Flower: Navigating Program Flow in the IDE

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

***Abstract*—Program navigation is a critical task for software developers. State-of-the-art tools have been shown to support effective program navigation strategies, and do so by adding widgets, secondary views, and visualizations to the screen. In this work, we build on prior work by exploring what types of navigation can be supported with relatively few interface elements. To that end, we designed and implemented a prototype tool, named Flower, that supports structural program navigation while maintaining a minimalistic interface. Flower enables developers to simultaneously navigate control flow and data flow within the Eclipse Integrated Development Environment. Based on a preliminary evaluation with eight programmers, Flower succeeds when call graphs contained relatively few branches, but was strained by complex program structures.**

I. Introduction

Although integrated development environments (IDEs) present code linearly in the order methods are defined in a file, successful developers do not navigate source code line by line starting at the top of the file. Instead, they methodically navigate the code's hierarchical semantic structures [1]. While navigating programs, developers need information about control flow and data flow throughout the program [2], [3]. We will jointly refer to these two concepts as *program flow*.

To realize their ideal program flow navigation strategies, developers rely on navigation tools that expose the links between sometimes distant locations in the source code. Many existing tools do so by displaying call graph visualizations or adding views to the screen. In general, evaluations have demonstrated the effectiveness of such tools [4]–[7].

However, these tools rely on cumbersome user interface widgets, views, and visualizations that occupy valuable screen real estate and might induce disorientation via thrashing [8]. In this work, we explore whether relatively few interface elements can enable similarly effective program navigation. We present a prototype tool, *Flower* (pronounced *flow-er*), that instantiates this minimalistic approach to program navigation. *Flower* leverages accurate program analysis techniques, avoids cluttering the IDE with superfluous interface elements, and presents its results integrated within the code.

Consider the following motivating example involving Charlie, a professional Java developer. Charlie is fictional, but this story is based on the experiences of real developers as described in our previous study [3]. While maintaining some old code, Charlie notices a static analysis warning: "This variable contains user-provided data. If it is used in a sensitive context before being sanitized, the code could be vulnerable."

Charlie sets out to determine if the variable is sanitized on all paths leading to the sensitive context. First, Charlie searches for the variable's name. Unfortunately, this search returns results mostly from comments and other documentation. Charlie vaguely recalls using an Eclipse tool to help trace control flow through a program, but is unsure how to invoke it or whether that tool also traces data flow. Charlie looks through various menus, tries out a few tools, but cannot locate the right tool. Turning back to the code, Charlie begins scrolling through the current file, unknowingly using Eclipse's *Mark Occurrences* tool while scanning for uses of the variable. After inspecting all the occurrences in the current file, Charlie now searches for methods that take the variable as input. Charlie stumbles upon Eclipse's *Call Hierarchy* tool, which seems helpful, but is unable to specify a variable when invoking the tool.

Undeterred, Charlie finds a method that takes the variable as a parameter and invokes *Call Hierarchy* on that method. Charlie's attention oscillates between the *Call Hierarchy* view and the code. Charlie uses the *Call Hierarchy* view to navigate chains of method calls and checks the code in each method for sanitization. Charlie repeats this process for each call site in the current file. Unsure of whether the variable gets sanitized along all paths, Charlie decides to ignore the warning. Two months later, an attacker exploits the vulnerability.

This work contributes an understanding of the essence of program navigation, through the lens of a minimalistic tool, *Flower*. This prototype tool addresses many of the issues that Charlie faced by implementing four design principles (Section II). For instance, *Flower* is easily invoked and enables developers to simultaneously trace data flow and control flow within the code view. We evaluated *Flower* to identify the types of tasks it effectively supports. Our preliminary findings failed to identify significant differences in task completion time or correctness between *Flower* and more complex navigation tools within Eclipse. This suggests that the simple affordances *Flower* implements may sufficiently aid developers.

II. Design Principles

In this section we describe the guiding design principles behind *Flower*. We derived these design principles from the information needs described in a study by Smith and colleagues [3] and by examining existing program navigation tools.

978-1-5386-0443-4/17/$31.00

LOW BARRIERS TO INVOCATION — Some tools are easier to invoke than others. Take, for example, Charlie's case. Charlie easily invoked *Mark Occurrences*, but initially struggled to locate and invoke *Call Hierarchy*. Tools with high barriers to invocation require users to sift through menus and include unintuitive widgets. Barriers to invocation inhibit adoption [9]. As developers navigate multiple program paths concurrently, difficulties repetitively invoking tools compound, especially if barriers are high. Once initially configured, *Flower* is automatically invoked as the user navigates.

ACCURACY — Simple textual analysis may lead to inaccurate results in many scenarios. For example, such analysis fails when programs include duplicated variable names that refer to different variables in different scopes [10]. Textual analysis also falls short when programs contain inheritance and when variable names are included in comments, documentation, or other syntactically irrelevant locations, as in Charlie's case. By leveraging program analysis techniques, navigation tools can provide more accurate information than simple textual analysis. *Flower* analyzes abstract syntax trees (ASTs) and call graphs to make accurate references to relevant variables and methods.

FULL PROGRAM NAVIGATION — Developers are not only interested in traversing programs' call graphs, but also how data flows through the call graph [3]. To simultaneously navigate the flow of data through call graphs, developers must inspect the relationship between methods, as well as the methods themselves. Often the methods of interest span across multiple source files. For Charlie, *Mark Occurrences* helped with navigation within a single file, but fell short while being used to inspect methods in other 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. *Flower* allows for upstream and downstream navigation across files.

IN SITU NAVIGATION — Switching between views in "bento box" style IDEs [11] can cause disorientation [8]. As developers navigate through code, navigation tools should present their results in that context. This was a problem for Charlie, who had to constantly switch between the *Call Hierarchy* view and the code view. When navigation tools present results outside the code, developers are burdened with the cognitive load of translating those results back to the code. *Flower* displays its results within the code to enable in situ navigation.

III. RELATED WORK

The need for better navigation tools has been well reported [12]. Here we discuss some of the existing tools that help developers navigate code. We also relate the existing tools back to Design Principles described in Section II.

Many modern IDEs provide tools that help developers navigate source code. For example, Eclipse [13] includes *Call Hierarchy* and *Find References*. When users invoke either of these tools, Eclipse opens a new view to display the results. For *Call Hierarchy*, this view contains a list of the selected method's callers and callees, whereas *Find References* only lists callers. Similarly, IntelliJ [14] provides *Analyze Data Flow*, which also displays its results in an external view. These tools enable users to navigate through their code with ACCURACY. However, unlike *Flower*, the separate views do not enable IN SITU NAVIGATION.

There are also several tools that reside strictly within the code editor, enabling a form of IN SITU NAVIGATION. Two examples of these tools are Eclipse's *Mark Occurrences* and *Open Declaration*. Eclipse automatically invokes *Mark Occurrences* whenever a user clicks on a variable or method name in the code. The tool then highlights occurrences of that element elsewhere in the current file. *Mark Occurrences* epitomizes LOW BARRIERS TO INVOCATION. These tools are similar to *Flower* in that they display the results within the editor rather than a separate view or panel. However, they do not enable FULL PROGRAM NAVIGATION.

Many other tools help developers navigate code by representing the code graphically and allowing developers to navigate those graphs [4]–[6], [15]–[18]. These works provide various views of control flow graphs, class and UML-like diagrams, trees, call graphs, and other images to describe the hierarchy and relationship between different variables or functions within the code. These tools utilize program analysis to generate visualizations with ACCURACY. Moreover, most of these tools implement some aspects of FULL PROGRAM NAVIGATION. *Flower* differs from these tools in the way it presents results (IN SITU NAVIGATION) and because it has LOW BARRIERS TO INVOCATION.

IV. FLOWER

Flower was designed to realize all of the principles described in Section II. We implemented *Flower* as a plugin to the Eclipse IDE [13]. We chose Eclipse because it is one of the most widely used open source IDEs for Java development and it provides many extension points for plugins. *Flower* extends *Call Hierarchy*, using its search functionality to track the flow of data across methods. Leveraging Eclipse's incremental compiler and JDT core, *Flower* also implements several AST visitors to detect proper and up-to-date variable references.

When active, *Flower* makes two modifications to Eclipse's user interface. First, it highlights on-screen references to selected variables. Second, when references to the selected variable appear off-screen — either elsewhere in the current method or in other methods — *Flower* adds links to those locations above and below the code view as well as to the code itself. Figure 1 depicts *Flower* invoked on a variable participants were asked to inspect as part of our evaluation. We also provide a video[1] and virtual machine image[2] demonstrating *Flower* online. To visualize how a programmer would interact with *Flower*, consider the following scenario, which corresponds with Figure 1:

[1]Screen cast of *Flower*: https://figshare.com/s/39eadd74502a014ae018
[2]VM containing *Flower*: https://figshare.com/s/76aa260f21cf4233dc1c

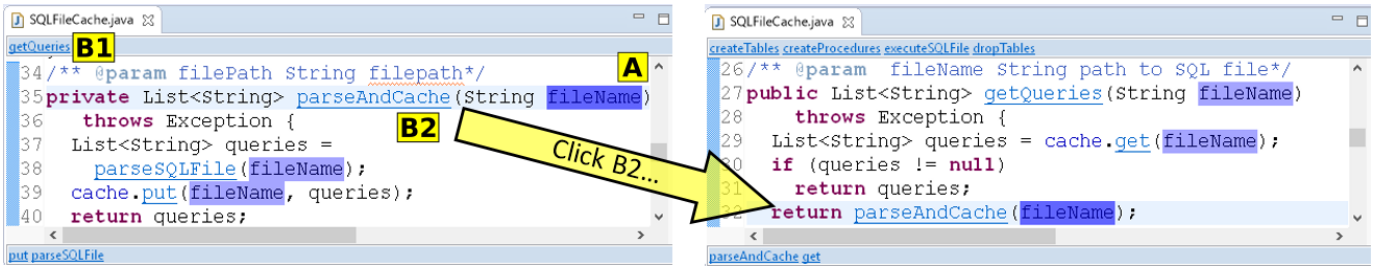


Fig. 1: Eclipse IDE with *Flower* invoked during Task 1. The left shows the tool running in the method `parseAndCache`. When the user clicks on the link in the editor (B1 or B2), they are taken to `getQueries`, which is a caller of `parseAndCache`.

Suppose you are a programmer and you notice that by tampering with the value of the `fileName` variable, malicious users could gain access to sensitive information in the database. You want to determine whether users can modify `fileName` before it gets passed into `parseAndCache`. First, you click on `fileName` (A). Much like *Mark Occurrences*, *Flower* is automatically invoked. To help you locate where the variable is modified and referenced, *Flower* highlights occurrences of that variable in the code. Since `fileName` is a formal parameter to `parseAndCache`, any method calling `parseAndCache` could modify `fileName`. Those methods reside in other files, so *Flower* provides links to their locations (B1). Rather than move your mouse up to the top of the editor window, you click on `parseAndCache` (B2), which conveniently links to the first call site, `getQueries`. *Flower* opens `getQueries` and highlights the location in that method where `fileName` passed to `parseAndCache` and shows that the value is passed to `getQueries` from `createTables`, `createProcedures`, `executeSQLFile`, and `dropTables` as seen in Fig. 1.

V. Preliminary Evaluation

We performed a preliminary evaluation of *Flower* with eight programmers performing two code navigation tasks.[3] As a baseline, we compare our tool against the existing suite of Eclipse tools (*Open Declaration*, *Mark Occurrences*, and *Call Hierarchy*). Our goals in this study were to (a) determine what types of navigation activities *Flower* effectively supports and fails to support and (b) get feedback on the usability of *Flower*. To answer (a), we measured how quickly and accurately participants completed different activities. The remainder of this section describes our participants, study design, and task selection, then concludes with our approach to answer (b).

All participants were graduate students at the time of the study with a mean of five years of professional programming experience. We recruited participants using a convenience sampling approach. Each participant used *Flower* for one task and Eclipse's tools for the other task. To control for learning and fatigue effects, we permuted the order participants received each tool and performed each task. Before the study, we asked participants to report whether they were familiar with the Eclipse IDE. This information helped us balance Eclipse novices across groups.

We based our tasks in this study on two tasks (Tasks 1 and 3) from a prior study [3]. For these tasks, participants navigated across several classes in an open source Java medical records application containing over 50,000 lines of code. In the previous study, participants used a think-aloud protocol to, among other things, describe their program navigation strategies. Here, we did not interrupt, prompt, or ask participants to think aloud until after they had completed the tasks, as to not distort their task completion time.

The two tasks we chose are complementary in that Task 1 required participants to navigate up the call graph, inspecting the callers of the initial method. On the other hand, Task 2 required participants to inspect the methods called by the initial method. 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, we trained participants on the appropriate tools preceding each task. To evaluate the effectiveness of the navigation tools rather than participants' familiarity with a particular code base, we asked participants to navigate code they had not previously contributed to.

We collected screen and audio recordings of each participant's tasks. To evaluate the usability of *Flower*, we administered an adapted version of the Post-Study System Usability Questionnaire (PSSUQ) [19]. We modified the questionnaire by replacing "this system" with "this tool" and asked questions from the System Quality and Interface Quality categories. To prompt discussion about the usability of *Flower*, we also asked participants open-ended questions based on applicable categories from Nielsen's usability heuristics [20].

VI. Results

Here we present the results of our preliminary evaluation. We tested whether participants performed differently — in terms of task completion time and correctness — using the full suite of Eclipse tools, compared with Flower. We tested overall task completion time using two-tailed, unpaired, two-sample, t-tests and task correctness using chi-squared tests. The tests failed to identify a significant difference in completion time and correctness between *Flower* and the Eclipse suite of tools.

[3]Study materials at: https://figshare.com/s/49edec2b4810fbf5b2a0

In the remainder of this section, we present our observations about the types of activities *Flower* seemed to support. We organize these results thematically into three topics:

Approachable Interface: Compared to other PSSUQ questions, participants responded most positively to the three questions about *Flower's* simplicity, how easy it was to use, and how easy it was to learn. These responses seem to indicate that participants were most enthusiastic about *Flower's* minimalistic interface. Participants reiterated these sentiments in their responses to the open-ended questions. For the most part, participants felt it was easy to remember how to use *Flower* and that it featured a "consistent interface."

Branchless Navigation: For Task 1, all participants correctly navigated up the call graph and did so faster with *Flower* compared to the Eclipse tools. With *Flower*, participants' mean completion time was 276 seconds, compared to 402 seconds with the Eclipse tools.

The first two steps in this task involved navigating a portion of the call graph that did not include any branches. In other words, participants started in the `parseSQLFile` method, which was only called in one location, `parseAndCache`. The `parseAndCache` method was only called in one method `getQueries`. Participants equipped with our tool were strictly faster in navigating the first step up this branchless chain. The mean times for participants to reach `parseAndCache` with *Flower* and the Eclipse tools were 8 seconds and 44 seconds, respectively. These two results suggest that *Flower* successfully adhered to the LOW BARRIERS TO INVOCATION design principle.

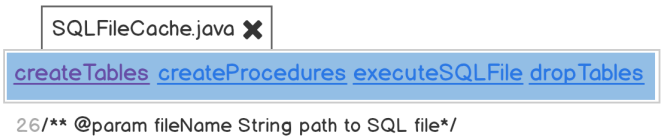
Branching and Backtracking: For Task 2, participants were more accurate with *Flower*. Two participants (P3 and P6) navigated to the correct validation method with *Flower*. Only one (P8) did so with the Eclipse tools.

However, the mean completion time for Task 2 with *Flower* was higher (385 seconds) compared to the Eclipse tools (251 seconds). Additionally, participants who used *Flower* for Task 2 scored the tool lower on the PSSUQ than those who used it for Task 1. Based on open-ended responses and our observations of participants, we provide one likely explanation for this deficiency. Participants were required to sift through more variable references and method calls to complete Task 2. In navigating this more complex program structure, when participants took missteps they found it difficult to backtrack. To a lesser extent, we observed this same difficulty during Task 1. After navigating through several chains of method invocations with *Flower*, P7 felt like she had reached a "dead end" and was unsure of how to navigate back to where she came from. Similarly, after reaching a top-level method, P1 asked, "How can I return back to where I came from?"

VII. DISCUSSION

A. Systematic Navigation

Participants completed simple navigation tasks quickly and accurately with *Flower*, perhaps due to its minimalistic interface. However, when the task required participants to navigate more complex semantic structures, participants demanded features that would allow them to navigate more systematically. Many existing tools support systematic exploration through the use of secondary views containing either hierarchically structured lists of methods (e.g. *Call Hierarchy* and *Analyze Data Flow*) or call graph visualizations (e.g. *Reacher [4]*). In keeping with *Flower's* minimalistic design and trying to preserve *Flower's* LOW BARRIERS TO INVOCATION, we envision several design changes that might enable *Flower* to support more systematic navigation. Similar to *Whyline* [5], *Flower* could use animation to transition more smoothly between locations, perhaps giving users a sense of naturally moving through the code. By tracking developers' progress, *Flower* could also display already-visited locations differently than unexplored methods either positionally or using colors:



B. Synergistic Tools

Navigation without tool support can be frustrating and unfruitful. However, full-featured navigation tools might be too cumbersome for simple navigation tasks and too complex for unfamiliar users. We envision *Flower* serving as a stepping stone to more sophisticated navigation tools. The design principles of LOW BARRIERS TO INVOCATION and IN SITU NAVIGATION enable users to quickly begin navigating using *Flower*. We imagine that *Flower* could detect when users reach "dead ends" or code that contains many complex branches. Upon detecting one of these situations, *Flower* could facilitate the user's transition to a more heavy-weight tool by either recommending or automatically invoking a tool that features additional navigation visualizations.

VIII. STUDY LIMITATIONS

Our study had several limitations. Due to the preliminary nature of this study, we recruited relatively few participants and only had them perform two tasks. Although our study materials do not indicate we created *Flower*, some participants may have deduced it was our tool. As a result, participants may have inflated their positive responses to the PSSUQ due to social desirability bias. Accordingly, we focus on participants' relative responses rather than their absolute values.

IX. CONCLUSION

We presented a new tool, *Flower*, that helps developers navigate program flow with its minimalistic interface. *Flower* explores the void between tool-less navigation strategies and cumbersome flow visualization tools. Based on our preliminary evaluation, *Flower* was most effective when developers wished to navigate program structures with few branches.

ACKNOWLEDGMENTS

We graciously thank our study participants for their time. We also thank Tyler Albert, Shubham Goyal, and members of the Developer Liberation Front for their thoughtful input. This material is based upon work supported by the National Science Foundation under grant number 1318323.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] T. D. LaToza and B. A. Myers, "Visualizing call graphs," in *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, Sept 2011, pp. 117–124.

[5] 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*, ser. CHI '04. New York, NY, USA: ACM, 2004, pp. 151–158.

[6] V. Sinha, D. Karger, and R. Miller, "Relo: Helping users manage context during interactive exploratory visualization of large codebases," in *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, ser. eclipse '05. New York, NY, USA: ACM, 2005, pp. 21–25.

[7] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers, "Stacksplorer: Call graph navigation helps increasing code maintenance efficiency," in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '11. New York, NY, USA: ACM, 2011, pp. 217–224.

[8] 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.

[9] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 35th ACM/IEEE International Conference on Software Engineering*, ser. ICSE '13. IEEE, 2013, pp. 672–681.

[10] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson, "Towards understanding programs through wear-based filtering," in *Proceedings of the 2005 ACM Symposium on Software Visualization*, ser. SoftVis '05. New York, NY, USA: ACM, 2005, pp. 183–192.

[11] R. DeLine and K. Rowan, "Code canvas: Zooming towards better development environments," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 207–210.

[12] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, Dec 2006.

[13] "Eclipse," https://eclipse.org/.

[14] "IntelliJ," https://www.jetbrains.com/idea/.

[15] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., "Code bubbles: A working set-based interface for code understanding and maintenance," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: ACM, 2010, pp. 2503–2512.

[16] R. DeLine and K. Rowan, "Code canvas: Zooming towards better development environments," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 207–210.

[17] P. Anderson and M. Zarins, "The codesurfer software understanding platform," in *Proceedings of the 13th International Workshop on Program Comprehension*, ser. IWPC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 147–148.

[18] 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*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 14–23.

[19] J. R. Lewis, "IBM computer usability satisfaction questionnaires: Psychometric evaluation and instructions for use," *International Journal of Human-Computer Interaction*, pp. 57–78, 1995.

[20] 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.