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. 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 navigate program flow upstream and downstream within the Eclipse Integrated Development Environment (IDE). Based on a preliminary evaluation with 8 programmers, 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. Finding and fixing bugs has become the most common software activity taking up 70-80% of software engineers' time [1], and a software testing study described a wide variety of tools developers use to validate their code in different ways but also found that inadequate testing and debugging tools were a contributing factor to buggy poor quality software in production [2]. Program navigation is a central task that 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]. Code navigation can also be applied to exploring new code bases and assessing security vulnerabilities.

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 [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 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, that represents a minimalistic approach to program navigation. Flower leverages powerful program analysis techniques and integrates all of its results into the code view. In evaluating Flower, we want to understand what types of tasks it can effectively support without leaning on the the additional user interface elements that characterize other tools.

In this work we designed, implemented, and evaluated a program navigation tool, Flower. In contrast to existing navigation tools, Flower embodies four key design principles (Section III) that enable developers to simultaneously trace data flow and control flow within the code view.

II. Motivating Example

Consider 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 various security vulnerabilities."

She vaguely recalls that Eclipse provides tools to help her trace control flow and data flow throughout the program, but is unsure how to invoke them. She spends some time looking 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, scanning for uses of the variable. Satisfied she has inspected all the occurrences in the current file, she now searches globally for the variable. Frustratingly, her search returns an overwhelming number of irrelevant results, including references to the variable's name in comments and documentation. Unsure of whether the the variable gets sanitized or if it gets used in a sensitive context, she decides to ignore the warning. Though Hillary is fictional, her story is based on the experiences of real developers we observed in a previous study [6].

III. 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 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. Tools with high barriers to invocation require users to sift through menus and include unintuitive widgets. Barriers to invocation inhibit adoption. 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. 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 Navigation — Switching between views in the IDE can cause disorientation [10]. 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.

IV. Background

There are a variety of tools available to help developers explore and navigate code. Here we discuss two types of tools, production tools and visualization tools and Table I shows a comparison between Flower and other related tools in terms of the four design principles used to implement our tool.

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 [11] and *Analyze Data Flow To/From Here* and *Analyze Dependencies* in IntelliJ [12]. These tools provide powerful program analysis for users to navigate throughout the entirety of a project, however they generally differ from our tool in that they may require multiple steps to start the tool or modify the user's navigation (i.e. switching between up and down search), may need repeated invocations of the plugin to track multiple paths or variables in the code, and force users to switch between the text editor of the IDE and a new panel displaying the results.

There are also several production tools that are strictly consolidated within the editor. Two examples of these types of commercial tools include *Mark Occurrences* [13] and *Open Declaration* in the Eclipse IDE. These approaches are similar to our tool in that they display the results within the editor rather than a separate 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 or variable declaration the user wants to open.

TABLE I: Design Principles

Tools	Powerful Analysis	Low Barriers	Full Prog. Nav.	In Situ Nav.
Call Hierarchy	✓	-	✓	-
Find References	✓	-	✓	-
Analyze Data Flow	✓	-	✓	-
Analyze Dependencies	✓	-	✓	-
Mark Occurrences	-	✓	-	✓
Open Declaration	✓	-	✓	✓
Code Bubbles	✓	-	✓	✓
Code Canvas	✓	-	✓	✓
Code Surfer	✓	-	✓	Sometimes
Dora	✓	-	✓	-
Reacher	✓	-	✓	-
Relo	✓	-	✓	-
Whyline	✓	-	✓	-

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* [14], *Code Canvas* [15], *Code Surfer* [16], *Dora* [17], *Reacher* [7], *Relo* [8], *Whyline* [1], and many more research tools. 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. Flower differs because, while all visualization tools can provide powerful data analysis, but we desire minimal steps to invoke the tool and present the information inside of the editor.

V. Flower

Flower was designed to realize all of the principles described in Section III. We implemented Flower as a plugin to the Eclipse IDE [11]. 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 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 modifying the value of the `fileName` variable, users could gain access to sensitive information in the database. You want to determine whether users can modify `fileName`

TABLE II: Participant Demographics

Participants	**Industry Exp.** *(years)*	**Java Exp.** *(years)*	**Used Eclipse**	**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

before it gets passed into `getQueries` or if `fileName` is always bound to hard-coded parameters. First, you click on `fileName` (A). 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 `getQueries`, any method calling `getQueries` could modify `fileName`. Those methods reside in other class 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 `getQueries` (B2), which conveniently links to the first call site, `createTables`. Flower opens `createTables` and highlights the location in that method where fileName is passed to `getQueries` as seen in Fig. 1. Additionally, if the selected variable has uses that do not appear in the user's current view of the editor then the tool will provide links to those off-screen line numbers in the top box if it appears above the current location and in the bottom box if it is used below (not shown).

VI. Preliminary Evaluation

We performed a preliminary evaluation of Flower with eight programmers performing two code navigation tasks. 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) get feedback on the usability of Flower and (b) determine what types of navigation activities Flower effectively supports. Section VI-A outlines our approach to answering (a). To answer (b), we measured how quickly and accurately participants completed different activities. The remainder of this section describes our participants, study design, and task selection.

All participants were graduate students at the time of the study with a mean of 5 years of professional programming experience; Table II 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. We used this information to 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 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 (Task Group column in Table II).

We analyzed the data from our previous study [6], 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 [6] partially inspired the development of Flower, we include the same two tasks in this study.

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. 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.

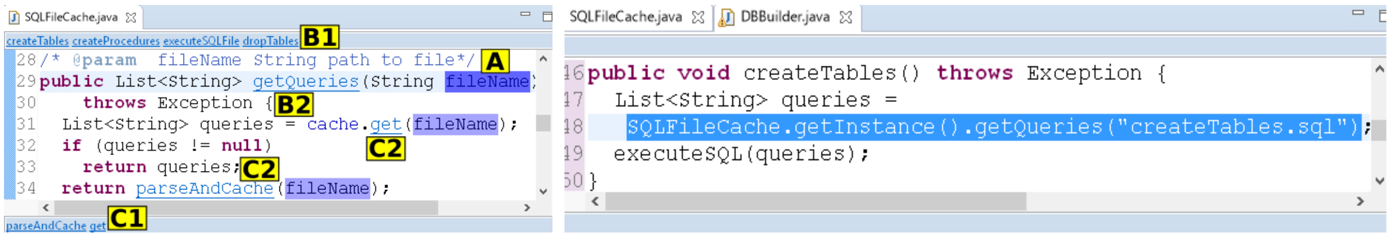
A. *Usability Evaluation*

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. We asked 10 questions; participants responded on a 7-point Likert scale from "Strongly Disagree" to "Strongly Agree." 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].

VII. Results

Approachable Interface: On the PSSUQ, participants responded most positively to questions about Flower's simplicity, how easy it was to use, and how easy it was to learn. The median response to these three questions was 6. 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. 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 itself only called in one method `getQueries`. Participants equipped with our tool were strictly faster in navigating



```
/* @param  fileName String path to file*/
public List<String> getQueries(String fileName)
    throws Exception {
  List<String> queries = cache.get(fileName);
  if (queries != null)
    return queries;
  return parseAndCache(fileName);
```

```
public void createTables() throws Exception {
  List<String> queries =
    SQLFileCache.getInstance().getQueries("createTables.sql");
  executeSQL(queries);
}
```

Fig. 1: Code view in the Eclipse IDE with Flower invoked on the method `getQueries` from Task 1

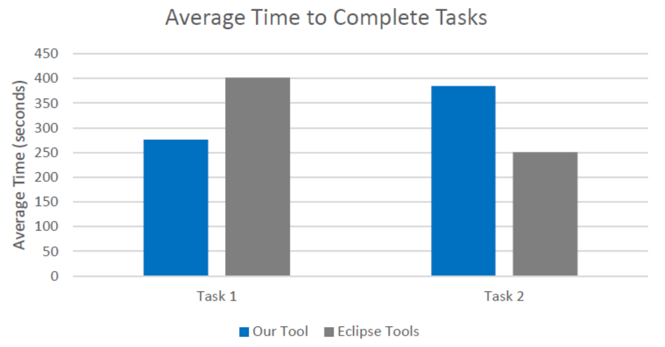


Fig. 2: Mean time to complete each task with and without Flower

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.

Get Me Outta Here: 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 (Figure 2). 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 confounding variables and branches to complete Task 2. When participants took missteps they found it difficult to retrace their navigation path. 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?"

VIII. 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, 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 any dedicated tools can be frustrating and unfruitful (Section II). 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.

IX. Limitations

Small number of participants, only two tasks.

X. Conclusion

The authors would like to thank...

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. 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

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

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

[13] "Mark occurrences," http://www.eclipse.org/pdt/help/html/mark_occurrences.htm.

[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