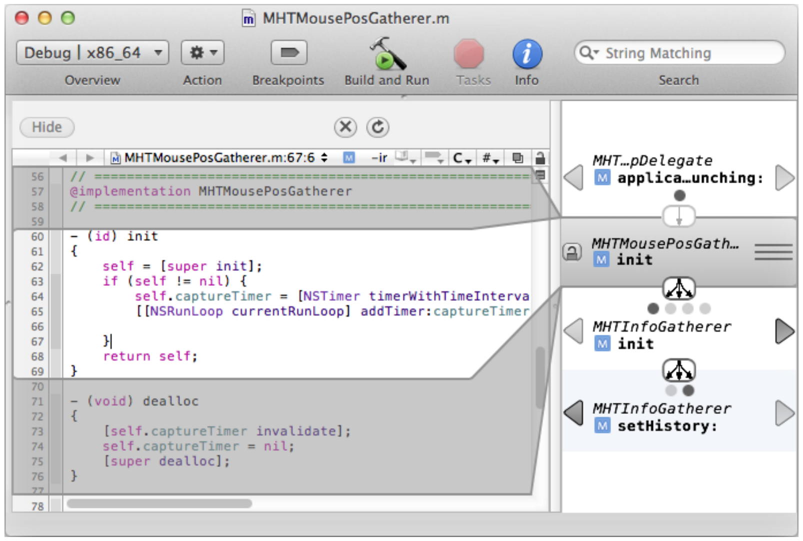
Blaze: Supporting Two-phased Call Graph Navigation in Source Code

Jan-Peter Krämer
Joachim Kurz
Thorsten Karrer
Jan Borchers

RWTH Aachen University
52062 Aachen, Germany
{kraemer, kurz, karrer, borchers}@cs.rwth-aachen.de



```
// ==========================================================
@implementation MHTMousePosGatherer
// ==========================================================

- (id) init
{
    self = [super init];
    if (self != nil) {
        self.captureTimer = [NSTimer timerWithTimeInterva
        [[NSRunLoop currentRunLoop] addTimer:captureTimer

    }
    return self;
}

- (void) dealloc
{
    [self.captureTimer invalidate];
    self.captureTimer = nil;
    [super dealloc];
}
```

Figure 1: Blaze visualizes a complete path in the call graph that includes the currently edited method. The developer can navigate up- and downstream from the focus method along the path, and change the path by selecting alternative entries using the arrow buttons next to each method.

Abstract

Understanding source code is crucial for successful software maintenance. A particularly important activity to understand source code is navigating the call graph [4]. Programmers have developed distinct strategies for effective call graph exploration [3, 9]. We introduce *Blaze,* a source code exploration tool tailored closely to these strategies. In a study, we compare Blaze to *Stacksplorer* [2], a tool that visualizes the immediate neighborhood of the current method in the call graph, to a tool resembling the standard Call Hierarchy view in the Eclipse IDE, and to an unmodified Xcode installation. The call graph exploration tools significantly increased success rates in typical software maintenance tasks, and using Stacksplorer or Blaze significantly reduced task completion times compared to using the Call Hierarchy or Xcode.

Keywords

Development Tools / Toolkits / Programming Environments; Visualization

ACM Classification Keywords

H.5.2 [**Information Interfaces and Presentation (e.g. HCI)**]: User interfaces. - Graphical user interfaces.

General Terms

Design, Human Factors

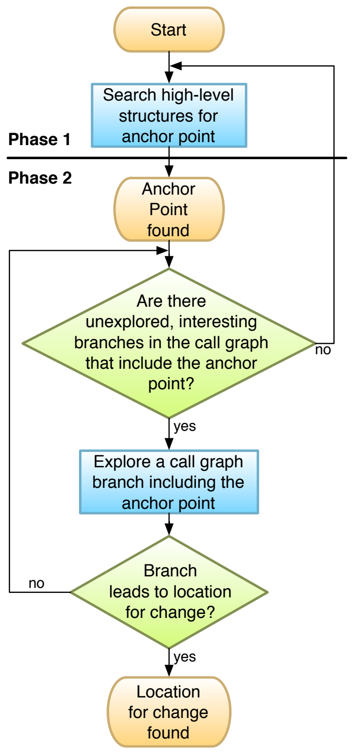

Start
Search high-level structures for anchor point
Phase 1
Phase 2
Anchor Point found
Are there unexplored, interesting branches in the call graph that include the anchor point?
no
yes
Explore a call graph branch including the anchor point
Branch leads to location for change?
no
yes
Location for change found

Figure 2: Developers intuitively explore source code in two phases. Figure taken from [2].

Introduction

Software maintenance is the task of fixing bugs, adding new features, and performing other modifications after software has been shipped. It accounts for up to 70% of the total expenses in a software project [7]. Successful software maintenance requires a modification of source code without interfering with its overall structure or introducing side effects. To do so, the developer has to thoroughly understand the code. While numerous tools have been proposed to support this task, software developers still consider comprehending unknown code one of their biggest problems [5].

A key activity in software maintenance is to navigate source code [8]. Among the various types of navigation, *call graph based navigation* is especially important to determine where to implement a change [2, 4, 10]. The call graph contains all methods as nodes, and each method is connected to its callees through outgoing edges and to its callers through incoming edges.

Previous studies [2, 3, 9] repeatedly document a *two-phase model* for navigation in the call graph (Fig. 2): Developers start by searching for an *anchor point* they consider interesting (Phase 1). Once they found an anchor point, they follow different paths starting from there until they reach the part of the code they need to work with (Phase 2). Developers seem to stick to this model intuitively, as it was observed in multiple, independent studies with different settings.

We introduce *Blaze* (Fig. 1), which specifically supports this model. Blaze always shows one path through the call graph next to the source code editor. During Phase 1, Blaze automatically updates to show a path containing the method currently being edited. Hence, Blaze provides additional information scent [6] during Phase 1. In Phase 2, Blaze can be locked to disable automatic updates and make sure the anchor point always remains on the displayed path. Thus, Blaze supports structured exploration of different paths involving the anchor point during Phase 2.

We compared Blaze to Stacksplorer [2], a previous research tool for call graph exploration, to a Call Hierarchy tool, similar to what is found in Eclipse[1], and an unmodified Xcode[2] installation in a quantitative study. Our preliminary results indicate that users can solve maintenance tasks significantly faster when using Blaze or Stacksplorer compared to the Call Hierarchy and Xcode.

Related Work

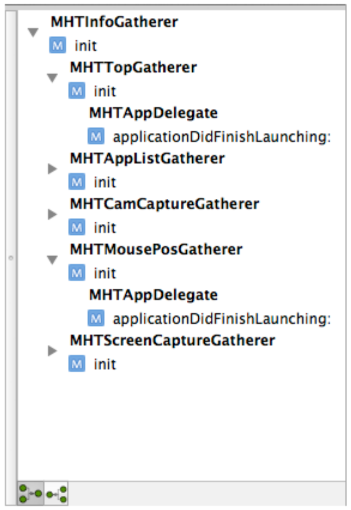
Navigation Behavior

In a study, Ko et al. [3] found that navigation accounted for 35% of the time developers needed to solve tasks concerned with a 500 SLOC[3] Java application using the widely adopted Eclipse IDE. Although powerful tools for call graph navigation are available in Eclipse, they were seldom used.

Latoza et al. [4] confirmed the importance of reachability questions, i.e., searches for feasible call graph paths, for software maintenance tasks. In a survey among 460 professional developers, all reported consistently that these questions have to be answered often (ten times a day or more), and over 80% of all participants considered these questions at least "somewhat hard" to answer.

Lawrence and Burnett [6] carried over results from information foraging theory, in which each link between two pieces of information has a certain *scent* that

[1]www.eclipse.org
[2]http://developer.apple.com/technologies/tools/
[3]SLOC: non-empty, non-comment source lines of code



```
MHTInfoGatherer
  M init
  MHTTopGatherer
    M init
    MHTAppDelegate
      M applicationDidFinishLaunching:
  MHTAppListGatherer
    M init
  MHTCamCaptureGatherer
    M init
  MHTMousePosGatherer
    M init
    MHTAppDelegate
      M applicationDidFinishLaunching:
  MHTScreenCaptureGatherer
    M init
```

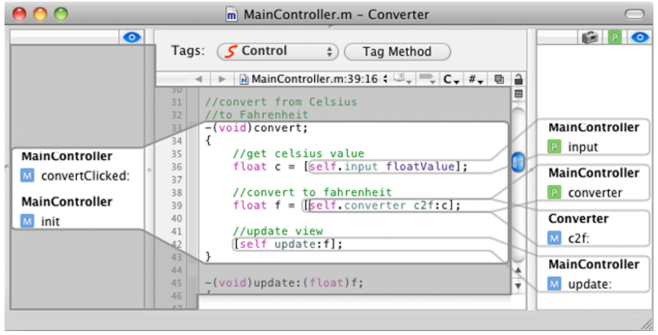
Figure 3: The screenshot shows the Call Hierarchy in caller view mode.

determines how likely a user is to follow this link while searching for a specific piece of information, to model where developers navigate to. In an experiment, the set of methods generated by their model mostly contained the methods actually visited by professional software developers while working on a bug-fixing task.

Call Graph Exploration Tools
Many current IDEs, for example Eclipse or Visual Studio, implement a Call Hierarchy tool that uses a tree view to show multiple levels of either callers or callees, starting from a given focus method as the root of the tree view. In Eclipse, expanding the children of an element in the tree view shows all callers or callees of this element. Two buttons are used to select whether children of an element are its callers or callees. To change the root node, every method identifier in the source code can be right-clicked to access the "Show Call Hierarchy..." context menu item. When callers are displayed, clicking a method opens it and highlights the call to the parent element. If the callee view is selected, clicking a method opens the parent method and highlights the call to the clicked method. For consistency, in both modes right-clicking an entry in the tree view and selecting the "Open" context menu item opens the clicked method without any highlights.

Code Bubbles [1], although not primarily a call graph exploration tool, simplifies navigation by changing the way code is laid out. It shows individual methods and auxiliary information in bubbles that can be arranged freely on a 2D plane. Arrows between bubbles indicate relationships between them, e.g., information about the call graph. This layout significantly reduced the need for back-and-forth navigation in a controlled experiment comparing Code Bubbles and Eclipse.

Stacksplorer [2] (Fig. 4) gives users access to the call graph neighborhood of a focus method, i.e., the method the user is currently working on. To show this information, two interactive views are added as side columns next to the source code editor. While the central source code editor shows the focus method in the usual way, the left side column shows a list of callers, and the right side column a list of callees. Clicking on any method in the side columns navigates to this method. Whenever the focus method changes in the central editor, the side columns update automatically. Optional graphical overlays connecting method calls in the source code with the corresponding entries in the side columns help making sense of the displayed information.



```
//convert from Celsius
//to Fahrenheit
-(void)convert;
{
    //get celsius value
    float c = [self.input floatValue];

    //convert to fahrenheit
    float f = [self.converter c2f:c];

    //update view
    [self update:f];
}

-(void)update:(float)f;
```

Figure 4: The left column in Stacksplorer lists all callers of the current focus method, the right column lists all callees.

Design of Blaze
The existing call graph exploration tools support the two-phase navigation model (Fig. 2) to varying degrees: The Call Hierarchy allows structured analysis of a sub-tree starting at a given anchor point, and thus supports primarily Phase 2. Stacksplorer primarily supports Phase 1 by providing additional information scent and direct

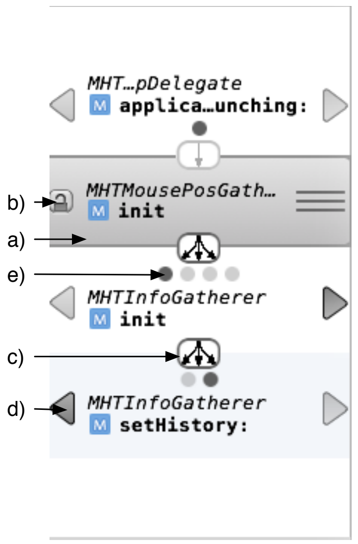


Figure 5: Blaze shows a single path through the call graph. Methods on the path are called from top to bottom.

navigation to the call graph neighborhood. Blaze is designed to support both phases.

Blaze adds an interactive view (Fig. 5) on the right side of the source code editor (Fig. 1). This view shows a single path through the call graph containing a focus method, which is displayed in gray (a)). During Phase 1, the focus method is *unlocked* and updates automatically to match the method the user is currently working on, i.e., when the cursor in the editor moves to a different method the focus method changes accordingly, and the displayed path is updated. Changes to the path are minimized, i.e., the path does not change at all if the developer navigates to a method that is already on the path. While the focus method is unlocked, Blaze provides important additional information scent, which is important during Phase 1.

During Phase 2, the focus method can be *locked* (b)) to prevent the focus method and the path displayed in Blaze from changing due to navigation in the editor. This makes finding an anchor point explicit. Blaze now allows developers to explore all possible paths involving the anchor point, i.e., the locked focus method.

For each entry on the path, several alternatives exist. Because the focus method has to remain on the path, each entry below the focus method can be exchanged with another callee of the preceding method; above the focus method, each entry can be exchanged with another caller of the following method. When an entry is exchanged, the following (below the focus method) or preceding (above the focus method) path changes accordingly. To exchange an entry on the path, a user can either click the arrows between two entries (c)) to reveal a list of all options, or use the arrows next to each entry (d)) to flip through the alternatives. The latter option is similar to turning discs on a combination lock. A "line of dots"-style page indicator (e)), similar to the one found on the iPhone's home screen, is displayed in each entry to show the number of options and the current selection. This interface contains a lot of information (as much as the Call Hierarchy tool), but consumes relatively little screen space because it never shows more than one single path at a time. It also allows for a very structured (depth-first) exploration of all possible paths involving the focus method.

To link the Blaze view to the source code editor, an overlay similar to those in Stacksplorer is shown, which connects the currently edited method in the editor to the corresponding entry in the side column (Fig. 1).

Study

We compared Blaze, Stacksplorer, the Call Hierarchy, and an unmodified Xcode installation to test if the call graph exploration tools can reduce the time spent on maintenance tasks. All tools were implemented as Xcode plugins using the same call graph parsing backend, so the tools only differed regarding the user interface. The Call Hierarchy was designed to resemble the behavior of the equally named tool in Eclipse as closely as possible. We carried out a between-subjects study with 35 subjects and four conditions (three call graph exploration tools and Xcode without any plug-ins). All participants were students, except for two professional software developers. On average, participants had 2.6 years ($SD = 2.1$) of experience with Objective-C. The assignment of participants to conditions was randomized. Two subjects were removed from the quantitative evaluation because of technical problems during the test, leaving us with 9 participants in the Call Hierarchy condition and 8 participants in each of the other conditions.

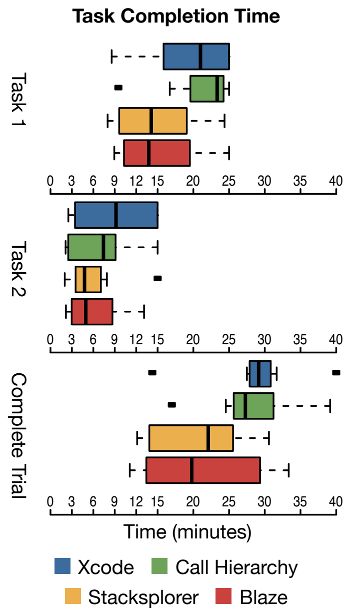


Figure 6: Task completion times by task and condition. Stacksplorer and Blaze outperformed Xcode and the Call Hierarchy for Task 1 and for the complete trial.

Participants had to work on two maintenance tasks in the source code of BibDesk, an open-source bibliography manager for Mac OS X (Rev. 17029, 80.000 SLOC, 400 classes). We used the same two subtasks that were used in the first task of the previous evaluation of Stacksplorer [2]. In both tasks no implementation was required. Task 1 was concerned with BibDesk's Autofile feature, which can automatically move PDF files of publications into a dedicated folder and rename them according to a naming scheme. Users were asked where a change had to be implemented to prepend a fixed string before the generated file name. In Task 2, we proposed one solution for Task 1 that changed the return value of a method. We asked users to identify side effects caused by this solution. Given participants found an optimal anchor point with the information we gave in the tasks, Task 1 required navigation along at least three edges in the call graph during Phase 2, Task 2 along at least two edges. For each task, we measured the completion time and the correctness of the solution.

Quantitative Results

Using any call graph exploration tool, more participants solved the complete trial (i.e., both tasks) successfully than with Xcode without any additional tools (Fisher's test, $p = 0.030$, one-sided).

After removing outliers (Fig. 6), the task completion times were compared using a planned contrast ANOVA. For the complete trial, the comparison shows a significant advantage of Stacksplorer and Blaze compared to Xcode and the Call Hierarchy ($p = 0.001$, $F(1,25) = 12.76$). The difference is also significant for Task 1 alone ($p = 0.004$, $F(1,28) = 9.95$), but not for Task 2 ($p = 0.111$, $F(1,27) = 2.71$). The boxplots in Figure 6 and our qualitative observations suggest a possible reason for this result: The Call Hierarchy might be useful specifically in Task 2, which dealt with the assessment of side effects of changes. We will explore this hypothesis in the future.

No significant differences in task completion time were found between Stacksplorer and Blaze (Complete Trial: $p = 0.882$, $F(1,25) = 0.02$). This result is particularly interesting because Blaze, which was designed to completely support the established two-phase navigation model, should aid developers better than Stacksplorer. We will analyze this effect in more depth in future work. One possible explanation for this result is that the advantage of Blaze offering explicit support for Phase 2 may be counterbalanced by Stacksplorer's superior support for Phase 1; having all choices of local navigation targets displayed at the same time might be an advantage during the initial code exploration.

Qualitative Results

All tools were used more frequently in Task 2. In the Blaze and Call Hierarchy conditions, where browsing the call graph is possible without switching the source code file in the editor, more than half of the participants made up theories entirely by browsing the call graph using the tool at hand. They called up methods in the source code editor only to verify their theories. This effect seems to be specific to Task 2, where a starting point was clearly given in the task description.

Automatic updates when users navigate in the editor and graphic connections between the tool and the source code, two features specific to Stacksplorer and Blaze, seem to be particularly important. When observing participants using the Call Hierarchy, we found that the lack of automatic updates led to frequent mode errors, because users forgot to open the method they discoverd using the editor in the

Call Hierarchy. Another common reason for mode errors in the Call Hierarchy condition was the explicit switch between caller and callee view modes. The visualizations of Blaze and Stacksplorer avoided these problems.

In the control condition (Xcode without additional tools), we observed that participants sometimes hesitated to navigate too far away from their anchor point in order not to lose it. Blaze and the Call Hierarchy solve this issue because they allow keeping an explicit reference to the focus method around.

Future Work

We are planning to study in more detail why Blaze was no significant improvement over Stacksplorer. For that, we will analyze the actual navigation paths from our video recordings of the study sessions to see how they were influenced by the tools used.

We also want to identify the influence of individual design aspects on the usefulness of a tool. Possible design aspects to look at are, e.g., whether a tool auto-updates and whether the tool is visually integrated with the source code editor. Once the beneficial properties of both designs are identified, we will try to combine them in an improved version of Blaze.

Acknowledgements

This work was funded in part by the German B-IT Foundation and by the German Government through its UMIC Excellence Cluster at RWTH Aachen University.

References

[1] 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.

[2] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers. Stacksplorer: Call graph navigation helps increasing code maintenance efficiency. In *Proc. UIST '11*. ACM, 2011.

[3] 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), 2006.

[4] T. D. LaToza and B. A. Myers. Developers Ask Reachability Questions. In *Proc. ICSE '10*. ACM, 2010.

[5] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *Proc. ICSE '06*. ACM, 2006.

[6] 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, 2008.

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

[8] 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.

[9] 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), 2008.

[10] T. Winograd. Breaking the complexity barrier again. *ACM SIGIR Forum*, 1974.