Stacksplorer: Call Graph Navigation Helps Increasing Code Maintenance Efficiency

Thorsten Karrer[†] *Jan-Peter Krämer*[†] *Jonathan Diehl*[†] *Björn Hartmann*[‡] *Jan Borchers*[†]

[†]RWTH Aachen University
52062 Aachen, Germany
{karrer, kraemer, diehl, borchers}
@cs.rwth-aachen.de

[‡]Computer Science Division
University of California
Berkeley, CA 94720
bjoern@cs.berkeley.edu

ABSTRACT

We present Stacksplorer, a new tool to support source code navigation and comprehension. Stacksplorer computes the call graph of a given piece of code, visualizes relevant parts of it, and allows developers to interactively traverse it. This augments the traditional code editor by offering an additional layer of navigation. Stacksplorer is particularly useful to understand and edit unknown source code because branches of the call graph can be explored and backtracked easily. Visualizing the callers of a method reduces the risk of introducing unintended side effects. In a quantitative study, programmers using Stacksplorer performed three of four software maintenance tasks significantly faster and with higher success rates, and Stacksplorer received a System Usability Scale rating of 85.4 from participants.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

General terms: Design, Human Factors

Keywords: Development Tools / Toolkits / Programming Environments, Visualization.

INTRODUCTION

A large part of a software developer's work is maintaining existing code bases. This includes adding new features, fixing bugs, and refactoring code after the software has been shipped. Maintenance has been shown to make up as much as 70% of the total expenses of a software project [19]. Because of this, analyzing and improving how programmers maintain code has recently become an active area of research.

Maintenance often comprises reading and understanding an unfamiliar piece of source code and identifying problematic sections. The developer then has to modify these sections while being aware of potential side effects [15]. To do this, the developer spends a large part of his time navigating through the code [20]. Navigating source code includes file-based navigation, such as switching between classes that are usually implemented in separate files, and navigating the semantics of the code, e.g., locating a variable definition or exploring the callers and callees of a certain method. Navigating the call graph has been shown to be particularly important to understand code, to find out where to modify it, and to assess the scope and side effects of any changes [13].

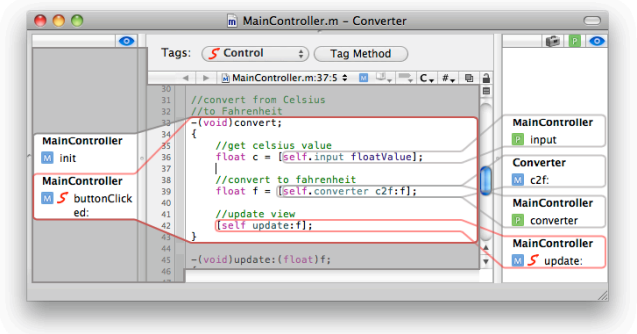

Figure 1: Stacksplorer visualizes and allows browsing the call graph of an application's source code. For the current *focus method*, it shows callers in the left column and methods called in the right.

We present Stacksplorer, a plug-in that modifies the user interface of the Xcode[1] integrated development environment (IDE) to visualize and navigate the neighborhood of any given method in the call graph (Fig. 1). Stacksplorer lets developers see and access the callers "upstream" of a method, reducing the risk of introducing unintended side effects. They can also navigate downstream to understand how operations are implemented in an unknown piece of code. The call graph visualization helps developers retain the context of a method and simplifies exploring and backtracking along interesting branches of the call graph.

In a controlled experiment, developers solved code maintenance tasks using Xcode with and without Stacksplorer. Our results show that Stacksplorer helps developers to solve more tasks correctly in less time, with a heightened awareness of possible side effects.

[1]http://developer.apple.com/technologies/tools/xcode.html

RELATED WORK

Navigation Behavior

Today, most IDEs offer only limited tools for call graph related code navigation. Moreover, these tools are often difficult to find and invoke. By observing programmers working with Eclipse on five change tasks in a 500SLOC[2] Java application, Ko et al. [11] found that about a quarter of the developers' time was spent navigating, either by following dependencies or by searching for names. Between code segments related through structural relationships, users navigated using scroll bars and the package browser (14% of the test period) or using the find and replace tool, although more suitable and effective tools are available in Eclipse (see also [18]). To perform comparisons, developers navigated back and forth between code segments because by default, Eclipse uses a single editing window.

Lawrance and Burnett [16] presented PFIS (Programmer Flow by Information Scent), a model based on information foraging theory to predict a programmer's navigation. In information foraging theory, each link between two pieces of information has a certain scent that determines how likely a user will follow this link on her search for relevant information. In PFIS, a link is defined as any navigation possible with one click; scent is determined by comparing words used in the source code with those used in a bug report, which verbally describes the users task. In an experiment, PFIS was capable of predicting the navigation behavior of professional software developers in a bug fixing task.

Recommender Systems

Recommender systems for software development aim at determining automatically which information is relevant to the developer's current task. Most systems first calculate a 'degree of interest' depending on the source file that is being edited, for all files in the project. They then recommend files with a degree of interest exceeding a threshold. Recommender systems may take into account: a programmer's navigation history [23]; the navigation history of all programmers in the same code [6]; the history of editing activities in addition to navigation [10]; and non-code resources such as version control systems [24].

For small tasks, user studies of these tools consistently found that the total effort spent on navigating through a project could be reduced significantly, and that users considered automated guidance in software projects helpful, especially when they were new to the project. For more complex projects, however, the effect recommender tools have may be minor compared to task and strategy specific effects [4]. In contrast to Stacksplorer, the recommender systems presented reduce the amount of information shown in the IDE, but they do not introduce new ways to visualize the structural relationships that led to the recommendations.

Call Graph Utilization

The Whyline [12] allows asking "Why did" or "Why didn't" questions about the textual and graphical output of an application. For example, a line drawn in a painting application may, after execution stopped, be queried "Why did this line's color = blue". Using a trace of the actual program execution, the Whyline computes a dynamic slice, i.e., it determines which method executions influenced the relevant property. Novice programmers using the Whyline could solve a bug fixing task significantly faster than expert developers without the Whyline. Stacksplorer is, in contrast to the Whyline, not a debugging but a navigation tool; it does not allow users to formulate an explicit question about a trace from a single execution but facilitates exploration of the call graph during code editing.

LaToza and Myers [14] scope searches in source code by restricting them to branches of the call graph originating in or leading to a certain method. The search results are visualized in a graph, revealing the paths leading from the starting point of the search to each result. Unlike Stacksplorer, this visualization is a feature of the search tool and not integrated with the source code editor as a means for navigation.

Commercial IDEs such as Visual Studio, Eclipse, or Xcode offer various tools for exploring the call graph. Eclipse, for example, is able to show multiple levels of callers and callees of a method in a tree view; Visual Studio offers similar functionality. These tools can be used for navigation, but they do not update automatically when the user navigates through the source code, and they are not part of, or visually connected to, the code editor. For navigation inside the source code editor, most IDEs provide functionality to directly jump to the definition of a method from where it is called, and some support jumping to callers; these tools, however, lack a permanent visualization that could provide additional information scent and help the user stay oriented.

Code Layout

Other tools focus solely on laying out relevant source code elements visually in order to help developers navigating and keeping track of source code. DeLine et al. [5] use the graphically scaled down source code text file as an overview map for navigation. While being too small to be readable, it allows discerning the structure of the source code, thus supporting inter- and intra-file navigation.

In Code Bubbles [2], code is viewed and edited in bubbles, which can be arranged freely on a 2D plane. The bubbles are connected with arrows representing existing relationships between them. Users can use a number of tools, e.g., "Open Declaration", to find exactly the bubbles they need. Code Bubbles is based on single methods as the basic navigational unit, thus breaking the prevalent paradigm of regular IDEs, which are based on a class-file dualism. In a qualitative study, programmers solved one of two maintenance tasks significantly faster with Code Bubbles than with Eclipse. The need for back-and-forth navigation was also reduced significantly. To combine the benefits of Code Bubbles and traditional IDEs, Stacksplorer offers the familiar file- (or class-)based navigation extended by a method-based navigation along the call graph.

[2]source lines of code (number of non-comment, non-empty lines of source code)

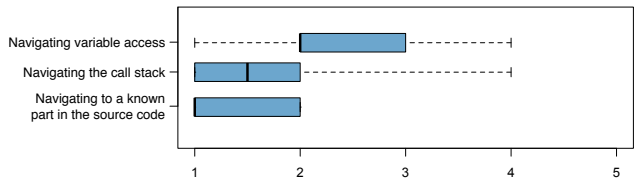

Figure 2: The boxplot diagrams above show how frequently (from 1=frequently to 4=seldom, 5=never) developers think they perform navigation of different types when working on bug fixing tasks.

FORMATIVE STUDY: HOW DEVELOPERS NAVIGATE IN XCODE

We conducted a small formative study to understand how developers navigate source code in different situations when using Xcode. To build on the findings of the aforementioned studies, which used Java and Eclipse, we first had to confirm that developers using Objective-C and Xcode navigate code in ways similar to Java developers.

Six Objective-C developers who worked on maintenance tasks at the time agreed to participate. Through a contextual inquiry, we analyzed the set of navigation actions our participants used during their work. For each navigation action we noted the goal (e.g., "find first definition of variable") motivating the action and the tool used to perform it (e.g., "jump to definition tool from contextual menu"). Afterwards, we asked the participants to estimate the importance of the different goals, and to rate the support the IDE provided for their navigation needs.

The most important goal when navigating source code was to edit sections of code that were well-known to the developers both in location and content (see Fig. 2). This kind of navigation happened mostly by switching between files and scrolling through the source, and it was perceived to be well supported by the IDE. When the developers' knowledge of the code was not sufficient to solve a task, and parts of the source code had to be analyzed and understood, they mainly followed structural relationships in the source code; most importantly, the call graph was explored, and they investigated which parts of the source code accessed an important variable. Since variable access usually happened through accessor methods, this also required an analysis of the call graph.

These findings confirmed our assumption that Xcode developers and Eclipse developers navigate through source code similarly. Interestingly, the developers not only navigated along edges of the call graph often, but they did so consciously, and regarded it as important. They also actively expressed that better tools for call graph navigation were needed.

NAVIGATING WITH STACKSPLORER

These results motivated the main idea of Stacksplorer's design: analyzing the code's *call graph* [21] to offer relevant information related to the user's *focus method*, i.e., the method she is currently working on or trying to understand. The call graph is a finite, directed graph in which each node corresponds to one method in the source code. For any pair (A, B) of methods, there exists an edge from A to B if (and only if) method B is called from the implementation of method A.

At any time, Stacksplorer shows the neighborhood of the focus method in the call graph, thus giving the user access to the callers of and methods called from the focus method. This is closely related to the way Herman *et al.* [8] suggest incremental exploration of graphs by placing a window on top of the graph, so that one *logical frame* is shown at a time. Huang *et al.* [9] coined the term *focus node* for the logical frame's central node, which defines which other nodes will belong to the logical frame. In our case, the focus method corresponds to the focus node, and, together with the information from its neighborhood in the call graph, forms the logical frame.

Stacksplorer directly integrates with Xcode, adding two interactive views alongside the code editor (Fig. 1). These three views together display the current logical frame, basically acting as a fisheye view [7] for the call graph. In the central source code editor, the focus method is displayed in the usual way. The side views show the names and classes of the neighboring methods in the call graph. The left view shows methods calling the focus method, the right shows methods called from the focus method.

The items in the side columns are always positioned to minimize the on-screen distance to the related code in the editor. This simplifies grasping the context of the focus method. Since the items in the right column reflect the order they appear in the code, this also provides a rough overview of what the focus method does. For densely written code, e.g., nested method calls, the assignment of items in the columns to code locations can become unclear. To address this, Stacksplorer optionally displays graphical overlays that connect the method call in the source code to the corresponding item in the side column (Fig. 1).

Most importantly, Stacksplorer adds a new 'axis' for navigation through a project's source code; in addition to navigating through a single class by scrolling vertically in the editor, Stacksplorer allows navigating horizontally by clicking on a method in one of the side columns. This shifts the logical frame inside the call graph and makes that method the focus method. For example, navigating to a method that calls the focus method by clicking on it in the left column will cause the contents of all three columns to shift to the right (Fig. 3). The method that was selected moves to the center and opens in the central editor, and the side columns are updated accordingly. The previous focus method becomes an entry in the list of called methods to the right. While this functionality is similar to the *go to definition* and *find all references* commands of most IDEs, it differs in the way that it continually displays information scent and navigation affordances that update dynamically.

Developers often find that existing means of documentation are either hard to maintain or do not convey enough information to fully understand how a given feature of an application is implemented [15]. Comments in the source code, for example, are tied to one location, making it difficult to communicate which paths in the call graph constitute a fea-

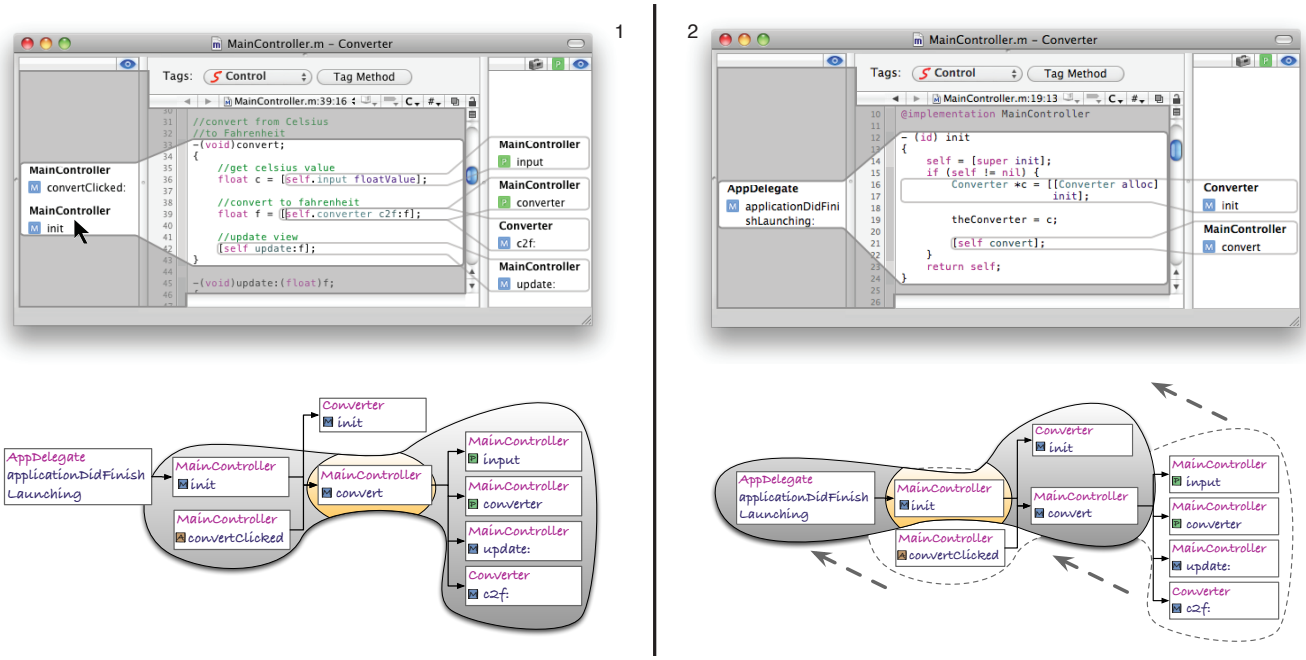

Figure 3: The focus method that is being edited in the IDE (top) corresponds to the focus node in the call graph (bottom). Clicking on a method in the left column of the Stacksplorer UI (1) shifts the logical frame inside the call graph and makes the clicked method the focus method (2).

ture and how the methods involved work together. To this end, Stacksplorer supports assigning *tags* to methods as a lightweight form of documentation; important paths through the call graph can be bookmarked for later reference by tagging their methods. Tags can be named, and they are visualized by colored frame overlays in the source code, such as the red frames in Fig. 1. This way, tags can communicate what purpose a certain method serves, or which features of the application use a specific method. For example, the methods that are involved with screen redraws could be assigned a common tag "redrawing" to clearly delineate this section of the source code.

One of the limitations of call graph navigation (and, therefore, of Stacksplorer) is that we cannot accurately determine all call stacks that may be executed during a real program execution; in other words, it is impossible to correctly and completely determine the call graph. In general, the question if a path is reachable at runtime is undecidable [17]. For example, static analysis is not able to detect if methods are dynamically dispatched to a different object or a different implementation at runtime. For these cases, tracing an execution of the application would be needed. Despite these limitations, we still decided to base Stacksplorer on static analysis: the list of callees is populated with all methods that match the name of the Objective-C selector and whose class or ancestor class matches the class of the receiver. Thus, we also get all possible overrides of a method, which is acceptable in Objective-C with its typically fairly flat inheritance structures. In terms of robustness against syntax errors, we rely on Xcode's parser. This approach yields a result more quickly than runtime traces, and it works even if the source code does not compile because of a bug. While providing live updates to the visualization on every keystroke is possible, we currently limit updates to every save operation for performance reasons.

EVALUATION

We tested the effectiveness of Stacksplorer's call graph visualization and navigation in an experiment with 16 experienced programmers performing code maintenance tasks with and without Stacksplorer support. Our main goals for this study were to better understand how practitioners use Stacksplorer's features when working with unknown code, and to quantitatively compare task performance and success rates. For the latter, we formulated the following hypotheses:

H1 More programmers can successfully solve a time-constrained task that requires browsing and understanding previously unknown source code when using Stacksplorer than when using a default Xcode installation.

H2 Programmers can solve tasks that require browsing and understanding previously unknown source code more quickly when using Stacksplorer than when using a default Xcode installation.

H3 Programmers can identify side effects of changes made to unknown code more quickly when using Stacksplorer than when using a default Xcode installation.

As a code base for the study, we used the open source project BibDesk[3], a bibliography manager for Mac OS X. We selected this project because it is sufficiently complex with

[3]http://bibdesk.sourceforge.net

about 88.000 lines of code in over 400 classes, it is stable, and it is written entirely in Objective-C.

The experiment was conducted on a Mac Pro computer with a 23" screen at a resolution of 1920x1200. Xcode was opened with the test project in a maximized window, and the screen was recorded for the entire session.

Methodology

In the experiment, we compared two conditions: in the *experimental condition* participants could make use of Stacksplorer's features; in the *control condition* Stacksplorer was not available. Participants had to solve two similarly difficult programming tasks, each consisting of two subtasks as explained below. We counterbalanced task order and task to condition assignment.

At the beginning of each user test, we demonstrated Stacksplorer using an unrelated code base from one of our own projects. We explained how to use the interactive visualization of the call graph neighborhood and the tagging feature. To mitigate possible learning effects between the two trials for each participant, we allowed them to then familiarize themselves with the BibDesk source code for 10 minutes.

The participants did not know all tasks beforehand; we always introduced only the task they should work on next. The time to complete each task was limited to 25 minutes for the first subtask and 15 minutes for the second subtask to limit the effects of fatigue. Additionally, participants were allowed to take breaks between the tasks. While solving a task, speed was the priority of the participants but it was made clear that they should arrive at a working solution. To be consistent with [2, 20], participants were allowed to use all of Xcode's code navigation features, except for runtime analysis tools such as the debugger. The participants were, however, allowed to examine the run-time behavior of the BibDesk software using a precompiled version of BibDesk as a reference.

For each task and subtask, we measured the time until the participant considered the task complete, and we checked whether the task was solved correctly. We also encouraged users to think aloud and tell us their motivation for using certain navigation techniques when working on the tasks. An experimenter was supervising each trial and took notes; additionally, the trials were videotaped for further analysis.

After the test, we asked our participants to fill out a questionnaire with the questions of the System Usability Scale (SUS) [3] and six additional questions (see Fig. 5) specifically addressing the usefulness of the extra features introduced by Stacksplorer. Finally, we conducted an open interview with each participant to gain more insights into their behavior.

Participants

Our group of participants consisted of 14 students (6 graduate, 8 undergraduate) and 2 professional software developers. By employing mostly students, we aimed to reduce the impact of different levels of programming expertise on the study, in accordance with [2]. Age ranged from 22 to 34 ($M = 27.7$), and there were no female participants (although this was not intended by design). All participants were familiar with Xcode; their experience with the IDE ranged between 3 months and 6 years ($M = 2$ years), and they reported to do programming work between less than an hour and 40 hours a week ($M = 13.1$ hours). Ten participants were already familiar with the BibDesk application, but none of them had seen the source code before.

Tasks

The tasks should include searching for a specific location in the code and identifying side-effects of a specific change of the code. Therefore, we designed two different task types: (1) identifying where a certain functionality is implemented and change it; (2) identifying what side effects a change of a certain functionality has. Tasks of type (1) were considered solved if the suggested modification by the participant would have achieved the intended effect. This was determined by the experimenter who was familiar with the code. For tasks of type (2), we considered any functional change in the behavior of the application (other than the one requested in the task) that was caused by the modifications applied to the code a side effect. For each type, we defined two separate tasks to allow testing both conditions with every participant.

The first task concerned BibDesk's *Autofile* feature, which automatically sorts PDF documents of publications into dedicated folders and renames them according to a user-definable naming scheme. In the first subtask (1.1), participants were asked to prepend the string "TRIAL" to every generated PDF file name. In the second subtask (1.2), we asked the participants to identify side effects that would occur if the change from the first subtask was implemented in a specific method.

The second task was to modify the behavior of BibDesk when generating BibTeX output. In the first subtask (2.1), participants were asked to prepend the string "TRIAL" to every publication's notes entry. The second subtask (2.2) concerned the search command from BibDesk's AppleScript API. Participants were asked to change the behavior of the search command to also match the search string against a publication's journal name. Furthermore, the participants had to identify what side effects this change had.

Both tasks were similar in complexity, because the subtasks of type 1 required navigating along at least three edges of the call graph to solve them, and the subtasks of type 2 likewise required at least two edges. A task was considered to be completed successfully if both subtasks were solved correctly.

Quantitative Results

Task Success. Only four participants were able to solve both tasks. All tasks except for task 2.1 were solved correctly more often by participants in the experimental condition. However, the difference in success rates was only significant for task 1 (Fisher's test, $p = 0.041$). Hence, the results are not conclusive enough to prove H1.

We assume that the non-significant results for task 2 were caused by two factors: firstly, in task 2.1 the participants could benefit from knowledge in standard APIs more than in task 1.1 because task 2.1 was concerned with document saving, which uses a standardized API. Secondly, task 2.2 was probably the easiest of all tasks; independent of the condition, it was completed successfully most often.

Task Completion Times. Average times are shown in Table 1. Developers solved task 1 in the experimental condition significantly faster than in the control condition ($t(14) = -2.32$, $p = 0.018$, $d = 1.16$, one-tailed). A comparison for task 2 did not yield a statistically significant result ($t(14) = 0.20$, $p = 0.84$, $d = 0.10$, two-tailed). Further analysis revealed that task 2.2, like task 1, was solved significantly faster in the experimental condition ($t(14) = -2.37$, $p = 0.016$, $d = 1.18$, one-tailed). The results for task 2.1 were inconclusive ($t(14) = 1.17$, $p = 0.26$, $d = 0.58$, two-tailed). Apart from subtask 2.1 these results support H2.

We specifically designed subtasks 1.2 and 2.2 to require finding and understanding the side effects that would arise from changes to the code. The comparison for subtask 2.2 clearly supports H3; the analysis of subtask 1.2 also indicated that developers could identify side effects better using Stacksplorer, but the results were not significant ($t(14) = -1.25$, $p = 0.12$, $d = 0.62$, one-tailed).

	experiment		control	
	M	*SD*	*M*	*SD*
Task 1	**20m 43s**	**6m 49s**	**28m 45s**	**7m 4s**
Task 1.1	14m 45s	5m 54s	19m 39s	6m 16s
Task 1.2	5m 58s	4m 5s	9m 5s	5m 49s
Task 2	**22m 55s**	**9m 22s**	**22m 0s**	**8m 34s**
Task 2.1	17m 14s	7m 46s	13m 3s	6m 29s
Task 2.2	5m 42s	2m 13s	8m 58s	3m 12s

Table 1: Average task completion times for the different tasks and subtasks in the experiment and control conditions.

Qualitative Results

We observed that all participants used a similar high level strategy for finding the correct location to implement a change. When searching for a location for a change, participants usually started with an *exploration phase*, in which they searched for an *anchor point*. These anchor points correspond to what Sillito et al. [22] called *focus points*. Ko et al. [11] also observed similar behavior and referred to the exploration phase as *search phase*. Once the users found an anchor point, a *traversal phase* followed, in which they traversed the call graph until they either found the correct location for a change or noticed that they got lost and had to start again with a new exploration phase. During the traversal phase, participants often navigated along an outgoing path in the call graph and came back to the previously viewed method or to the anchor point if they decided to discard the path. This simple model is depicted in Fig. 4.

In tasks 1.1 and 2.1, all users started with some exploration phase, the length of which varied substantially. In task 1.2 and 2.2, the starting point for participants was indicated clearly, so the exploration phase was much shorter and had much less influence on success and completion time for these tasks. During the exploration phases, the participants used different techniques: Most performed a project-wide search for a term probably related to the task. Another popular technique was to find the UI related to the task and to look up which methods were called by the controls in the interface.

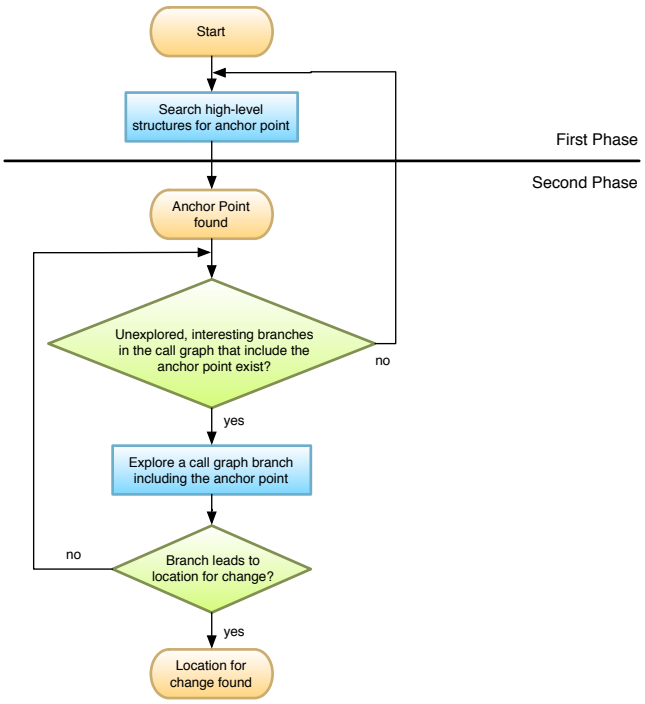

Figure 4: A simple model of programmers' navigation behavior splitting exploration of source code into two phases: developers first search for a place to start their investigation from, then they use the call graph to navigate through the source code.

Stacksplorer is not designed specifically to support the initial exploration phase, as it does not provide a high-level overview of the project and does not facilitate searching or opportunistic browsing [20]. It is much more useful during the traversal phase. Two techniques participants employed to utilize the plug-in could be differentiated clearly: Most participants read a part of the source code and tried to understand (at varying levels of detail) its purpose. When they had identified the relevant section of the source code for the task at hand, they enabled Stacksplorer's overlays to see which methods were called from this section and then navigated there. In contrast, another group of participants read the methods presented by Stacksplorer in the right column as a summary of the method. In the extreme case, these participants did not inspect the source code at all; instead, they only browsed through the called methods and navigated to whatever they found interesting. Once Stacksplorer no longer revealed interesting called methods, the participants read the focus method's source code to decide if they had found the correct location for the requested change. The latter technique is of course much more prone to error, but it can be very fast.

Using Stacksplorer resulted in a significantly increased use of Xcode's forward and back navigation buttons. They work like the forward and back buttons in a web browser to navigate through a history stack of visited locations in the source code. In the preliminary study, these buttons were only used on very rare occasions. Stacksplorer tempted participants to explore a path to see where it brought them and to discard

it if they found themselves getting stuck. When a path was discarded, the participants used the back button to navigate back to a previous anchor point and started exploring another path from there. In a refined version of Stacksplorer we now accommodate for this behavior by showing the five most recently visited methods in the side columns with a colored background, the saturation of which is lower the further away the method's position in the history stack is.

All participants highly appreciated Stacksplorer's ability to help identify side effects. To determine the side effects of a change in a method, it is generally required to analyze all paths in the call graph that lead to the changed method. In Xcode, this requires searching for the changed method's name in the project; some other IDEs provide a dedicated tool to reveal callers of a method. Using Stacksplorer for this analysis was faster and more robust against errors in comparison to the project-wide search that otherwise was required in Xcode. Participants often missed Stacksplorer in the second task if they had already used it in the first task ("Can I get the plug-in for that [task 2.2] again?").

We observed that users analyzing side effects of a change tended to only inspect the direct predecessors of the changed method in the call graph. For example, in task 2.2 the changed method is only called once. Side effects exist only because this single caller of the changed method is used in a different context than the one described in the task. When using Stacksplorer, participants navigated back once from the starting point and then found the relevant information in the left side column. In the control condition, users performed a search to find callers of a method, but they often hesitated to explore higher degree neighborhoods in the same way, in some cases causing them to fail the task.

Postsession Questionnaire

The combined score from the SUS questions was 85.4 on average ($SD = 7.4$). According to Bangor *et al.* [1], this score can be considered "excellent". This result is quite promising given that Stacksplorer is still a research prototype with some performance issues and minor bugs.

Nearly all participants strongly agreed that Stacksplorer makes it easier to understand source code compared to Xcode without the plug-in (Fig. 5, Q12). However, quite a few participants still found understanding source code challenging even when using Stacksplorer. This comes as no surprise since a large, feature-rich software project is a complex artifact and hard to understand without prior knowledge.

More than half of the participants strongly agreed that navigation with Stacksplorer is faster than without it. We think that this is primarily due to Stacksplorer's support for navigation to callers of a method. This type of navigation seems to be important for programmers: Stacksplorer improved their overall impression of how quickly they can navigate through source code solely by improving call graph navigation.

More than half of the participants did not feel lost in the source code when using Stacksplorer (Fig. 5, Q16). Compared to Xcode, users found that Stacksplorer supports orientation in the source code considerably better (Fig. 5, Q15).

SUMMARY AND FUTURE WORK

We presented Stacksplorer, a new tool to support source code navigation and comprehension. Stacksplorer visualizes the call graph neighborhood of a method in an application and supports navigating through it. Thus, Stacksplorer exploits a semantic aspect of source code to suggest relevant methods for exploration. Information displayed in Stacksplorer is visually linked to the source code. A prototype of Stacksplorer was implemented as a functional IDE plug-in.

Our user study showed that software maintenance tasks in a large open-source application could be completed significantly faster with Stacksplorer support. Participants reported that they were very satisfied with the plug-in and would like to use it for real world projects.

Two aspects present themselves as promising directions for future work: Firstly, having found that structural relationships in source code are of particular value for developers to better comprehend source code, more of theses relationships besides the call graph could be visualized and made accessible in IDEs. Secondly, we found that in some situations an additional, higher-level overview of the source code than what is provided by Stacksplorer would have been beneficial to get a first idea of the application's structure. This visualization could support transitioning fluently between a Stacksplorer-like view, which shows much source code with some context, and a new kind of visualization, showing less source code but a greater part of the application's structure.

ACKNOWLEDGEMENTS

This work was funded in part by the German B-IT Foundation and by the German Government through its UMIC Excellence Cluster for Ultra-High Speed Mobile Information and Communication at RWTH Aachen University.

REFERENCES

1. A. Bangor, P. Kortum, and J. Miller. An Empirical Evaluation of the System Usability Scale. *Intl. Journal of Human-Computer Interaction*, 24(6):574–594, Aug. 2008.

2. A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola. Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance. In *Proc. CHI '10*. ACM, 2010.

3. J. Brooke. *SUS-A quick and dirty usability scale*. Taylor and Francis, London, 1996.

4. B. de Alwis, G. C. Murphy, and M. P. Robillard. A Comparative Study of Three Program Exploration Tools. In *Proc. ICPC '07*, pages 103–112. IEEE, 2007.

5. R. Deline, M. Czerwinski, B. Myers, G. Venolia, S. Drucker, and G. Robertson. Code Thumbnails: Using Spatial Memory to Navigate Source Code. In *Proc. Visual Languages and Human-Centric Computing 2006*. IEEE, 2006.

6. R. DeLine, M. Czerwinski, and G. Robertson. Easing Program Comprehension by Sharing Navigation Data.

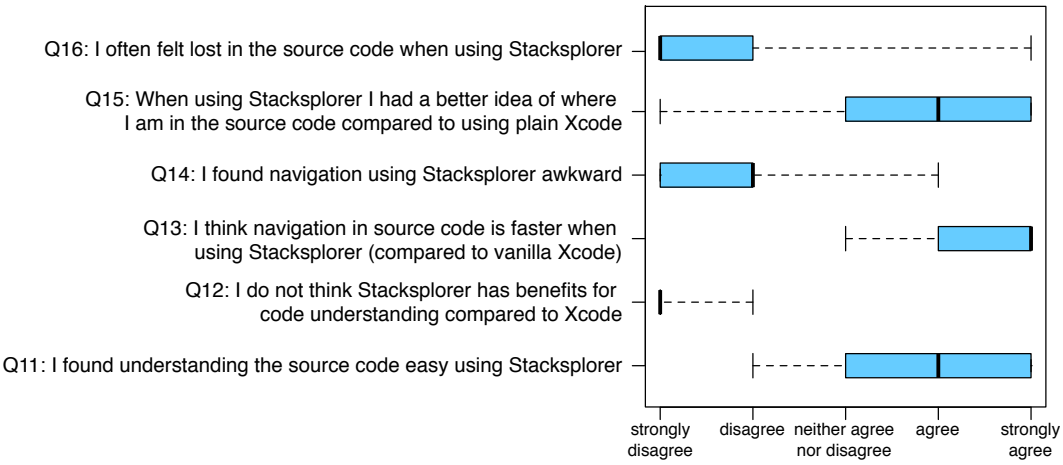

Figure 5: The boxplot diagrams show how the participants of our study responded to the six questions of the post-session questionnaire that did not belong to the SUS.

In *Proc. Visual Languages and Human-Centric Computing 2005*. IEEE, 2005.

7. G. Furnas. Generalized Fisheye Views. In *Proc. CHI '86*. ACM, 1986.

8. I. Herman, G. Melancon, and M. Marshall. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

9. M. Huang, P. Eades, J. Wang, and P. R. China. Online Animated Graph Drawing Using a Modified Spring Algorithm. *Journal of Visual Languages and Computing*, 9(6), 1998.

10. M. Kersten and G. C. Murphy. Mylar: A Degree-of-Interest Model for IDEs. In *Proc. Aspect-oriented Software Development*, pages 159–168. ACM, 2005.

11. A. Ko, B. Myers, M. Coblenz, and H. Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006.

12. A. J. Ko and B. A. Myers. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *Proc. ICSE '08*, pages 301–310. IEEE, 2008.

13. T. D. LaToza and B. A. Myers. Developers Ask Reachability Questions. In *Proc. ICSE '10*, pages 185–194. ACM, 2010.

14. T. D. LaToza and B. A. Myers. Searching Across Paths. In *2nd Intl. Workshop on Search-driven development: Users, Infrastructure, Tools and Evaluation*, 2010.

15. T. D. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *Proc. ICSE '06*, pages 492–501. ACM, 2006.

16. J. Lawrance, R. Bellamy, M. Burnett, and K. Rector. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proc. CHI '08*. ACM Press, 2008.

17. H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

18. G. C. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

19. R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 7. edition, 2010.

20. M. P. Robillard, W. Coelho, and G. C. Murphy. How Effective Developers Investigate Source Code:An Exploratory Study. *IEEE Transactions on Software Engineering*, 30(12), 2004.

21. B. G. Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, 1979.

22. J. Sillito, G. C. Murphy, and K. D. Volder. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

23. J. Singer, R. Elves, and M.-A. Storey. NavTracks: Supporting Navigation in Software Maintenance. In *Proc. IEEE Software Maintenance*. IEEE, 2005.

24. D. Čubranić and G. C. Murphy. Hipikat: Recommending Pertinent Software Development Artifacts. In *Proc. ICSE '03*. IEEE, 2003.