Flower: 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. The current state-of-the-art tools support program navigation by leaning heavily on visualizations and cumbersome interface widgets. To assist developers in effectively navigating program flow, we designed and implemented a new tool that leverages powerful program analysis techniques while maintaining a minimalistic interface. Our tool enables developers to simultaneously navigate control flow and data flow within the Eclipse Integrated Development Environment (IDE). Based on a preliminary evaluation with 8 programmers, our tool succeeds when call graphs contained relatively few branches, but was strained by complex program structures.**

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. A software testing study described a wide variety of tools developers use to validate their code, but also found that inadequate testing and debugging tools contributed to poor quality software in production [2].

Finding, comprehending, and fixing bugs has become the most common software activity, consuming 70-80% of software engineers' time [1]. Program navigation can help improve developers' program comprehension, which Brooks argues is vital because this type of domain-specific knowledge plays a key role in nearly every software tasks including development, code reviews, debugging, code maintenance, testing, and more [3].

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 [4]. While navigating programs, developers ask questions about control flow and data flow throughout the program [5], [6]. We will refer to these two concepts together 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 the screen. In general, evaluations have demonstrated the effectiveness of such tools [1], [7]–[9].

However, these tools rely on cumbersome user interface widgets that new users might struggle to understand and occupy valuable screen real estate. We present a tool, Flower (pronounced flow-er), that represents a minimalistic approach to program navigation. Flower leverages powerful program analysis techniques, leans on relatively few cumbersome interface elements, and presents its results integrated within the code.

Consider the following motivating example involving Hillary, a professional Java developer at a large software company. While maintaining some old code, Hillary notices a warning from a static analysis tool — this variable contains user-provided data. If it is used in a sensitive context before being sanitized, this code could be vulnerable to security vulnerabilities.

Hillary sets out to determine if the variable is sanitized on all paths leading to the sensitive context. She vaguely recalls using an Eclipse tool to help her trace control flow through a program, but is unsure how to invoke it or whether it also traces data flow. She looks through various menus, tries out a few tools, but cannot locate the right tool. Turning her attention back to the code, she 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, she now begins searching for methods that take the variable as input. She stumbles upon Eclipse's *Call Hierarchy* tool, which seems helpful, but she is unable to specify the variable she is interested in when invoking the tool.

Undeterred, she finds a method that takes the variable as a parameter and invokes *Call Hierarchy* on that method. Eclipse opens a new view containing *Call Hierarchy's* representation of the call graph. Hillary continuously switches back and forth between the *Call Hierarchy* view and the code. She uses the *Call Hierarchy* view to navigate chains of method calls and inspects the code in each method to check whether the variable gets sanitized correctly. She repeats this process for a few more call sites, but grows fatigued. Unsure of whether the the variable gets sanitized along all paths, she decides to ignore the warning. Two months later an attacker exploits the vulnerability, costing her company hundreds of millions of dollars. Hillary gets fired for her oversight. Though Hillary is fictional, her story is based on the experiences of real developers we observed in a previous study [6].

The contribution of this work is the design, implementation, and evaluation of a new program navigation tool, Flower. As we will discuss, Flower addresses many of the types of issues that Hillary faced by implementing four key design principles (Section II). We evaluate Flower, to better understand what

types of tasks it effectively supports. In contrast to existing navigation tools, Flower enables developers to simultaneously trace data flow and control flow within the code view.

II. Design Principles

In this section we describe the design principles that we used to shape Flower. We derived these design principles from a previous study [6] and by examining existing program navigation tools.

Powerful Program Analysis — 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. Textual analysis also falls short when programs contain inheritance and when variable names are included in comments, documentation, or other syntactically irrelevant locations. By leveraging powerful program analysis techniques, navigation tools can provide more accurate information than simple textual analysis. By analyzing abstract syntax trees (ASTs) and call graphs, tools can make references to relevant variables and methods.

Low Barriers to Invocation — Some tools are easier to invoke than others. Take Hillary's case for example. She 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 [10]. As developers navigate multiple program paths concurrently, repetitively invoking tools may be cumbersome, especially if barriers are high.

Full Program Navigation — Developers are not only interested in traversing programs' call graphs, but also how data flows through the call graph [6]. 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. For Hillary, *Mark Occurrences* helped her navigate a single file, but fell short when she wanted 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.

In Situ Navigation — Switching between views in the IDE can cause disorientation [11]. As developers navigate through code, navigation tools should present their results in that context. This was a problem for Hillary, who had to constantly switch between the *Call Hirerchy* 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.

III. Related Work

Here we discuss many of the various existing tools that help developers explore and navigate code. We also relate the exiting tools back to Design Principles Described in Section II.

Many modern IDEs provide tools that help developers navigate through their code. For example, Eclipse [12] includes *Call Hierarchy* and *Find References*. When users invoke *Call Hierarchy*, Eclipse opens a new view that displays the callers and callees for a selected method. IntelliJ [13] also provides navigation tools, namely *Analyze Data Flow To/From Here* and *Analyze Dependencies*. Much like Eclipse's *Call Hierarchy*, the *Analyze Data Flow* tools display their results in external views. These tools provide *Powerful Program Analysis* for users to navigate throughout the entirety of a project. However, they generally differ from Flower because they lack *Low Barriers to Invocation* and *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 approaches are similar to Flower in that they display the results within the editor rather than a separate view or panel. However, they do not provide *Powerful Program Analysis* or enable *Full Program Navigation*.

Many other tools help developers navigate code by representing the code graphically and allowing developers to navigate those graphs [1], [7], [8], [14]–[17]. 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. To generate accurate visualizations, these types of tools utilize *Powerful Program Analysis*. Additionally, most of these tools implement some aspects of *Full Program Navigation*. Flower differs because 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 [12]. 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.

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

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

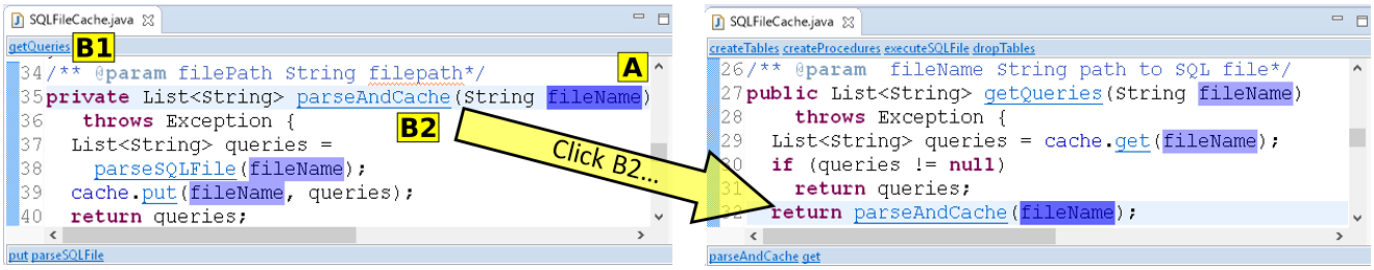

Fig. 1: Code view in the Eclipse IDE with Flower invoked during Task 1. The image on the left shows the tool running in the method `parseAndCache`. When the user clicks on the link in the editor (B1 or B2), they are taken to the function `getQueries`, which is a caller of `parseAndCache`.

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.[1] As a baseline, we compare our tool against the existing suite of tools available in Eclipse (*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, and concludes with our approach to answer (b).

All participants were graduate students at the time of the study with a mean of 5 years of professional programming experience; Table I provides additional information about each participant. We recruited participants using a convenience sampling approach. 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.

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. Each participant was assigned to one of four groups (Task Group column in Table I).

[1]Materials available online at tinyurl.com/flowerTool

We based our tasks in this study on two tasks (Tasks 1 and 3) from a previous study [6]. 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.

To evaluate the usability of Flower, we administered an adapted version of the Post-Study System Usability Questionnaire (PSSUQ) [18] 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. To prompt discussion about the usability of Flower, we also asked participants open-ended questions based on applicable categories from Nielsen's usability heuristics [19].

VI. Results

Here we present the results of our preliminary evaluation. We present our results thematically to illustrate the types of activities Flower effectively supported.

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."

TABLE I: Participant Demographics

Participants	Industry Exp. *(years)*	Java Exp. *(years)*	Eclipse Exp.	Task Group
P1	9	5	✓	T1* - T2
P2	0	6	✓	T2* - T1
P3	3	2	-	T1 - T2*
P4	5	0	-	T2 - T1*
P5	12	10	✓	T1* - T2
P6	0	3.5	✓	T2* - T1
P7	1	9	✓	T1 - T2*
P8	5.5	3.5	✓	T2 - T1*

* Used Flower for task

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. While working on Task 2 with Flower P6 attempted to use Eclipse's built-in back buttons to backtrack, but still had difficulty reorienting himself. 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*). 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. By tracking developers' progress, Flower could display already-visited locations differently than unexplored methods either positionally or through the use of colors. Additionally, in a similar approach to that of Whyline [1], Flower could use animation to transition more smoothly between locations, perhaps giving users a sense of naturally moving through the code.

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. 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

In this work we presented a new tool, Flower, that helps developers navigate program flow with its minimalistic interface. Flower fills a void between ad-hoc, 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.[2]

[2]This work is supported by NSF grant number 131832

REFERENCES

[1] 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. [Online]. Available: http://doi.acm.org/10.1145/985692.985712

[2] *The Economic Impact of Inadequate Infrastructure for Software Testing*. National Institute Of Standards & Technology, May 2002, no. Planning Report 02-3. [Online]. Available: http://www.nist.gov/director/planning/upload/report02-3.pdf

[3] R. Brooks, "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies*, vol. 18, no. 6, pp. 543 – 554, 1983. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020737383800315

[4] 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.

[5] 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.

[6] 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

[7] 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.

[8] 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. [Online]. Available: http://doi.acm.org/10.1145/1117696.1117701

[9] 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. [Online]. Available: http://doi.acm.org/10.1145/2047196.2047225

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

[11] 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

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

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

[14] 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. [Online]. Available: http://doi.acm.org/10.1145/1753326.1753706

[15] 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. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810331

[16] 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. [Online]. Available: http://dx.doi.org/10.1109/WPC.2005.37

[17] 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. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321637

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

[19] 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