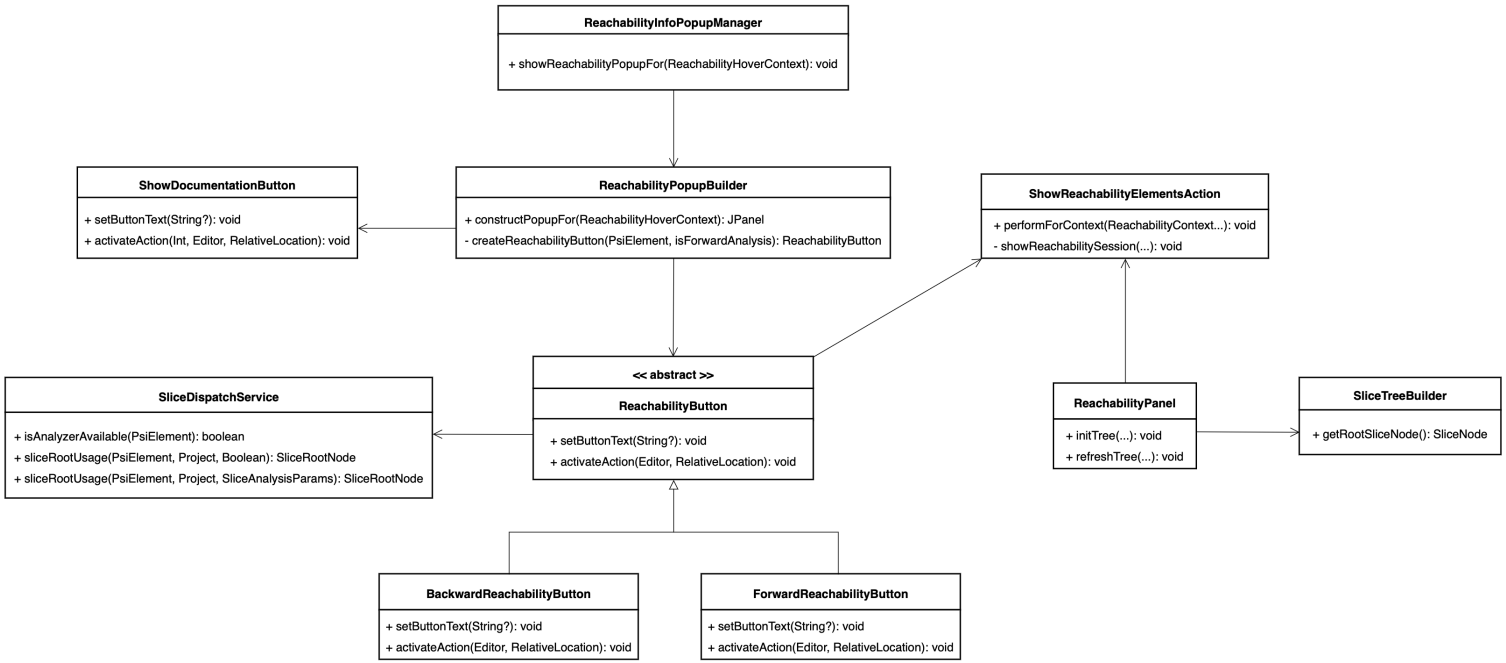


Figure 4.6: A simplified architectural diagram of the ReachHover back-end.

Chapter 5

Evaluation

ReachHover aims to make it easier for developers to answer upstream and downstream reachability questions through the use of data-flow information. We conducted a controlled user study [43] to investigate two research questions that helped us explore our goal:

RQ2 Are developers able to answer data-flow reachability questions more correctly with ReachHover?

RQ3 Does ReachHover make it easier to answer data-flow reachability questions?

The study compared the use of ReachHover with the built-in data-flow analysis capabilities of the IntelliJ IDEA IDE. Each study participant was asked to complete two tasks where each task involved either a forward and a backward reachability question involving data-flow for a medium-sized software system implemented in a statically-typed language. The order of tasks and order of tools—ReachHover or IntelliJ—was randomized across participants. At the end of the study, participants were asked to report on their experiences.

5.1 Recruitment

20 individuals participated in the study. These participants had a wide range of years of professional software development experience: 25% of participants had up to 2 years of professional software development experience; 50% had two to five years of experience; and the remaining 25% had at least 5 years of experience. 90% of our participants reported that they currently use a statically-typed programming language (e.g., Java, Scala, Go) in their professional work. The remaining 10% reported using a statically-typed language for professional work within the last five years.

5.2 Tasks

We used the Apache NetBeans [4] project as a target system for the tasks in the study. NetBeans is an open-source IDE comprising approximately 900,000 lines of Java code. We designed tasks that represented both forward and backward data-flow related reachability questions (i.e., "How was ... " and "How is ... "). The information needed to answer the reachability question for one task was localized to a single file (i.e., *intra*-file); the question for the other task required the consideration of many files (i.e., *inter*-file). The first task, t_{document}, posed a forward reachability question within a single 1,035 line file. Participants were asked: "How is `findReplaceResult` modified?" where `findReplaceResult` is a variable that can hold an object containing data related to the find-and-replace operation in the NetBeans IDE. Participants were asked to invoke ReachHover or IntelliJ and report the number of times a null check for the variable `findReplaceResult` appeared along with their locations in code downstream of a particular starting point. Additionally, they were also asked to provide the methods where it might have been used, (i.e., calling methods on it, using it in an expression), and the methods where its value is not used at all (i.e., passthrough methods). Participants were asked these questions to make specific the reachability concept of 'using a value.'

The second, t_{bookmark}, posed a backward inter-file reachability question. Specifically, participants were asked: "How was `bookmark` created?" where `bookmark` is a method argument that represents a user-defined bookmark within a file in the NetBeans IDE. Participants were asked to invoke ReachHover or IntelliJ and to provide the number of times a call to the `bookmark` constructor appeared, the unique locations in code where it might have been invoked, the names of the methods where the value of `bookmark` might have been used before being passed as a method argument, and the locations where `bookmark` or any related objects might have been assigned a value. These specific questions were asked to clarify the reachability concept of 'creating and assigning a value.'

5.3 Method

A study session consisted of asking a participant to perform the t_{bookmark} and t_{document} tasks in a counter-balanced design. Table 5.1 outlines the four treatments, labeled T_{A} through T_{D}. For example, participants assigned to T_{A} first completed the t_{bookmark} task with ReachHover, and then completed the t_{document} task with the built-in data-flow analysis features supplied by IntelliJ. The order of tasks and tools were swapped to mitigate the efforts of transfer learning.

Participants undertook the study on their own time remotely. Each participant was shown a short video before each task that demonstrated the features of either ReachHover or the built-in data-flow analysis tooling of IntelliJ IDEA. After completing the tasks, participants were asked to report on their experiences with ReachHover and how it compared to tools they would have used otherwise to complete the tasks. We assigned each participant to a treatment randomly, with the only constraint being to have an equal number of participants per treatment as much as possible.

Table 5.1: User study treatments.

Treatment	Task Ordering	Tool	# Subjects
T_{A}	t_{bookmark} t_{document}	ReachHover Built-in data-flow	6
T_{B}	t_{bookmark} t_{document}	Built-in data-flow ReachHover	5
T_{B}	t_{document} t_{bookmark}	ReachHover Built-in data-flow	5
T_{C}	t_{document} t_{bookmark}	Built-in data-flow ReachHover	4

We collected usage metrics (e.g., mouse click events, editor interactions) within the IntelliJ IDEA IDE during the task. All logs remained on-device until participants were asked to upload them at the end of the study. We concluded the session by posing the following experiential questions to participants:

EQ1 "Did you find any difference using ReachHover or IntelliJ for the tasks in this study?"

EQ2 "Any other thoughts or comments?"

Ultimately, the number of participants in each treatment was not perfectly balanced, due to factors such as attrition or incorrect logs uploaded by participants.

5.4 Data Analysis

We considered three measures to analyze the data we collected from each experimental session. The first was *correctness*, where we measured how participants performed on each task using ReachHover or the built-in data-flow analysis capabilities of IntelliJ. The second was *visual momentum* [12], which we used as a proxy for the amount of context-shifting actions each participant was performing. Lastly, we considered the *experience* each participant had with ReachHover and IntelliJ by conducting a separate card sort [35] on their comments.

5.4.1 Correctness

We defined a correctness score (S) to quantify how participants performed on each task.

$$S = \mathrm{ANS}_{\mathrm{correct}} - \mathrm{ANS}_{\mathrm{incorrect}}$$

This score is composed of two components where:

- $\mathrm{ANS}_{\mathrm{correct}}$ is the number of correct answers;

- $\text{ANS}_{\text{incorrect}}$ is the number of incorrect answers.

For example, incorrect answers might be lines of code or methods that do not appear in the data-flow trace, or lines of code or methods that do appear in the data-flow trace, but are not related to the value of interest.

5.4.2 Visual Momentum

Research has shown that developers can become disoriented when user interfaces lack visual momentum, a qualitative measure of a users' ability to extract information across changing displays [12]. A display that is low in visual momentum might induce developers to thrash between files to collect information necessary for a task [12]. To examine the visual momentum experienced with IntelliJ's built-in data-flow analysis tool and and ReachHover, we consider a measure of editor cursor jumps. A cursor jump is recorded when a developer moves their cursor from one location in a file to another location, either within the same file or another open file. Editor cursor jumps in the *main* editor can affect the context (i.e., cursor highlighting, lines of code presented, viewport) within the editor. We measure cursor jump events in the main editor as a coarse-grained proxy for the amount of information-seeking and context-shifting a developer is encountering.

Editor cursor jumps in the *preview* editor are recorded when a developer moves the cursor to a different location within a file open in the preview editor. Unlike main editor cursor jumps, preview editor cursor jumps do not affect the context of the main editor or any open files, and constrain the changing information in the bounds of the preview editor. Preview editor cursor jumps indicate that the developer might be looking at a different line in the same preview editor. We measure cursor jump events in the preview editor as a proxy to help determine whether a developer is actively using ReachHover's preview editor to obtain information related to their task.

5.4.3 Experience

To investigate comments that participants made about experiences with the different tools, a researcher unaffiliated with the development of ReachHover and the study conducted a separate card sort [35] on the qualitative responses to the experiential questions (EQ1 and EQ2) at the end of the survey. The results of analyzing our participant's responses to EQ1 are presented in Chapter 5.5.2, while we discuss EQ2 in Chapter 6.2.

5.5 Results

We analyzed the results of our user study with respect to our initial research questions that asked whether developers are able to answer reachability questions correctly with ReachHover, and whether ReachHover makes it easier for developers to answer reachability questions.

5.5.1 RQ2

For tooling to be useful, it must help developers answer questions about their code correctly. We considered RQ2 to help us investigate this within the context of our user study:

> **RQ2** Are developers able to answer data-flow reachability questions more correctly with ReachHover?

To investigate this question, we computed a correctness score (S) for each of the participant's two tasks in the study. Table 5.2 presents the average correctness score ($\bar{S}$) for each combination of tool and task in the study.

Table 5.2: Average correctness scores ($\bar{S}$) for each combination of tool and task in the study.

Tool	**Task**	$\bar{S}$
ReachHover	t_{bookmark}	4.1
Built-in data-flow	t_{bookmark}	0.8
ReachHover	t_{document}	4.6
Built-in data-flow	t_{document}	4.1

From this table, we can observe that participants had a much higher average correctness score with ReachHover for t_{bookmark} (4.1) than with the built-in data-flow analysis tools of the IDE (0.8). There was less of a difference observed for t_{document}. Figure 5.1 and Figure 5.2 provide a more in-depth look into how participants fared with each tool on t_{bookmark} and t_{document}, respectively.

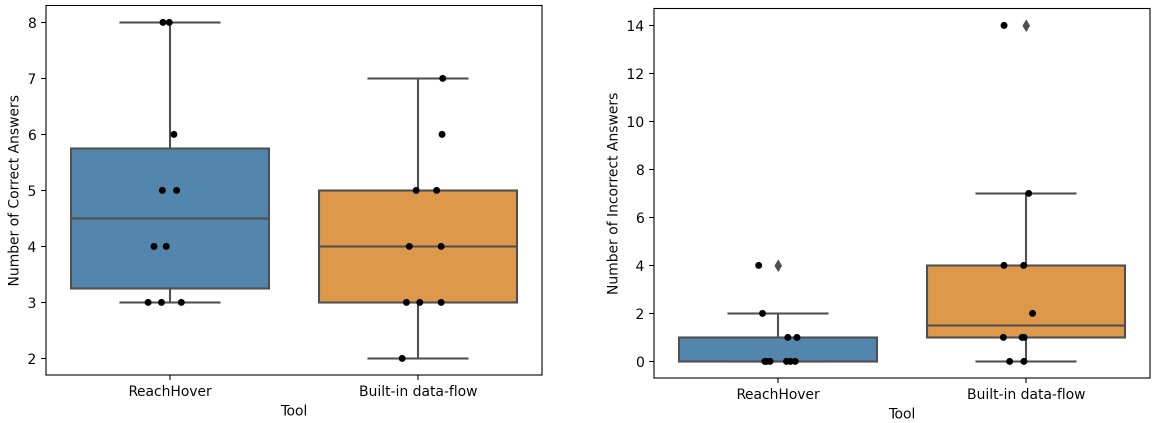


Figure 5.1: A comparison between ReachHover and the standard data-flow analysis capabilities of IntelliJ IDEA in terms of correct and incorrect answers given to t_{bookmark}.

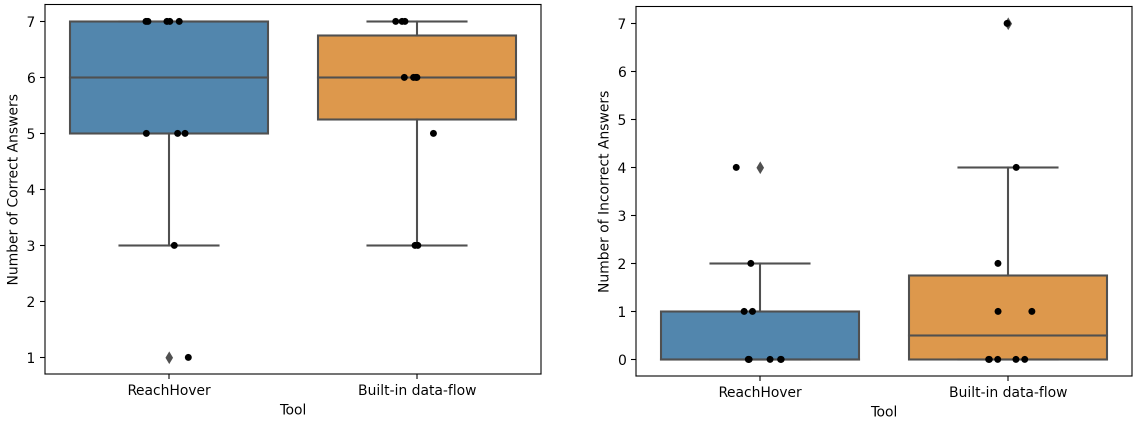


Figure 5.2: A comparison between ReachHover and the standard data-flow analysis capabilities of IntelliJ IDEA in terms of correct and incorrect answers given to t_{document}.

5.5.2 RQ3

Whether or not a tool is easy to use requires considering how developers perceive and experience a tool. To investigate these factors, we posed RQ3:

RQ3 Does ReachHover make it easier to answer data-flow reachability questions?

We considered quantitative data from logs of activity within the IDE provided by our participants, and the qualitative answers they provided to a comparative question about the two tools. Figure 5.3 presents the weighted average of main editor cursor jumps that were recorded per tool in each task of the study.

For both tasks, we observe that there are fewer weighted average main editor cursor jumps when ReachHover was used in comparison to when the built-in data-flow analysis tooling of IntelliJ was used. For t_{document}, the inter-file question, this difference is statistically significant (Mann-Whitney $U = 6$, $n = 10$, $p = 0.001$, $\alpha = 0.05$). We also measured the number of jumps in the preview editor for each task where ReachHover was used. We observed a weighted average of 58.7 jumps for t_{bookmark}, and 89.2 jumps for t_{document}. The higher number of preview editor jumps compared to main editor jumps when ReachHover was used suggests that our participants used the preview editor that is part of the tool to locate task-relevant information in each task.

We conducted a separate coding for each of our participant's responses to the questions we posed regarding their experiences using ReachHover and IntelliJ's built-in data-flow analysis tooling. The coding was undertaken by a researcher that was unaffiliated with the our formative study, the development of ReachHover, or the user study.

Table 5.3 reports a summary of the codes assigned to participant responses to the question "Did you find any difference using ReachHover or IntelliJ for the tasks?" (EQ1). Nineteen of the twenty participants responded to the question; each participant is coded as one response in Table 5.3.

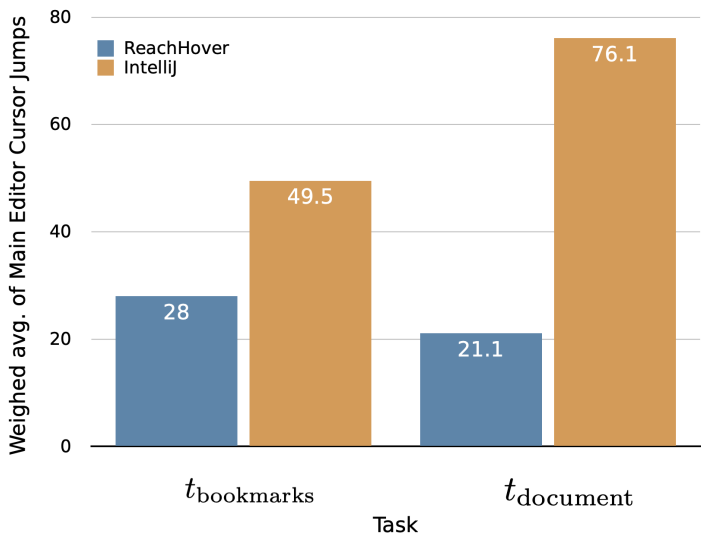


Figure 5.3: The weighted average of main editor cursor jumps recorded per tool in each task.

Table 5.3: Codes assigned to each participant's response to EQ1.

Code	Description	Count
Prefers IntelliJ	User prefers IntelliJ's built-in data-flow analysis tooling	2
No Perceived Difference	User did not find a difference between ReachHover and IntelliJ's built-in data-flow analysis tooling	4
Easier or Simpler	User found ReachHover easier and/or to use than IntelliJ's built-in data-flow analysis tooling	5
Context-Preserving	User found ReachHover preserved more of their working context during a task	5
Usable	User found ReachHover more usable than IntelliJ's built-in data-flow analysis tooling	3

Thirteen of the participants (65%) expressed positive comments toward ReachHover. Five (25%) expressed that ReachHover was easier to use:

> *"ReachHover felt less cluttered and made the task easier"* - P18

Three participants remarked that ReachHover was more usable than IntelliJ, both for asking and answering reachability questions:

> *"ReachHover was easier to call as it didn't require navigating menus."* - P9

> *"I find having ReachHover's separate floating windows with a view into where the related line of code is, is more usable than IntelliJ's data-flow analysis feature."* - P10

Five of the participant's responses (25%) helped confirm what we observed from analyzing editor jumps. These particiapnts noted that ReachHover helped to maintain their working context:

> *"I preferred that I could look at the matches in the same modal window without affecting the open file"* - P19

> *"I felt like I did not lose the context using ReachHover compared to IntelliJ. I felt they showed me the same content, but ReachHover was easier to use (maybe beause [sic] I didnt [sic] lose the context)"* - P15

A smaller number of participants (6 or 30%) did not note any positive difference for ReachHover. Four participants (20%) did not report a perceived difference between ReachHover and IntelliJ, while two (10%) expressed a preference for IntelliJ's built-in data-flow analysis tooling.

5.6 Threats to Validity

We present an overview of the threats to the validity of our findings from analyzing the data collected from our controlled user study.

5.6.1 Internal Validity

Our user study was conducted entirely remotely, and participants were not observed as they worked on each task. This provided flexibility to our participants in that they were able to participate whenever it was most convenient for them. However, this made it difficult for us to directly observe our participants during the study, and impeded our ability to investigate questions or "think-aloud" comments posed by participants on-the-fly. The system in which our study was conducted might not have been familiar to our participants. The participants may have performed differently if the study was conducted in a system that they were familiar with, regardless of whether they used ReachHover or built-in data-flow analysis tooling.

Finally, we presented two reachability questions within two tasks in our user study. It is likely that developers encounter more types of reachability questions than can be reasonably captured in our user study within a limited period of time. We attempted to minimize this threat by using the results from our survey that investigated the most popular reachability developers ask in practice.

5.6.2 External Validity

Of our 20 participants, a majority (15) reported not being aware of IntelliJ's or any other IDE's data-flow analysis feature. Of the five who were aware of this feature, only two reported using these features on a regular basis. This suggests that our results might not be generalizable to populations who might already be familiar with data-flow analysis tooling. Although the reported

years of experience was varied among our participants in both the initial survey and the evaluative study, we are not able to concretely infer their subjective experiences, knowledge, and other personal factors. Consequently, further investigation is required to determine whether our results may be applicable to a general population of software developers.

5.6.3 Construct Validity

We considered both quantitative (i.e., correctness on each task, number of cursor jumps in the main editor) and qualitative (i.e., experiential questions) data in our study. In assessing how successful participants were in answering the reachability questions we posed, we calculated a correctness score (S) based on the number of correct and incorrect answers provided by each participant. This measure is useful in providing a concrete value that helps to quantify the performance of our participants using each tool. However, other measures such as accuracy and precision might be of further use in future investigations of our tool.

Main editor cursor jumps provide a measure that acts as a proxy for the amount of information-seeking that our participants might be performing as they attempt to complete each task. Although this measure does capture our participants' actions in the main code editor, it does not account for actions that they performed in other regions of the IDE. Other actions to consider might be the number of unique files viewed and additional explicit information-seeking actions such as the number of code search invocations and manual traversal of task-related control- or data-flow paths.

Chapter 6

Discussion

We have demonstrated that ReachHover can make answering certain data-flow reachability questions easier for developers. In this chapter, we demonstrate how ReachHover might be extended to support other kinds of reachability questions and present suggestions participants in our controlled user study had for future versions of the tool.

6.1 Asking and Answering Additional Reachability Questions

ReachHover enables developers to directly ask and answer two specific data-flow reachability questions that involve the creation and modification of data throughout a program. ReachHover could be extended to other questions that developers reported asking in the survey we conducted. For example, to answer Q8 ("Which paths in a program are executed given a certain value?"), we might pose a question such as "When is this path executed?" via a tooltip when a developer hovers over the branches of a conditional statement. In this case, we may exploit run-time information to provide a developer with a record of relevant variables and their values when a branch is executed. To answer Q7 ("Given two subtypes and their implementation of a given method, how do they handle data differently?") we may pose the question "How does this method modify data for each subtype?" when a developer inspects the top-level method declaration in the supertype. A developer might then be asked to select two subtypes for which the implementations of the method may be compared. The answer to this question might be expressed in the form of a diff of execution traces for each method that highlights similarities and differences in how they mutate or access data. This provides a possible starting point for how additional *compare*-type questions might be supported by ReachHover.

To answer Q6 ("Is deleting what appears to be unused code going to break anything?"), ReachHover might pose the question ("Is deleting this going to break anything?") when a developer highlights a block of code during a refactoring task. In this case, ReachHover might compute a diff of an execution trace of the relevant subset of the program before and after the block of code is

deleted, and present it to the developer should they chose to explore further.

More generally, to answer questions that involve dynamic run-time information, ReachHover could be extended to collect execution traces in the background during the execution of a test suite. To avoid the overhead that might be associated with storing execution traces for multiple test suite executions, ReachHover might execute a test suite on-demand (i.e., when a developer actually invokes ReachHover) and collect only the execution traces relevant to the subset of the program under investigation. In cases where test suite execution is not feasible, dynamic slicing [3] might be a viable alternative.

6.2 Tool Improvements

Not all of the participants in the study found ReachHover easy to use. We posed the following question at the end of our user study:

> **EQ2** "Do you have any additional thoughts or comments?"

Table 6.1: Codes assigned to each participant's response to EQ2.

Code	Description	Count
Task Confusion	User made a comment that suggested they were confused by the tasks in the study	2
Unsure or Misleading	User was unsure of the information provided by ReachHover or found some aspects of it misleading	3
Helpful	User found ReachHover helpful	1
Implementation-suggestion	User offered suggestions on current and future implementations of ReachHover	7

Table 6.1 presents the codes assigned to the responses to EQ2. We observe that three participants reported being unsure how to use one or both of the tools in the study. Seven of the participants did offer helpful suggestions to improve ReachHover. One participant remarked that they wanted ReachHover to be available in more locations in code:

> *"It would be useful if the hover was available at more locations (i.e. when hovering over an arbitrary identifier"* - P13

Another participant expressed interest in having the ability to directly query the code, much like the support already provided by Get Me Here:

> *"It would be nice to be able to use ReachHover to query the code"* - P8

Other suggestions were more cosmetic in nature, relating to code highlighting and the display of file names and paths in ReachHover's bottom component:

> *"I think having code highlighting (maybe also with the lines in the right-side gutter like are used for search and warnings) for identified related elements could potentially be helpful"* - P12

> *"I would prefer the full path (ie. the path + file name together) rather than spearating [sic] the file name on the left and the path on the right, as I have already grown accustomed to reading the fulll [sic] path elsewhere on the IDE + on git"* - P3

The comments provided by our participants generally alluded to minor updates to the plugin that does not require drastic re-implementations of functionality. One participant expressed interest in making ReachHover available in more locations in code (P13). This might be indicative of users generally finding ReachHover to be useful and wanting to use it in a wider range of locations that is currently supported. Support for this feature could be built by extending the code element detection functionality in ReachHover's front-end to identify arbitrary variables.

Another participant noted that it would be "nice to be able to use ReachHover to query the code" (P8). Actively querying code to answer reachability questions is a feature that is supported by existing tools such as REACHER and Get Me Here. Future work extending ReachHover could incorporate this functionality to enable querying code that is directly presented in the results window that is presented in the context of the code under investigation.

Chapter 7

Summary

Developers often ask reachability questions while performing their daily tasks [25]. Through a survey of 72 practicing software developers, we ranked how frequently nine reachability questions are asked in their work. As many of the highly-ranked questions pertained to data-flow, we chose to focus our investigation of direct user interface support for asking and answering reachability questions on the upstream "How was this value created?" and downstream "How is this value modified?" questions.

We built ReachHover, an open-source plugin for the IntelliJ IDEA IDE to support these questions and conducted a controlled user study comparing the use of ReachHover to the built-in data flow analysis support of the IDE. We found that ReachHover enabled participants in the study to answer an inter-file data-flow related reachability question more accurately and an intra-file question as accurately as the built-in IDE data-flow analysis support. Participants using ReachHover provided these answers with less context-shifting than with the built-in IDE support, a benefit commented upon by participants. ReachHover shows promise to enable developers to cope with reachability questions they encounter frequently and provides a framework for researchers to investigate direct support for other types of reachability questions.

Bibliography

[1] The State of Developer Ecosystem 2021, 2021. URL https://www.jetbrains.com/lp/devecosystem-2021/. Accessed December 4, 2021. → page 21

[2] Stack Overflow Developer Survey 2021, 2021. URL https://insights.stackoverflow.com/survey/2021#technology. Accessed December 4, 2021. → page 21

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256, jun 1990. ISSN 0362-1340. doi:10.1145/93548.93576. URL https://doi.org/10.1145/93548.93576. → page 42

[4] Apache Software Foundation. Apache Netbeans. https://netbeans.apache.org/community/index.html, 2022. [Online; accessed '01-September-2022]. → page 33

[5] M. Barnett, R. DeLine, A. Lal, and S. Qadeer. Get me here: Using verification tools to answer developer questions. Technical report, Technical Report MSR-TR-2014-10, 2014. → pages 2, 11, 13

[6] A. Begel, Y. P. Khoo, and T. Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 125–134, 2010. doi:10.1145/1806799.1806821. URL https://doi.org/10.1145/1806799.1806821. → page 5

[7] N. C. Bradley, T. Fritz, and R. Holmes. Sources of software development task friction. *Empirical Software Engineering*, 27(7):175, 2022. doi:10.1007/s10664-022-10187-6. URL https://doi.org/10.1007/s10664-022-10187-6. → page 6

[8] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola. Code bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 455–464, 2010. doi:10.1145/1806799.1806866. URL https://doi.org/10.1145/1806799.1806866. → page 13

[9] A. Bragdon, K. Rowan, J. Jacobsen, R. DeLine, and R. DeLIne. Debugger canvas: Industrial experience with the code bubbles paradigm. In *International Conference on Software Engineering*, June 2012. URL https://www.microsoft.com/en-us/research/publication/debugger-canvas-industrial-experience-with-the-code-bubbles-paradigm/. → page 13

[10] D. Campbell and M. Miller. Designing refactoring tools for developers. In *Proceedings of the 2nd Workshop on Refactoring Tools*, WRT '08, 2008. doi:10.1145/1636642.1636651. URL https://doi.org/10.1145/1636642.1636651. → page 7

[11] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code. 2021. → page 1

[12] B. de Alwis and G. Murphy. Using visual momentum to explain disorientation in the Eclipse IDE. In *2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'06)*, pages 51–54, 2006. doi:10.1109/VLHCC.2006.49. → pages 11, 22, 24, 34, 35

[13] Eclipse Foundation. Eclipse Desktop & Web IDEs. https://www.eclipse.org/ide/, 2022. [Online; accessed 02-Sept-2022]. → page 2

[14] P. Feiler, K. Sullivan, K. Wallnau, R. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, L. Northrop, and D. Schmidt. *Ultra-Large-Scale Systems: The Software Challenge of the Future.* Software Engineering Institute, Carnegie Mellon University, 2006. → page 5

[15] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 175–184, 2010. doi:10.1145/1806799.1806828. URL https://doi.org/10.1145/1806799.1806828. → page 5

[16] T. Grossman, G. Fitzmaurice, and R. Attar. A survey of software learnability: Metrics, methodologies and guidelines. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, page 649–658, 2009. doi:10.1145/1518701.1518803. URL https://doi.org/10.1145/1518701.1518803. → page 7

[17] Q. Guo and E. Agichtein. Ready to buy or just browsing? detecting web searcher goals from interaction data. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '10, page 130–137, 2010. doi:10.1145/1835449.1835473. URL https://doi.org/10.1145/1835449.1835473. → page 25

[18] J. Huang, R. W. White, G. Buscher, and K. Wang. Improving searcher models using mouse cursor activity. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, page 195–204, 2012. doi:10.1145/2348283.2348313. URL https://doi.org/10.1145/2348283.2348313. → page 25

[19] JetBrains s.r.o. IntelliJ IDEA: The Capable & Ergonomic Java IDE by JetBrains. https://www.jetbrains.com/idea/, 2022. [Online; accessed 02-Sept-2022]. → page 2

[20] JetBrains s.r.o. Visual Studio: IDE and Code Editor for Developers. https://visualstudio.microsoft.com, 2022. [Online; accessed 06-Sept-2022]. → page 8

[21] A. Ko, H. H. Aung, and B. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 126–135, 2005. doi:10.1109/ICSE.2005.1553555. → page 1

[22] A. J. Ko and B. A. Myers. Designing the Whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, page 151–158, 2004. doi:10.1145/985692.985712. URL https://doi.org/10.1145/985692.985712. → pages 2, 6, 7, 22

[23] A. J. Ko and B. A. Myers. Finding causes of program output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, page 1569–1578. Association for Computing Machinery, 2009. ISBN 9781605582467. doi:10.1145/1518701.1518942. URL https://doi.org/10.1145/1518701.1518942. → pages 7, 8, 22

[24] T. D. LaToza and B. A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, 2010. doi:10.1145/1937117.1937125. URL https://doi.org/10.1145/1937117.1937125. → pages 10, 16, 21

[25] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 185–194, 2010. doi:10.1145/1806799.1806829. → pages 1, 2, 9, 11, 16, 21, 44

[26] T. D. LaToza and B. A. Myers. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 117–124, 2011. → pages 2, 11, 12

[27] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 492–501, 2006. doi:10.1145/1134285.1134355. URL https://doi.org/10.1145/1134285.1134355. → page 1

[28] C. Maltzahn. Community help: Discovering tools and locating experts in a dynamic environment. In *Conference Companion on Human Factors in Computing Systems*, CHI '95, page 260–261, 1995. doi:10.1145/223355.223664. URL https://doi.org/10.1145/223355.223664. → page 7

[29] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz. The work life of developers: Activities, switches and perceived productivity. *IEEE Transactions on Software Engineering*, 43(12):1178–1193, 2017. doi:10.1109/TSE.2017.2656886. → page 1

[30] E. Murphy-Hill, R. Jiresal, and G. C. Murphy. Improving software developers' fluency by recommending development environment commands. FSE '12, 2012. doi:10.1145/2393596.2393645. URL https://doi.org/10.1145/2393596.2393645. → page 7

[31] C. V. Ramamoorthy and S.-B. F. Ho. Testing large software with automated software evaluation systems. *IEEE Transactions on Software Engineering*, SE-1(1):46–58, 1975. doi:10.1109/TSE.1975.6312819. → page 6

[32] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst.*, page 449–477, 1983. doi:10.1145/2166.357218. URL https://doi.org/10.1145/2166.357218. → page 1

[33] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *2012 34th International Conference on Software Engineering (ICSE)*, pages 255–265, 2012. doi:10.1109/ICSE.2012.6227188. → page 7

[34] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, page 23–34, 2006. ISBN 1595934685. doi:10.1145/1181775.1181779. URL https://doi.org/10.1145/1181775.1181779. → pages 5, 6

[35] A. L. Strauss and J. M. Corbin. *Basics of qualitative research: techniques and procedures for developing grounded theory.* 1998. → pages 34, 35

[36] W. Teitelman and L. Masinter. The Interlisp programming environment. *Computer*, 14(4): 25–33, 1981. → pages 5, 6

[37] M. A. Toleman and J. Welsh. Systematic evaluation of design choices for software development tools. *Software - Concepts & Tools*, 19(3):109–121, Mar 1998. doi:10.1007/s003780050014. URL https://doi.org/10.1007/s003780050014. → page 6

[38] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, page 107–119, 1991. → page 6

[39] P. Viriyakattiyaporn and G. C. Murphy. Challenges in the user interface design of an ide tool recommender. In *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, pages 104–107, 2009. doi:10.1109/CHASE.2009.5071421. → page 7

[40] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, page 439–449, 1981. → page 6

[41] J. Yoo and G. C. Murphy. ReachHover Plugin Source Code. https://doi.org/10.5281/zenodo.7038701, . [Online; accessed 03-Sept-2022]. → pages 3, 22

[42] J. Yoo and G. C. Murphy. ReachHover Formative Study Dataset. https://doi.org/10.5281/zenodo.7006242, . [Online; accessed 04-Sept-2022]. → page 16

[43] J. Yoo and G. C. Murphy. ReachHover User Study Dataset. https://doi.org/10.5281/zenodo.7008262, . [Online; accessed 03-Sept-2022]. → page 32

Appendix A

Supporting Materials

A.1 Survey Questions

We detail each of the software development questions that were presented to our survey participants. Questions 1 through 8 presented code excerpts that served to contextualize the question within a software development context. Scenario 9 did not include a code excerpt as it presented a program navigation question, which cannot be easily encompassed by a short code excerpt.

A.1.1 Survey Question 1

Question text: This figure (Figure A.1) shows a method. As a developer looks through a code base, they may be interested in tracing the values passed to and from methods. In this case, `items` is passed as a parameter to this method.

Question presented to participant:
"Where does this value (in this case, `items`) come from, and how is it formed?"

A.1.2 Survey Question 2

Question text: This figure (Figure A.2) represents a scenario where a developer is rewriting a method. The code on the left is the method before the developer made changes, while the one on the right is an updated version.

Question presented to participant:
"Did I introduce any unwanted changes in the new version of this code?"

Figure A.1: Survey Question 1

Are you interested in how this parameter is formed? E.g., the method calls and parts of the program that contribute to its creation.

```
public Map<Category, Double> computeTotalsByCategory(Map<ShopItem, Integer> items) {
    Map<Category, Double> totalsByCategory = new HashMap<>();
    items.forEach((item, quantity) -> {
        double cost = item.getCost() * quantity;
        totalsByCategory.put(item.getCategory(), cost);
    });
    return totalsByCategory;
}
```

Figure A.2: Survey Question 2

```
public Map<Category, Double> computeTotalsByCategory(Map<ShopItem, In
    Map<Category, Double> totalsByCategory = new HashMap<>();
    items.forEach((item, quantity) -> {
        double cost = item.getCost() * quantity;
        totalsByCategory.put(item.getCategory(), cost);
    });
    return totalsByCategory;
}
```

Before Changes

```
public Map<Category, Double> computeTotalsByCategory(Map<ShopItem,
    Map<Category, Double> totalsByCategory = new HashMap<>();
    for (Map.Entry<ShopItem, Integer> entry : items.entrySet()) {
        double cost = entry.getKey().getCost() * entry.getValue();
        totalsByCategory.put(entry.getKey().getCategory(), cost);
    }
    return totalsByCategory;
}
```

After Changes

A.1.3 Survey Question 3

Question text: This figure (Figure A.3) represents a scenario where an object is constructed within a method. The implementation of the constructor and the various sources of data used in creating the object, may not be immediately clear.

Question presented to participant:
"How is an instance of this class (in this case, `ShoppingCart`) created/initialized?"

Figure A.3: Survey Question 3

```
public ShoppingCart provision() {
    logger.info(msg: "Provisioning new cart");
    ShoppingCart cart = new ShoppingCart();
    this.register(cart);
    return cart;
}
```

Are you interested in how this object is created? E.g., the various data sources for this object, potential invariants for values that are used to construct this object.

A.1.4 Survey Question 4

Question text: This figure (Figure A.4) represents a scenario where some data is created (the **cart** object), and passed along as an argument to some methods. It may not be immediately clear how this data is being **accessed** (e.g., which fields of the **cart** object are being used).

Question presented to participant:
"Given some data (in this case, **cart**), which parts of it are **accessed** downstream?"

A.1.5 Survey Question 5

Question text: This figure (Figure A.5) represents a scenario where some data is created (the **cart** object), and passed along as an argument to some methods. It may not be immediately clear how this data is being **modified** (e.g., which fields of the **cart** object are being modified or changed).

Question presented to participant:
"Given some data (in this case, **cart**), which parts of it are **modified** downstream?"

A.1.6 Survey Question 6

Question text: This figure (Figure A.6) represents a scenario where a developer inspects a method. A popup appears that notifies them that the method being inspected is not used

Figure A.4: Survey Question 4

```
public ShoppingCart provision() throws CartValidationException {
    logger.info( msg: "Provisioning new cart");
    ShoppingCart cart = CartBuilder.build();
    if (this.hasValidState(cart)) {
        this.register(cart);
        return cart;
    }
    throw new CartValidationException("Provisioning a new cart failed");
}
```

Are you interested in how **cart** could be **accessed** in these method calls?

Figure A.5: Survey Question 5

```
public ShoppingCart provision() throws CartValidationException {
    logger.info( msg: "Provisioning new cart");
    ShoppingCart cart = CartBuilder.build();
    if (this.hasValidState(cart)) {
        this.register(cart);
        return cart;
    }
    throw new CartValidationException("Provisioning a new cart failed");
}
```

Are you interested in how **cart** could be **modified** in these method calls?

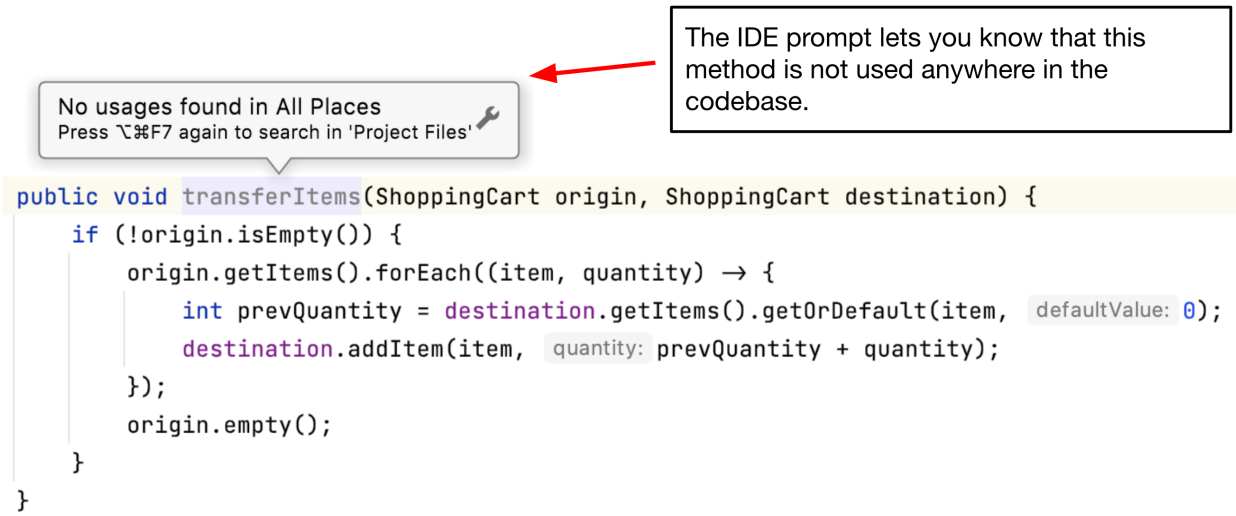
anywhere in the codebase.

Question presented to participant:
"Is deleting what appears to be unused code (in this case, **transferItems**) going to break anything?"

A.1.7 Survey Question 7

Question text: In this scenario, a developer may be inspecting an interface. They are interested in how the method **provision** may be implemented. A popup window shows that there are two subtypes of the interface that implement **provision**.

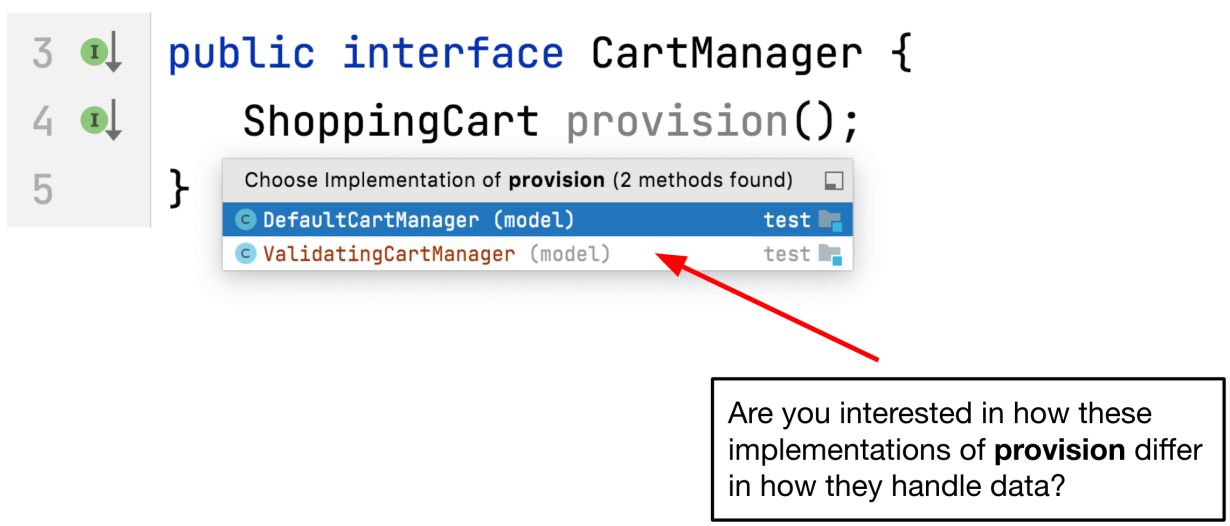
Figure A.6: Survey Question 6



Question presented to participant:

"Given two subtypes and their implementations of a common method (in this case, **provision**), how do they handle data differently?"

Figure A.7: Survey Question 7



A.1.8 Survey Question 8

Question text: This figure (Figure A.8) shows a method with a conditional statement (code within the branches are omitted). This is a scenario where control-flow is **dependent** on the value of data that a method is operating on. A similar scenario might be a switch statement.

Question presented to participant:
"Given some part of a program that depends on a value, which parts of it are executed or reachable?"

Figure A.8: Survey Question 8

```
public void resetCart(ShoppingCart cart) {
    this.logger.info( msg: "Resetting cart...");
    if (cart.hasStoredItems()) {
        cart.getItems().forEach((item, quantity) -> {
            this.cartRecord.put(item.getId(), quantity);
        });
        cart.empty();
    } else {
        this.registerEmptyCartReset();
    }
    this.logger.info( msg: "Reset operation complete");
}
```

Only one of these branches will be executed depending on the value of **`cart.hasStoredItems()`**.

A.1.9 Survey Question 9

Question text: A developer is inspecting a method and its call hierarchy. She begins to read it line-by-line, and click through other method calls that are within it (e.g., jump to method implementation). She continues this process until she eventually arrives at a location where she wants to stop.

Having navigated through a number of methods to arrive at this location, she wants to trace back to where she started to synthesize a high-level overview of the statements between the method where she started, and the location in code where she ended.

Question presented to participant:
"What does the control-flow look like between two locations in code?"