OUR TOOL: Navigating Program Flow in the IDE

Chris Brown, Justin Smith, Tyler Albert, and Emerson Murphy-Hill
Department of Computer Science
North Carolina State University
Raleigh, North Carolina 27606
Email: {dcbrow10, jssmit11, tralber2}@ncsu.edu, emerson@csc.ncsu.edu

***Abstract*—Program navigation is a critical task for software developers. Unfortunately, the current state-of-the-art tools do not adequately support developers in simultaneously navigating both control flow and data flow (i.e. program flow). To assist developers in effectively navigating program flow we designed and implemented a tool that leverages powerful program analysis techniques while maintaining low barriers to invocation. Our tool enables developers to systematically navigate program flow upstream and downstream within the Eclipse IDE. Based on a preliminary evaluation with 8 participants, our tool compares well to existing tools.**

I. Introduction

Modern software systems contain millions of lines of source code. As software grows in size and complexity, developers increasingly rely on tools to help them navigate the programs they create. Program navigation is a central task tied to many critical activities, including exploring new code bases, debugging, and assessing security vulnerabilities.

While navigating programs, developers ask questions about control flow and data flow throughout the program [1], [2]. We will refer to these two concepts together as *program flow*. Developers are interested in navigating program flow to trace how data is modified across multiple method invocations.

Integrated development environments (IDEs) present code linearly in the order methods are defined. However, successful developers do not navigate source code linearly (line by line starting at the top of the file). Instead, they methodically navigate the code's hierarchical semantic structures [3]. To resolve this conflict and realize their ideal navigation strategies, developers rely on program navigation tools.

In this work we will design, implement, and evaluate a program navigation tool. To address the limitations of existing program navigation tools, our tool will embody five key design principles (Section II). We will implement our tool as a plugin to the Eclipse IDE.

This paper makes the following contributions:

II. Design Principles and Implementation

In this section we describe the design principles that we used to shape OUR TOOL. We derived these design principles by examining existing program navigation tools.

Powerful Program Analysis - By leveraging powerful program analysis techniques, navigation tools can provide more accurate information. For example, by analyzing abstract syntax trees (ASTs) and call graphs, tools can make proper references to variables and methods. Simple textual analysis may lead to inaccurate results, especially when programs include inheritance and duplicate variable names.

Low Barriers to Invocation - Barriers to invocation may inhibit adoption. As developers may wish to navigate multiple program paths concurrently, repetitively invoking the tool may be cumbersome, especially if barriers are high.

Full Program Navigation - Developers are not only interested in traversing programs' call graph, but also how data flows through the call graph. To do so, developers must inspect the relationship between methods as well as the methods themselves. Often the methods of interest span across multiple source files. Furthermore, program navigation tools should support this traversal both upstream and downstream. That is, tools should highlight variable assignments and also subsequent variable uses.

In Situ Results - Switching between views in the IDE can cause disorientation [4]. As developers navigate through code, navigation tools should present their results in that context. When navigation tools present results outside the code, developers are burdened with the cognitive load of translating those results back to the code.

III. Background

There are a variety of tools available to help developers explore and navigate code.

A. Production Tools

Production tools are represented by plugins in integrated development environments that provide detailed information and analysis on variables and methods in the source code. Examples of these types of tools include Call Hierarchy and Find References in Eclipse [5] and Analyze Data Flow To/From Here and Analyze Dependencies in IntelliJ [6]. These tools provide powerful program analysis for users to navigate throughout the entirety of a project, however they differ from our tool in that they require multiple steps to start the tool, need repeated invocations of the plugin to track multiple paths in the code, and force users to switch between the text editor of the IDE and a new panel displaying the results.

TABLE I
DESIGN PRINCIPLES

Tool Category	Powerful Program Analysis	Low Barriers	Full Program Navigation	In Situ Results
Production	Yes	No	Yes	No
Visualization	Yes	No	Yes	No
IDE Integrated	No	Yes	Sometimes	Yes

B. Visualization Tools

Code navigation tools that provide users with a graphical representation of the results are visualization tools. Some examples of these types of tools include Code Bubbles [7], Code Canvas [8], Code Surfer [9], Dora [10], Reacher [11], Relo [12], Whyline [13], and many more research tools. These works provide various views of control flow graphs, class and UML-like diagrams, call graphs, and other images to describe the relationship between different variables or functions within the code. OUR TOOL is different because we want to provide minimal steps to invoke the tool and present the information inside of the editor while also exploring the full project's source code and providing powerful data analysis.

C. IDE Integrated Tools

IDE Integrated tools are navigation tools that are consolidated into the editor. Some examples of these include Mark Occurrences [14] and Open Declaration in the Eclipse IDE. These approaches are similar to our tool in that they do not present the results in a new view or panel, but in some cases they may not provide a detailed enough analysis to explore the entire program or provide extra unnecessary information, for example Mark Occurrences only highlighting the instances of data within a class in addition to highlighting the value in a comment. These can require repeated work to start the tool, such as Open Declaration requiring the user to right-click or enter a keyboard shortcut for each method declaration the user wants to open.

IV. OUR TOOL

OUR TOOL was designed to realize all of the principles described in Section II. We implemented OUR TOOL as a plugin to the Eclipse IDE [5]. We chose Eclipse because of its popularity and extensibility. Eclipse is one of the most widely used open source IDEs for Java development and it provides many extension points for plugins.

Figure 1 depicts OUR TOOL invoked on a variable participants were asked to inspect as part of our evaluation. To visualize how a programmer would interact with OUR TOOL, consider the following scenario:

You are concerned that users could modify the value of `fileName` before it gets passed into `getQueries`. Because it is the variable you are concerned with, you click on `fileName` (A). To help you locate where the variable is modified and referenced, OUR TOOL highlights proper occurrences of that variable in the code. Since `fileName` is a formal parameter to `getQueries`, any method calling `getQueries` could modify `fileName`. Those methods reside in other class files, so OUR TOOL provides links to their locations (B1). Rather than move your mouse up to the top of the editor window, you click on `getQueries` (B2), which conveniently links to the first call site. OUR TOOL opens `createTables` and highlights the location in that method where fileName is passed to `getQueries`.

TABLE II
PARTICIPANT DEMOGRAPHICS

Participant	Industry Experience (years)	Java Experience (years)	Previous Eclipse Use
P1	9	5	Yes
P2	0	6	Yes
P3	3	2	No
P4	5	0	No
P5	12	10	Yes
P6	0	3.5	Yes
P7	1	9	Yes
P8	5.5	3.5	Yes

V. PRELIMINARY EVALUATION

We performed a preliminary evaluation of OUR TOOL with eight programmers performing two code navigation tasks. Our goals in this study were to get feedback on the usability of our tool and evaluate whether our approach shows promise in enabling developers to effectively navigate program flow.

All participants were graduate students at the time of the study and on average had 5 years of professional programming experience; Table II provides additional information about each participant. We recruited participants using a convenience sampling approach.

Each participant used OUR TOOL for one task and the Eclipse's tools (call hierarchy, mark occurrences, and open declaration) for the second task. To control for learning and fatigue effects, we used a 2x2 latin square design which permuted the order participants received each tool and performed each task. Accordingly, each participant was assigned to one of four groups – one group for each pair of conditions. Before the study, we asked participants to report whether they were familiar with the Eclipse IDE. We used this information to balance Eclipse novices across groups.

A. Tasks

We analyzed the data from our previous study [2], which included two tasks that required program flow navigation. In the previous study, developers expressed a willingness to navigate the program, but used sub-optimal strategies to do so. Because the challenges participants faced in [2] partially inspired the development of OUR TOOL, we include the same two tasks in this study. For Task 1 we asked participants to tell us whether a method ever receives user-provided input. For Task 2 we asked participants to tell us whether a form field is validated before being sent to the database. To ensure all participants had a baseline familiarity with both tools, we trained participants on the appropriate tools preceding each task. To evaluate the effectiveness of the navigation

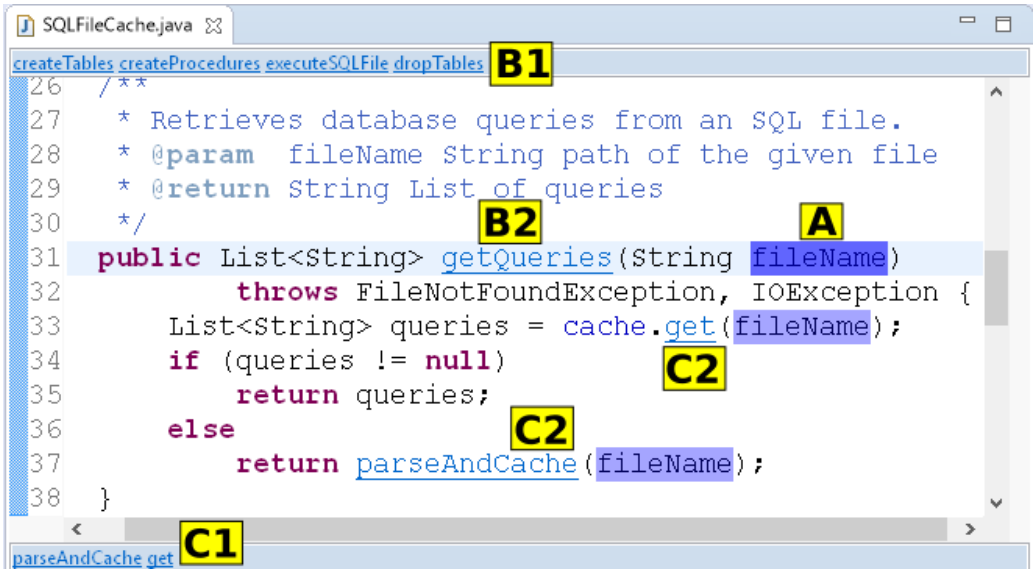

Fig. 1. Code view in the Eclipse IDE with OUR TOOL invoked on Task 1

tools rather than participants' familiarity with a particular code base, we asked participants to navigate code they had not previously contributed to. Because think aloud protocols distort the amount of time required to complete tasks, we did not interrupt or prompt participants until after they had completed the tasks.

B. Usability Evaluation

To evaluate the usability of OUR TOOL, we administered an adapted version of the Post-Study System Usability Questionnaire (PSSUQ) [15] after participants had completed both tasks. We modified the questionnaire by replacing "this system" with "this tool" and asked questions from the System Quality and Interface Quality categories. We asked 10 questions; participants responded on a 7-point Likert scale from "Strongly Disagree" to "Strongly Agree." We also asked participants open-ended questions based on applicable categories from Nielsen's usability heuristics [16] To capture participant's experiences that these two metrics overlooked, two of the authors independently examined each audio/video recording and recorded memos.

VI. Results

A. Quant stuff...

Graph with average time to complete each task split by tool and no tool different colors. Time to first method for Task 1 split by tool and no tool.

Time to first method

Correctness

B. Qualitative Results

Satisfied with easy to use and Simple to use and Easy to learn Participants responded positively

Responded low for complete tasks quickly (performance issues and disorientation)

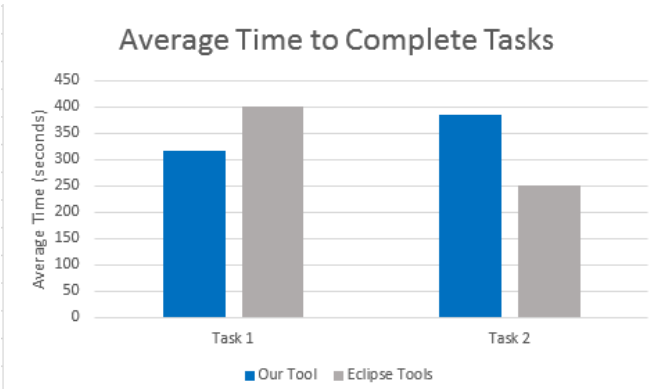

Fig. 2. Average time to complete each task with and without OUR TOOL

All the functions I expected (Missing tracability and mark bars, ...)

With our tool: Task 1 Faster than other tool, less correct. Task 2 slower, but more correct.

VII. Discussion - Scenarios

List scenarios when our tool worked and when it didn't

A. Wins

Getting started. Low barriers to invocation

Linear Naviagtion When participants navigated linearly in one direction (as in Task 1). Our tool was most effective. Few branches

B. Losses

Why were they slower and less accurate with our tool? Second task: Lots of confounding variables meant participants spent time switching between variables. Requires more knowledge about iTrust. Even though we signposted all the areas that (required) knowledge about iTrust...

Using our tool for task 1, everyone located a hard-coded string. But the two participants that answered incorrectly (misinterreted the information)

C. Systematic Evaluation

Program navigation tools should help developers keep track of where they have been and where they are going. Especially while attempting to resolve complex defects, developers may want to thoroughly explore all program paths. Their navigation tools should help them keep track of their progress.

D. Design Implications

Participants were taken to new locations but couldn't backtrack. Back buttons exist, but most didn't use and they didn't help the person who did use them... Points to an inherent limitation of our minimalistic approach. People would be more oriented with a graph or more persistent breadcrumbs...
Overwhelmed when the call graph had a high branching factor. Participants quickly navigated up single paths as in Task 1 When call paths branched in many directions our tool could reccomend more full featured tools.
Couldn't keep track of where they had been. See systematic evaluation. Links could turn purple and get moved to the end of the list.
Performance. Whenever the user clicks on a new variable OUR TOOL has to search through the entire project for locations where the method is declared or invoked. We observed that participants navigated within one file more often than between files. To improve the performance of our tool, we could optimize the search to return local results before returning global search results. Participants also repeated invocation on the same variables (we could do some caching)

VIII. Limitations

IX. Conclusion

Acknowledgment

The authors would like to thank...

References

[1] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Evaluation and Usability of Programming Languages and Tools*. ACM, 2010, p. 8.
[2] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford, "Questions developers ask while diagnosing potential security vulnerabilities with static analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015, pp. 248–259. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786812
[3] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: An exploratory study," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 889–903, 2004.
[4] B. de Alwis and G. C. Murphy, "Using visual momentum to explain disorientation in the eclipse ide," in *Proceedings of the Visual Languages and Human-Centric Computing*, ser. VLHCC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 51–54. [Online]. Available: http://dx.doi.org.prox.lib.ncsu.edu/10.1109/VLHCC.2006.49
[5] "Eclipse," https://eclipse.org/.
[6] "Intellij," https://www.jetbrains.com/idea/.
[7] "Code bubbles," http://www.andrewbragdon.com/codebubbles_site.asp.
[8] "Code canvas," http://research.microsoft.com/en-us/projects/codecanvas/.
[9] "Code surfer," http://www.grammatech.com/products/codesurfer.
[10] E. Hill, L. Pollock, and K. Vijay-Shanker, "Exploring the neighborhood with dora to expedite software maintenance," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 14–23.
[11] "Reacher," https://softvis.wordpress.com/2011/11/25/visualizing-call-graphs/.
[12] "Relo," http://relo.csail.mit.edu/.
[13] "Whyline," http://www.cs.cmu.edu/ NatProg/whyline.html.
[14] "Mark occurrences," http://www.eclipse.org/pdt/help/html/mark_occurrences.htm.
[15] J. R. Lewis, "Ibm computer usability satisfaction questionnaires: Psychometric evaluation and instructions for use," *International Journal of Human-Computer Interaction*, pp. 57–78, 1995.
[16] J. Nielsen, "Finding usability problems through heuristic evaluation," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '92. New York, NY, USA: ACM, 1992, pp. 373–380. [Online]. Available: http://doi.acm.org/10.1145/142750.142834