How Tools in IDEs Shape Developers' Navigation Behavior

Jan-Peter Krämer, Thorsten Karrer, Joachim Kurz, Moritz Wittenhagen, Jan Borchers
RWTH Aachen University
52062 Aachen, Germany
{kraemer, karrer, kurz, wittenhagen, borchers}@cs.rwth-aachen.de

ABSTRACT

Understanding source code is crucial for successful software maintenance, and navigating the call graph is especially helpful to understand source code [12]. We compared maintenance performance across four different development environments: an IDE without any call graph exploration tool, a Call Hierarchy tool as found in Eclipse, and the tools Stacksplorer [7] and Blaze [11]. Using any of the call graph exploration tools more developers could solve certain maintenance tasks correctly. Only Stacksplorer and Blaze, however, were also able to decrease task completion times, although the Call Hierarchy offers access to a larger part of the call graph. To investigate if this result was caused by a change in navigation behavior between the tools, we used a set of predictive models to create formally comparable descriptions of programmer navigation. The results suggest that the decrease in task completion times has been caused by Stacksplorer and Blaze promoting call graph navigation more than the Call Hierarchy tool.

Author Keywords

Development Tools / Toolkits / Programming Environments; Analysis Methods

ACM Classification Keywords

H.5.2. Information Interfaces and Presentation (e.g. HCI): User Interfaces

INTRODUCTION

Software maintenance, the process of fixing bugs or performing other modifications after the software has been released, accounts for up to 70% of the total expenses in a typical software project [19]. Since it requires developers to modify the source code without introducing side effects or otherwise interfering with its structure, they need to gather knowledge about the program. For example, they have to find out which methods are responsible for a given feature, and how methods rely on each other. Despite the efforts to support source code comprehension with modern development environments, developers still consider comprehending source code one of their biggest problems [14].

CHI 2013, April 27–May 2, 2013, Paris, France.

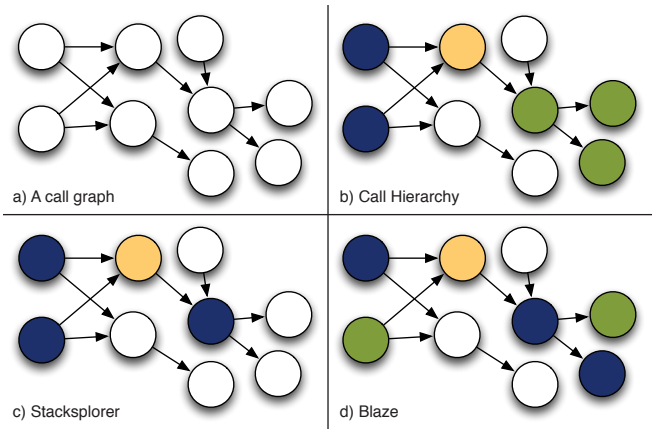

Figure 1. a) An example call graph. Each node represents one method, edges point to callees. b) With the light orange method being the focus method the Call Hierarchy allows browsing either the medium green or the dark blue subtree (including the focus method). c) Stacksplorer shows the neighborhood of the focus method. d) Blaze shows one path including the focus method (dark blue nodes); medium green nodes are options for alternative paths.

The process of navigating through source code has been found to be especially important for developers to build up their mental model of the application [20, 23]. This is particularly true for navigation along the call graph [12, 25]. The call graph represents methods as nodes, and an edge from a to b means that method a, the *caller*, calls method b, the *callee*.

The call graph for many real world software projects, however, is a complex structure that cannot easily be visualized completely. Tools to support call graph exploration thus have to restrict visualization to a subgraph. Most tools let the developer pick a *focus method* and only visualize subgraphs including this focus method. The most widespread tool for call graph exploration is the Call Hierarchy, which is, for example, found in the Eclipse IDE[1]. The subgraph visualized in the Call Hierarchy is a tree with its root at the focus method; the developer can choose if this tree is built up in caller or callee direction. We previously proposed two alternative tools for call graph exploration that offer access to a smaller subgraph than the Call Hierarchy: In Stacksplorer [7], the visualized subgraph is the direct neighborhood of the focus method.

[1] http://www.eclipse.org/

Blaze [11] visualizes a single path through the focus method. The three different approaches are visualized in Figure 1.

In the first half of this paper, we show that developers using Stacksplorer and Blaze are faster in solving certain maintenance tasks than developers using the Call Hierarchy. This result is unexpected because the Call Hierarchy facilitates access to a superset of the methods visible in Stacksplorer or Blaze. In the second half of this paper, we analyze if this result can be explained by changes in the navigation behavior of developers caused by using the different tools. This is done by describing developers' navigation histories using a set of seven quantifiable features, where each feature is determined by calculating how well the navigation can be predicted by a stereotypic navigation model. These descriptions are then compared between the tools.

Thus, this paper makes the following contributions: (1) We present a study comparing the three call graph exploration techniques described above to an IDE without any dedicated call graph navigation tool. (2) We introduce an analysis technique to describe navigation behavior based on predictive models. (3) We apply this technique to offer possible explanations for the performance differences between the call graph exploration tools.

RELATED WORK

Navigation Strategies

Ko et al. [9] found that navigation accounted for 35% of the time developers needed to perform tasks within a 500SLOC[2] Java application using the Eclipse IDE. The powerful navigation tools available in Eclipse were rarely used. Similar results were found by Murphy et al. [16] when analyzing Eclipse usage logs from 41 Java developers.

LaToza et al. [12] found that one of the most important questions developers ask are reachability questions, i.e., searches for feasible control flow paths. They surveyed 460 professional software developers, who consistently reported those questions were at least "somewhat hard" to answer and came up ten times a day or more.

Previous studies [7, 9, 21] repeatedly describe strategies for navigation in the call graph by a *two-phase model*. In the first phase, developers search for an anchor point they consider interesting. In the second phase, they explore different paths starting from the anchor point until they found the relevant location in the source code.

Lawrence et al. [15] carried over results from information foraging theory [18] to model how developers navigate. Each link to a piece of source code has a certain scent that determines how likely a developer will follow the link when searching for specific information. The model incorporates structural aspects of the source code as well as linguistic similarity of source code to a bug report. Predictions from this model were on par with predictions generated from historical navigation data recorded from actual developers.

[2]SLOC: Non-comment, non-empty lines of source code

Navigation Tools

Recommender tools use models of navigation behavior to constantly predict and show the methods the developer will most likely navigate to in order to make them more easily accessible. These tools calculate a *degree of interest* (DOI) for all methods in a project and show navigation shortcuts to those with the highest DOI in a list. DOI may be determined by factors such as the reading and editing history or textual similarity to information from version control systems [4, 8, 22, 24]. For all of these tools, controlled experiments revealed a significantly reduced navigation effort.

REACHER [13] can restrict searches to reachable branches of the call graph originating in or leading to a specific method. The relevant portions of the connection between the method and the search result are displayed graphically. A study showed significantly increased success rates among developers using REACHER compared to those using Eclipse in six tasks involving reachability questions. In contrast to the tools we compare, REACHER is a search tool, requiring the developer to formulate a query.

The Whyline [10] allows to determine the cause of certain aspects of an application's graphical and textual output. Users can, for example, formulate the question "Why did this circle's color = blue" about a circle drawn in the interface. The Whyline then computes a dynamic slice from a runtime trace of the application, i.e., it computes which methods influenced the relevant property. This dynamic slice is presented as a graph. The tool allowed novice programmers to fix a bug significantly faster than expert developers not using the tool. In contrast to the navigation tools we compare in this paper, the Whyline is a dedicated debugging tool.

Code Bubbles [2] and Code Canvas [5] introduced IDE concepts in which source code is laid out by arranging pieces of information, such as individual methods or bug reports, in bubbles on a 2D plane. Bubbles are connected to indicate relationships between two items, such as a method call connecting two methods. This layout greatly simplifies glancing at related methods after they have been first visited and opened in a bubble.

REACHER, the Whyline, Code Bubbles, and Code Canvas all introduce new possibilities to explore the call graph. In contrast, we compare call graph navigation tools that visualize a subgraph of the call graph and that extend an existing IDE. This subgraph depends on a focus method the user has to select in some way. A comparison of these tools with the ones presented above is an interesting endeavor for future work.

CALL GRAPH NAVIGATION TOOLS

In our first experiment, we compare three call graph navigation tools in terms of how they impact participants' performance and navigation behavior: a Call Hierarchy tool, Stacksplorer, and Blaze. The tools differ in which subgraph of the call graph they visualize (Figure 1), how the displayed subgraph can be changed, and how developers can navigate using the tool. All tools were implemented as plug-ins for Apple's Xcode 3 IDE and were using the same backend for code analysis and call graph creation.

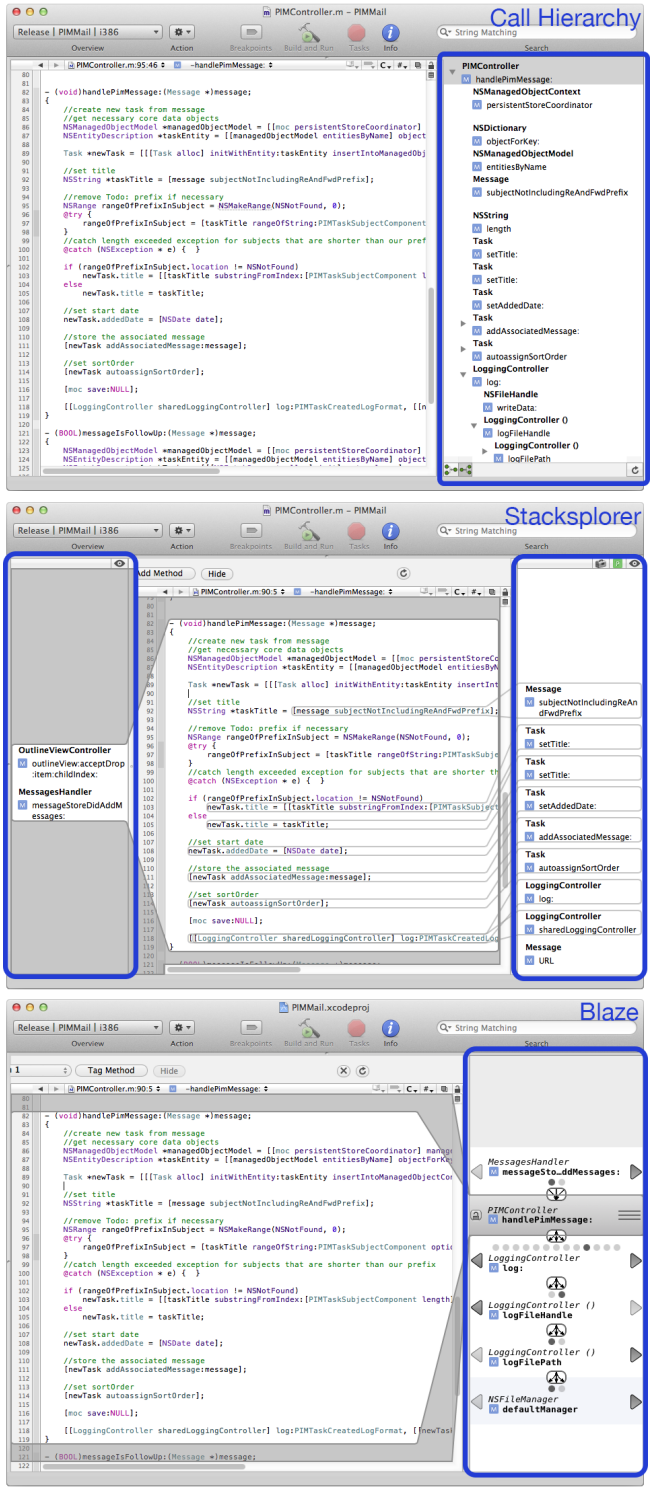

Figure 2. The screenshots show how each tool is presented in Xcode. The user interface elements circled in blue are provided by the tool and are not part of a standard Xcode installation.

Call Hierarchy

Our Call Hierarchy plug-in (see top of Figure 2) closely resembles the equally named tool in Eclipse. The user starts by selecting a *root method* (the light orange node in Figure 1b) for a tree view. Expanding any element in the tree view shows its callers (the dark blue nodes in Figure 1b) or callees (the medium green nodes in Figure 1b), depending on the mode the Call Hierarchy tool is in. The mode can be changed using two buttons at the bottom. If a method appears multiple times in the list of callers or callees of a method, it is displayed only once. In callee view mode, children of an element are ordered in the same way they appear in the implementation of the element; in caller view mode the order is arbitrary.

The root method is changed by selecting the "Show Call Hierarchy..." command from the context menu for a method identifier anywhere in the source code. To navigate, users can click on the elements in the tree view. If callers are shown, this opens the method in the editor and the call to the parent method is highlighted. In case callees are shown, clicking a method opens its parent method and the call to the clicked method is highlighted. Consistently across modes, the "Open" command in the context menu of an element in the tree view opens this element in the editor.

Our Call Hierarchy tool does not detect cycles in the call graph, but our user study avoided such cycles. Xcode does not allow users to freely place tools in the interface, so our Call Hierarchy is always displayed on the right side of the source code editor. This allows a fair comparison in terms of screen real estate to the other tools that both reduce the width but not the height of the editor.

Stacksplorer

Stacksplorer [7] (see middle of Figure 2) visualizes the call graph neighborhood of the *focus method* (the blue nodes in Figure 1c). It shows one interactive side column view on each side of the source code editor. The left column shows a list of callers of the focus method, the right column shows a list of callees. In Stacksplorer, the focus method is always synchronized to the method in which the cursor is placed in the central source code editor. Because Stacksplorer occupies two side columns, it takes up more space than the other tools.

Clicking any method in one of the side columns opens it in the central editor. This implicitly also causes the focus method to change and the side columns to update. The focus method also changes when the user navigates to a different method in the central editor by other means. Thus, even when the developer is not exploring the call graph, the automatically-updated side columns provide auxiliary information.

Like in the Call Hierarchy tool, callees in the right column are sorted by their order of appearance in the source code, but Stacksplorer shows the same method more than once if it is called more than once. The side columns' content scrolls automatically to keep the on-screen distance to the related code minimal. Additional overlays can be turned on, which connect an entry in the side column with the corresponding method call in the source code. This is especially helpful for densely written code, e.g., nested method calls.

Blaze

Blaze [11] (see bottom of Figure 2) implements depth-first call graph exploration, and shows one path through the call graph including the focus method (the blue nodes in Figure 1d). Blaze shows all methods on this path in a view at the right side of the source code editor. The path is displayed top-to-bottom, i.e., each entry calls the one immediately below. As in Stacksplorer, clicking any method in the Blaze column navigates to this method.

To change which path through the call graph is displayed, Blaze uses a combination lock metaphor: For each entry in the path, several alternatives exist (the medium green nodes in Figure 1d). Because the focus method has to be part of 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 arrow between two entries to reveal a list of all options, or use the arrows next to each entry. A line of dots is displayed in each entry to show the number of possible options and the current selection.

As in Stacksplorer, the focus method in Blaze is automatically synchronized to the method the user is currently working on. When the focus method changes, the displayed path is updated accordingly, but changes are kept minimal, i.e., if the developer navigates to a method that is already visible on the path, the path does not change at all. Optionally, Blaze can be locked to prevent automatic updates to the focus method. To keep users aware of their location on the path, an overlay is shown that connects the currently edited method in the editor to the corresponding entry in the side column.

The two states Blaze supports (locked and unlocked) can be mapped to the two-phase navigation model described before. While Blaze is unlocked, the automatically displayed path is additional auxiliary information that might help finding an anchor point during the first phase. When Blaze is locked with the anchor point being the focus method, it allows to browse all paths involving the anchor point in the second phase.

STUDY SETUP

We analyzed the data from two previous studies [7, 11], of participants working on maintenance tasks for BibDesk[3], an open-source bibliography manager for Mac OS X. Participants were given a typical maintenance task: firstly, they had to identify a location for a change (task 1), and secondly, they had to identify possible side effects (task 2). The first task concerned BibDesk's Autofile feature, which automatically moves and renames PDF files according to a user specified naming convention. Subjects were asked to change the feature so that it would prepend a fixed string to the name regardless of the specified naming convention. To successfully solve this task, the participant had to suggest a modification that would have achieved the intended effect. Because we were only interested in how developers *navigate* between methods for finding a change location, implementing the change was not required. The second task required finding a side effect introduced by a given solution for the first task.

For both tasks, an expert judged a solution as correct when the suggested change would lead to the effect described in the task. We saw no non-standard solution that would have required more effort to be verified. The unsuccessful participants either could not complete the task in time or presented solutions that would clearly not have the desired effect. The complete maintenance task was considered correct if both individual tasks were solved correctly. In [7], the tasks were referred to as tasks 1.1 and 1.2.

We recruited 33 subjects—31 students and two professional software developers—for our study. On average, participants were 26.3 years old ($SD = 2.6$), spent an average of 12.6 hours ($SD = 11.6$) on programming per week, and had an average of 2.6 years ($SD = 2.1$) of experience with Objective-C. A minimum of half a year of experience with Objective-C was required to participate in the study. No participant had seen the BibDesk source code before, although half of them had used BibDesk before.

Participants were randomly assigned to one of four conditions: *Xcode* (XC), the control condition, used the unmodified interface of Apple's Xcode 3 IDE. It includes no dedicated call graph navigation tool, but users can navigate to a callee from within the editor using the "Jump to definition" context menu command for a method call. *Call Hierarchy* (CH), *Stacksplorer* (SP), and *Blaze* (BL), the experimental conditions, used the same version of Xcode but extended with the respective plug-in. Each condition was similarly sampled in terms of number of participants (XC: 8, CH: 9, SP: 8, BL: 8), programming experience (in years), and coding done per week. In the three experimental conditions, the session started with a brief introduction to the tool using an unrelated code base. To make sure not to bias participants, the experimenter did not tell the participants if the tools were designed by us or not.

After the introduction, participants were allowed to familiarize themselves with the BibDesk project for up to 10 minutes. Then, we handed tasks 1 and 2 to the participants one task at a time, so that while working on task 1 they would not be aware of task 2 yet. Time to finish the tasks was limited to 25 minutes for task 1 and 15 minutes for task 2. We encouraged participants to work quickly but to make sure to arrive at a correct solution, as they had only one chance to provide an answer. When exceeding the time limit or giving an incorrect answer, we used the respective time limit as the task completion time.

Consistent with previous studies [2, 20], using runtime analysis tools such as the debugger was prohibited. However, participants were allowed to run a compiled executable of the application.

PERFORMANCE COMPARISON

Firstly, we compared success rates between the different tools. We found a significant increase in success rates for the

[3]Rev. 17029, Objective-C, 80.000SLOC

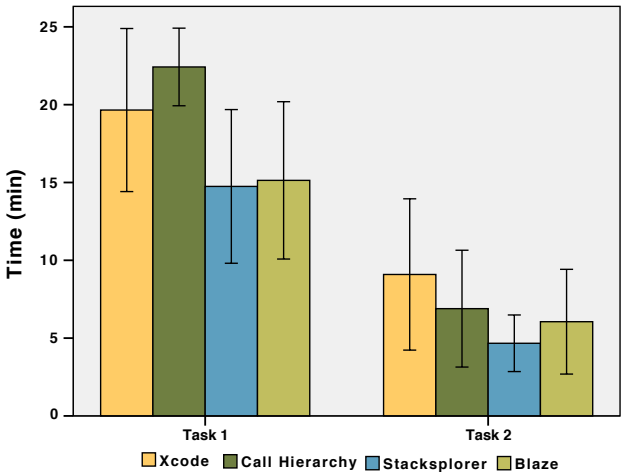

Figure 3. The graph shows task completion times per condition and 95% confidence intervals. Task and condition both have a significant effect on task completion times.

complete maintenance task between the experimental conditions and the control (one-sided Fisher's exact test: $p = 0.015$). No significant differences in terms of success rates were found between the experimental conditions.

Next, we compared the task completion times and assumed to find similar results. An ANOVA[4] revealed a significant effect of task and of condition.

Task:	$F(1, 29) = 65.281$	$p < 0.001$	$\eta^2 = 0.666$
Condition:	$F(3, 29) = 3.720$	$p = 0.022$	$\eta^2 = 0.278$

Using a post-hoc Dunnett t-test we found that task completion times in the Stacksplorer and Blaze conditions were significantly lower than in the control condition, but those in the Call Hierarchy condition were not. A post-hoc Tukey's test comparing Blaze and Stacksplorer was not significant.

XC vs.	CH: $p = 0.662$	ST: $p = 0.038$	BL: $p = 0.020$
ST vs.	BL: $p = 0.992$		

In summary, call graph exploration tools have a positive impact on the success of a typical code maintenance task. Stacksplorer and Blaze also have a significant effect on task completion times, while the Call Hierarchy unexpectedly does not. Blaze and Stacksplorer perform similarly although they implement different approaches for presenting a subgraph of the call graph to the user.

MODEL-BASED ANALYSIS OF NAVIGATION BEHAVIOR

The comparison shows that performance differences between the tools exist. We argue that there are two possible reasons for these differences: Either the user interface of the more successful tools was better or easier to use, or the tools encouraged different navigation strategies, which then caused the differences in efficiency. In the following we focus on the latter.

[4]All ANOVAs carried out are Repeated-Measures ANOVAs, where task is a within-groups factor and condition is a between-groups factor.

To compare the different navigation strategies, we first need a consistent way to formally describe navigation. This description should be independent of the tool and development environment used. We therefore propose to characterize navigation behaviors by a set of quantifiable features, where each feature represents the degree to which the navigation behavior conforms to one of a set of well known micro navigation patterns. To measure a feature for a recorded session, we use the navigation models compared by Piorkowski et al. [17]: Since these models all predict navigation targets according to different micro strategies of navigating source code, the prediction accuracy of each model is a quantitative indicator for how well the overall navigation behavior of the session resonates with these micro strategies. We thus use these prediction accuracies as our features.

Note that, while originally Piorkowski et al. tried to find the most accurate model for predicting developers' navigation, we do *not* evaluate the models in terms of their prediction accuracies, but we characterize the different navigation tools in terms of their effect on the individual prediction accuracies of all models.

Formally, the models as described by Piorkowski et al. [17] assume that navigation in a single study session is coded as a sequence of visited methods $H = (m_1, m_2, ..., m_n)$ where $\forall m_i, m_{i+1} \in H : m_i \neq m_{i+1}$. Using the navigation sequence up to an element m_j as input, the models try to predict m_{j+1}.

To do so, they calculate the probability that a developer navigates to a method for all methods in $M_j - \{m_j\}$, where M_j is a set of methods approximating all methods known to the developer and comprises all methods in files that have been opened so far or that have been visible in call graph exploration tools or search results.

Models calculate their prediction by creating an activation function $A_j : M_j - \{m_j\} \mapsto \mathbb{R}$, with higher activation values indicating a higher probability of the developer navigating to a method. Then, a ranking function $R_j : M_j - \{m_j\} \mapsto \mathbb{N}$ is obtained by rank-transforming A_j. For methods with the same activation value, the average of all involved ranks is used. All models share a parameter N that determines how many of the top ranked methods are returned by the model. If more than N methods are assigned the highest available rank, the models do not predict anything. Characteristic for each model is the definition of A_j.

We will provide only a qualitative description of A_j for the different models here, and explain which navigational "micro-pattern" we think is represented by each model. For the formal definition of each model please refer to [17].

Recency assigns higher activation values if a method has been visited more recently. It correlates with navigating back and forth between related methods to understand their connection, which was shown to be a common and important pattern in [9].

Frequency assigns higher activation values the more frequently a method has been visited. It correlates with go-

ing back frequently to very important methods, such as the anchor point found in the two-phase navigation model.

Working Set assigns an activation of 1 to all methods visited during the last δ navigation steps and 0 to all other methods. δ is an estimate of the size of the working set, in our analysis we used $\delta = N$. The Working Set model is similar to Recency, but implies that there is a fixed-sized set of methods that are particularly important to the task, as suggested in [3].

Bug Report Similarity assigns a method m the tf-idf weight [1] of the bug report compared to the words in m. Before calculating the tf-idf weight, stop words are removed and camelCase identifiers are split apart. The Bug Report Similarity model correlates with the micro-pattern of searching for locations in the source code based on textual clues in the bug report, which was reported previously in [15].

The following models all maintain a graph G containing the methods in M_j as nodes. $A_j(m)$ is calculated inversely proportional to the distance between m and m_{j-1}. The models differ in what edges are included in G.

Within-File Distance adds an undirected edge between two nodes if they are adjacent in a source file document. It correlates with scrolling in a file, which is commonly used to explore the file based decomposition of the software [20].

Forward Call Depth adds a directed edge between two nodes m_a, m_b if m_b is called from the implementation of m_a. It correlates with navigation to callees, which is possible in many IDEs even without the use of a dedicated tool using the "Jump to definition..." command from the context menu for a method call in the editor.

Undirected Call Depth is a modified variant of Forward Call Depth with directed edges being replaced by undirected edges. This model correlates with the call graph navigation supported by the call graph exploration tools we compared.

Analysis of Navigation Behavior

Navigation events were coded manually in video recordings of the user study sessions using ChronoViz [6], annotating all clicks on UI elements that lead to changes in the source code editor as well as navigations via text search and scrolling. User actions that were less than 0.5s apart were consolidated into one navigation action, e.g., clicking the back button twice. For every navigation, the tool used for navigation and the target of the navigation action were recorded. Targets of navigations are methods, unless the target method could not be clearly determined. If the target method was unclear, e.g., after opening a file, either a set of possible target methods was recorded or a more abstract target, such as the file, was annotated as navigation target.

A model prediction was counted as correct if the next method m_{j+1} was contained in the set of suggestions returned by the model. When the navigation led to a set of methods, a model predicted this navigation correctly if it predicted any method in the set; when navigating to anything else than a method or a set of methods a prediction could not be correct.

We compare prediction accuracy of the models for different prediction list sizes $1 \leq N \leq 20$. Model prediction accuracies for different N and the different models are depicted in Figure 4. In the following, we will analyze one model at a time. Statistical tests were performed only for $N = 1$, $N = 10$, and $N = 20$, results of which are listed in Table 1.

Frequency

For the Frequency model we expected to see a lower prediction accuracy in the more successful tools Stacksplorer and Blaze, because previous studies [9] showed the importance of back and forth navigation to gather contextual information but also that it requires a lot of time.

There is a significant effect of condition on the prediction accuracy of the Frequency model for $N = 10$. A post-hoc Tukey test only shows a significantly higher prediction accuracy in the Stacksplorer condition than in the Call Hierarchy condition ($p = 0.018$). There is no evidence that this increased number of revisits did decrease the number of distinct methods visited throughout the session ($p = 0.815$).

Stacksplorer allows, in contrast to our assumption, to perform more navigation to previously viewed methods but fast enough to still save time compared to the control condition and the Call Hierarchy. This indicates that frequent revisits to previously explored methods do not necessarily slow down the process of understanding source code. One explanation for that might be found in the two-phase navigation model, which states that revisits occur when backtracking to the focus method. In Stacksplorer, this often happens using the "Back" button in Xcode, because the focus method is not explicitly stored as in Blaze or the Call Hierarchy.

Recency & Working Set

We expected to see no effect of the condition on the prediction accuracy of the Recency and the very similar Working Set model. Methods might be included in a working set for a variety of reasons, not only because they are connected in the call graph, which is what a call graph exploration tool would support.

No effect of condition was found for the Recency model. We found a significant effect of condition, though, for the Working Set model for $N = 10$. Here again, a difference exists between Stacksplorer and the Call Hierarchy (Tukey test, $p = 0.017$).

Together with the previous result that the number of revisits was higher when using Stacksplorer, we can conclude that developers using Stacksplorer performed longer exploration phases in rather limited subsets of methods (likely connected by the call graph). Because they were also faster in solving the tasks than participants using Xcode alone or the Call Hierarchy, this result supports previous results by Sillito et al. [21], pointing out the importance of thoroughly understanding closed subsets of related methods.

For all conditions, prediction accuracy in the Recency and Working Set model becomes constant roughly for $N > 10$. This can be interpreted as an estimate about the maximum size for a working set for our tasks.

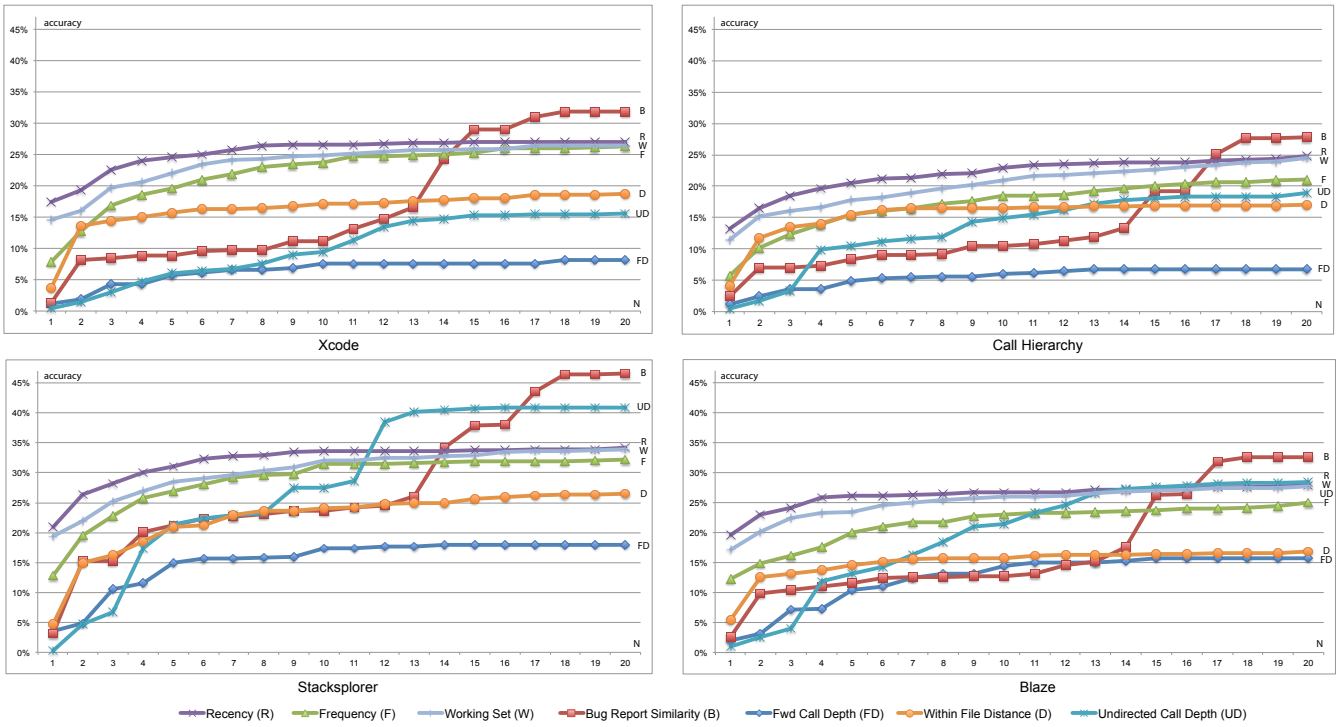

Figure 4. Plots of prediction accuracy against prediction list sizes N per condition for all models. When using a call graph navigation tool, Fwd Call Depth and Undirected Call Depth improve considerably.

Bug Report Similarity

We assumed that call graph navigation tools have no effect on the prediction accuracy of the Bug Report Similarity model, because the call graph, from which all navigation targets available in the tools are taken, does not provide any information about the textual content of the methods.

The prediction accuracy of the Bug Report Similarity model changes significantly with condition for $N = 20$. Here, the most important differences exist between Stacksplorer and both the Call Hierarchy and the control condition (Post-hoc Tukey's test, XC: $p = 0.035$, CH: $p = 0.004$).

One possible explanation for this result is that Stacksplorer, in contrast to the Call Hierarchy, automatically updates the list of navigation targets from which developers then preferably select textually similar navigation targets [15]. Of course, the degree to which the call graph neighborhood contains textually similar navigation targets depends on the source code.

Within-File Distance

In the BibDesk source code, as in most object-oriented source code, one file implements one class. Hence, by scrolling through a file, the different methods comprising the class can be explored. This structure of methods belonging to classes is orthogonal to the call graph, and consequently we assumed navigation behavior through this hierarchy would not be influenced by any call graph exploration tool.

No significant effect of condition was found for the Within-File Distance model. This result matched our expectations.

Forward Call Depth & Undirected Call Depth

We did expect an effect of the condition on the Call Depth models. Navigation along the call graph is what all tested tools specifically support.

This was confirmed; significant effects of condition are found for $N = 10$ and $N = 20$ for both the Forward Call Depth model and the Undirected Call Depth model. For the Forward Call Depth model, post-hoc tests (for $N = 20$) reveal significantly higher accuracy for Stacksplorer and Blaze compared to the control condition (one-sided Dunnett t-test, ST: $p = 0.004$, BL: $p = 0.022$) and compared to the Call Hierarchy (Tukey's test, ST: $p = 0.003$, BL: $p = 0.022$). Similar results are obtained for the Undirected Call Graph model, however, here no significant difference between Blaze and the Call Hierarchy was found in post-hoc tests. Over all tests, we did not find differences in prediction accuracy between the Call Hierarchy and Xcode.

In all call graph navigation tool conditions, accuracy of the Call Depth models increases substantially for $N > 6$. This can be explained by the average neighborhood size of a method. Among the 277 methods visited in all sessions, the average number for callers or callees is 1.81 ($SD = 5.93$) and 3.49 ($SD = 4.63$), respectively. Considering only methods that have been visited by at least half of the participants, the averages are even higher (callers: 3.33 ($SD = 2.58$), callees: 11.33 ($SD = 6.83$)). The Call Depth models do rank all neighbors (or callees in case of the Forward Call Depth model) of a method equally, so if there are more than N neighbors (or callees) they predict nothing.

		$N=1$		$N=10$		$N=20$	
		F	p	F	p	F	p
Frequency	T	27.13	.001	28.60	.001	18.31	.001
	C	2.729	.062	3.384	**.031**	2.482	.081
	I	1.733	.182	.384	.766	.813	.497
Recency	T	27.82	.001	11.64	.002	9.215	.005
	C	2.4	.088	2.728	.062	2.009	.122
	I	.696	.562	.793	.508	1.248	.311
Working Set	T	22.28	.001	14.49	.001	9.518	.004
	C	2.757	.06	3.432	**.030**	2.222	.107
	I	.823	.492	1.559	.221	1.124	.356
Bug Report Similarity	T	3.8	.061	36.88	.001	142.3	.001
	C	.366	.778	3.056	**.044**	5.279	**.005**
	I	.182	.908	2.784	.059	1.681	.193
Within-File Distance	T	39.09	.001	11.30	.002	13.55	.001
	C	.679	.572	1.178	.335	1.682	.193
	I	1.269	.303	.914	.447	.561	.645
Forward Call Depth	T	.048	.828	.548	.465	0.299	.589
	C	.1.688	.191	6.470	**.002**	7.023	**.001**
	I	.334	.801	1.693	.190	1.771	.175
Undirected Call Depth	T	5.969	.021	6.000	.021	5.395	.027
	C	.857	.474	5.791	**.003**	9.514	**.001**
	I	.125	.944	2.141	.117	5.344	.005

Table 1. The table shows comparisons between prediction accuracies of each model. The factors analyzed are task (T), condition (C), and their interaction (I). For task $df = 1$, for condition and interaction $df = 3$, for error $df = 29$. All significant effects of condition are in bold face.

For all tools, roughly two thirds of call graph navigations were performed using the tool and not via means existing in Xcode.

CH: $M = 68.2\%$ $SD = 37.1\%$
ST: $M = 65.0\%$ $SD = 31.2\%$
BL: $M = 65.0\%$ $SD = 39.8\%$

An ANOVA, however, reveals a significant effect of task and condition on the percentage of navigations that happened along an edge in the call graph, and a significant interaction.

Task: $F(1, 29) = 10.892$ $p = 0.003$ $\eta^2 = 0.211$
Condition: $F(3, 29) = 11.002$ $p < 0.001$ $\eta^2 = 0.532$
Interaction: $F(3, 29) = 3.877$ $p = 0.019$ $\eta^2 = 0.226$

A post-hoc Tukey's test shows a significantly higher percentage in the Stacksplorer condition as in all other conditions; a Dunnett test additionally shows significantly higher percentage in the Blaze condition but not in the Call Hierarchy condition compared to the control condition.

Tukey's: ST vs. XC: $p < 0.001$ CH: $p < 0.001$ BL: $p = 0.023$
Dunnett: XC vs. CH: $p = 0.602$ BL: $p = 0.036$

These results indicate that Xcode and the Call Hierarchy promoted call graph navigation similarly, Blaze did so significantly more and Stacksplorer again more than Blaze.

Differences in how the tools were used also show up when looking at the average length of a call graph navigation sequence. A call graph navigation sequence is a subsequence $S = (m_m, ..., m_n)$ of H, such that for all $m \leq i < n$ m_i and m_{i+1} are connected in the call graph. The condition has a significant effect on the average length of these sequences.

Condition: $F(3, 29) = 5.819$ $p = 0.003$ $\eta^2 = 0.376$
No significant effect of task, no interaction.

Post-hoc one-sided Dunnett t-tests revealed significantly longer sequences in the Stacksplorer and Blaze conditions but not in the Call Hierarchy condition when compared to Xcode.

XC vs. CH: $p = 0.451$ ST: $p = 0.002$ BL: $p = 0.009$

Blaze and the Call Hierarchy allow call-graph navigation along more than one edge at a time. When accommodating for that by allowing navigations back to a method previously visited in the sequence, results do not change. The reason is that, despite the possibility to navigate in that manner, these navigations only occurred 9 times over all sessions.

When no call graph navigation tool is used, developers seem to exhibit other strategies to find the desired information, one of which is using the project wide search. There is a significant effect of condition on the percentage of navigations performed using the project wide search ($F(3, 29) = 9.487$, $p < 0.001$, no effect of task). Post-hoc one-sided Dunnett t-tests show that the project wide search was used significantly more in the control condition than in all other conditions ($p < 0.001$ for all conditions). The project wide search in Xcode is often used to navigate to callers of a method by searching for the method name. This workaround to access callers is slow to invoke and error prone because of similarly named methods. Consequently, people stopped using this technique when dedicated tools were available.

We can conclude that Stacksplorer and Blaze can promote navigation along the call graph effectively. But the opportunity to shape navigation behavior by making additional navigation targets available seems to be limited: The option to navigate along multiple edges of the call graph at once, as it is offered by Blaze and the Call Hierarchy, was seldom used. We assume that backtracking multiple edges in the call graph at once is cognitively too challenging and developers fear to get lost in relatively unknown source code. However, this may be different when developers are familiar with the source code.

Comparing the three call graph exploration tools we tested, we find that Stacksplorer, which provides access to just the neighborhood of the focus method and synchronizes the focus method to the method currently being edited, promotes call graph navigation the most. Blaze still encourages more call graph navigation than the Call Hierarchy or Xcode alone. With Blaze, developers can perform some call graph exploration completely within the tool, without the need to navigate to each method on the path. This might explain why we observed less call graph navigation being performed using Blaze than using Stacksplorer.

Both Stacksplorer and Blaze present meaningful parts of the call graph: The direct call graph neighborhood usually contains very closely related methods; a single path through the call graph is familiar to many developers, e.g., from call stacks in a debugger. Unrestricted exploration as it is possible in the Call Hierarchy seems to be overwhelming and could

not substantially change how developers explore the source code. It was, however, successful in replacing the cumbersome project wide searches to find callers of a method, which were utilized in the Xcode condition.

LIMITATIONS

UI Differences

We cannot clearly differentiate between the effects caused by the tools exposing different parts of the call graph or by the presentation of this information. Stacksplorer and Blaze have user interfaces specifically designed for the respective exploration strategy; the Call Hierarchy was designed to be comparable with currently existing tools.

One of the most obvious differences between Blaze and Stacksplorer and the Call Hierarchy are automatic updates to the focus method. With automatic updates, Blaze and Stacksplorer present additional information with no interaction required. In post-session interviews half of the participants in the Call Hierarchy condition suggested to add automatic updates, even though they were not aware of the other tools. However, adding automatic updates to the Call Hierarchy tool would require adding a locking state as in Blaze, because otherwise we would lose any way to navigate back to the original root once we have navigated to a subtree node.

If the information displayed in the tool can be refined manually, as in Blaze and the Call Hierarchy, we observed mode errors happening. For example, in Blaze problems occurred if the path length required scrolling the part of the path up- or downstream from the focus method, which could cause methods to be hidden between the focus method and the method directly above or below it on the screen. This could be solved by allowing the full path to scroll instead of scrolling in two separate parts. Using the Call Hierarchy, many participants had issues with the caller and callee view modes, and some forgot that the method they read in the editor is not the one selected as root of the Call Hierarchy. The latter problem did not occur in Blaze even if it was locked, because overlays always maintain a graphical connection between the method currently inspected in the editor and the information displayed in Blaze. Overlays are another property specific to Blaze and Stacksplorer, which simplifies parsing the additional information displayed while trying to understand source code in the central editor.

We already identified several problems with the design of Blaze: Mode errors might happen, and navigation targets that were more than one edge away from the currently edited method were rarely used. Nevertheless task completion times were on par with those in the Stacksplorer condition. Hence, the information displayed in Blaze seems relevant even though not all methods displayed are also navigated to directly using Blaze.

Effects of Task and Setup

The task had a significant effect on the prediction accuracy of all models except for the Forward Call Depth model. This indicates that the navigation behavior overall is influenced considerably by the task at hand. Further, both tasks used in our study were concerned with the same code base, which also might have an influence on the navigation behavior. This makes comparisons with other studies using different tasks and environments difficult.

Another problem when trying to study tools in development environments seems to be the large diversity in developers' strategies [20]. While in our experiment the groups of participants were comparable in terms of programming experience (in years) and coding done per week, we acknowledge that these measures cannot capture individual differences in coding strategies.

Of all methods we saw being visited during the sessions, 45% were visited by only one participant; only six methods were visited by more than half of the participants. These six methods were the ones essential for the task, they were either the solution (i.e., the method to be changed) or very closely related to it. All those 45% of methods were not important to the task at hand and hence likely visited during the initial exploration and search phase. This divergence in the first phase added noise to our model-based analysis.

SUMMARY AND FUTURE WORK

We presented a comparative study of three call graph navigation tools: the Call Hierarchy, which is ubiquitous in current IDEs, Stacksplorer, and Blaze. Call graph navigation tools support developers in performing software maintenance and yield higher task success rates compared to an IDE without any of these tools. Stacksplorer and Blaze could also decrease task completion times and thus make developers potentially more productive.

An analysis of navigation patterns in the different conditions indicated how the tools change developers' strategies. Without call graph navigation tools developers resort to workarounds for call graph navigation, such as text searches. Stacksplorer and Blaze changed developers behavior to include more call graph navigation than we observed in the Call Hierarchy condition. This is one potential explanation for the increased efficiency of these tools.

Call graph navigation mostly happens between neighboring methods, which benefits Stacksplorer. Only rarely developers navigate along multiple edges at a time, even though the tool at hand might support it.

For the comparison of the call graph exploration tools, we presented a new method to formally describe navigation behavior. This method quantifies the degree to which any given navigation history complies with a number of characteristic navigation models, yielding a representation that can be analyzed statistically. We found the results to be very helpful to find and analyze differences in navigation behavior among the conditions tested.

In future work it would be interesting to determine the influence of various other factors on navigation behavior. Promising factors to look at would be the programming language or API used, the task at hand, or the design patterns used in the application. It would also be interesting to examine how other performance measures, e.g., learnability or user satisfaction,

are influenced by the different tools. Further, while we chose to compare three tools that could each be embedded into the same IDE to maintain a high internal validity of the study, it is an important task for future work to analyze other navigation tools as well.

REPLICHI

The navigation sequences obtained through video annotation that were used for this study are available for further analysis as XML files generated with ChronoViz [6]. We also published a Mac OS X tool that uses the ChronoViz files and a call graph stored in an XML format to analyze the navigation behavior using the methodology we presented. The material can be downloaded at http://hci.rwth-aachen.de/developerNavigation. We would like to invite others to pick up our format to annotate navigation and our methodology to quantify navigation in their own studies. This would allow a comparison of navigation behavior among a wide variety of environments, tasks, and developers.

ACKNOWLEDGMENTS

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. Baeza-Yates, R. A., and Ribeiro-Neto, B. *Modern Information Retrieval*. Addison-Wesley Longman, 1999.
2. Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., and LaViola, J. J. Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance. In *Proc. CHI '10*, ACM (2010).
3. Coblenz, M. J., Ko, A. J., and Myers, B. A. JASPER : An Eclipse Plug-In to Facilitate Software Maintenance Tasks. In *Proc. 2006 OOPSLA Workshop on Eclipse Technology eXchange*, ACM Press (2006).
4. DeLine, R., Czerwinski, M., and Robertson, G. Easing Program Comprehension by Sharing Navigation Data. In *Proc. VLHCC '05*, IEEE (2005).
5. DeLine, R., and Rowan, K. Code canvas: zooming towards better development environments. In *Proc. ICSE '10*, ACM (2010).
6. Fouse, A., Weibel, N., Hutchins, E., and Hollan, J. D. ChronoViz: a system for supporting navigation of time-coded data. In *Proc. CHI '11*, ACM (2011).
7. Karrer, T., Krämer, J.-P., Diehl, J., Hartmann, B., and Borchers, J. Stacksplorer: Call graph navigation helps increasing code maintenance efficiency. In *Proc. UIST '11*, ACM (2011).
8. Kersten, M., and Murphy, G. C. Mylar: A Degree-of-Interest Model for IDEs. In *Proc. AOSD*, ACM (2005).
9. Ko, A., Myers, B., Coblenz, M., and Aung, H. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering 32*, 12 (2006).
10. Ko, A. J., and Myers, B. A. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *Proc. ICSE '08*, IEEE (2008).
11. Krämer, J.-P., Kurz, J., Karrer, T., and Borchers, J. Blaze: supporting two-phased call graph navigation in source code. In *Proc. CHI EA '12*, ACM (2012).
12. LaToza, T. D., and Myers, B. A. Developers Ask Reachability Questions. In *Proc. ICSE '10*, ACM (2010).
13. LaToza, T. D., and Myers, B. A. Visualizing Call Graphs. In *Proc. VLHCC '11* (2011).
14. LaToza, T. D., Venolia, G., and DeLine, R. Maintaining Mental Models: A Study of Developer Work Habits. In *Proc. ICSE '06*, ACM (2006).
15. Lawrance, J., Bellamy, R., Burnett, M., and Rector, K. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proc. CHI '08*, ACM (2008).
16. Murphy, G. C., Kersten, M., and Findlater, L. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software 23*, 4 (2006).
17. Piorkowski, D., Fleming, S. D., Scaffidi, C., John, L., Bogart, C., John, B. E., Burnett, M., and Bellamy, R. Modeling programmer navigation: A head-to-head empirical evaluation of predictive models. In *Proc. VL/HCC '11* (2011).
18. Pirolli, P., and Card, S. K. Information Foraging. *Psychological Review 106*, 4 (1999).
19. Pressman, R. S. *Software Engineering: A Practitioner's Approach*, 7th ed. McGraw-Hill, 2010.
20. Robillard, M. P., Coelho, W., and Murphy, G. C. How Effective Developers Investigate Source Code:An Exploratory Study. *IEEE Transactions on Software Engineering 30*, 12 (2004).
21. Sillito, J., Murphy, G. C., and Volder, K. D. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering 34*, 4 (2008).
22. Singer, J., Elves, R., and Storey, M.-A. NavTracks: Supporting Navigation in Software Maintenance. In *Proc. ICSM '05*, IEEE (2005).
23. Singer, J., Lethbridge, T., Vinson, N., and Anquetil, N. An Examination of Software Engineering Work Practices. In *Proc. 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, IBM Press (1997).
24. Čubranić, D., and Murphy, G. C. Hipikat: Recommending Pertinent Software Development Artifacts. In *Proc. ICSE '03*, IEEE (2003).
25. Winograd, T. Breaking the complexity barrier again. *ACM SIGIR Forum* (1974).