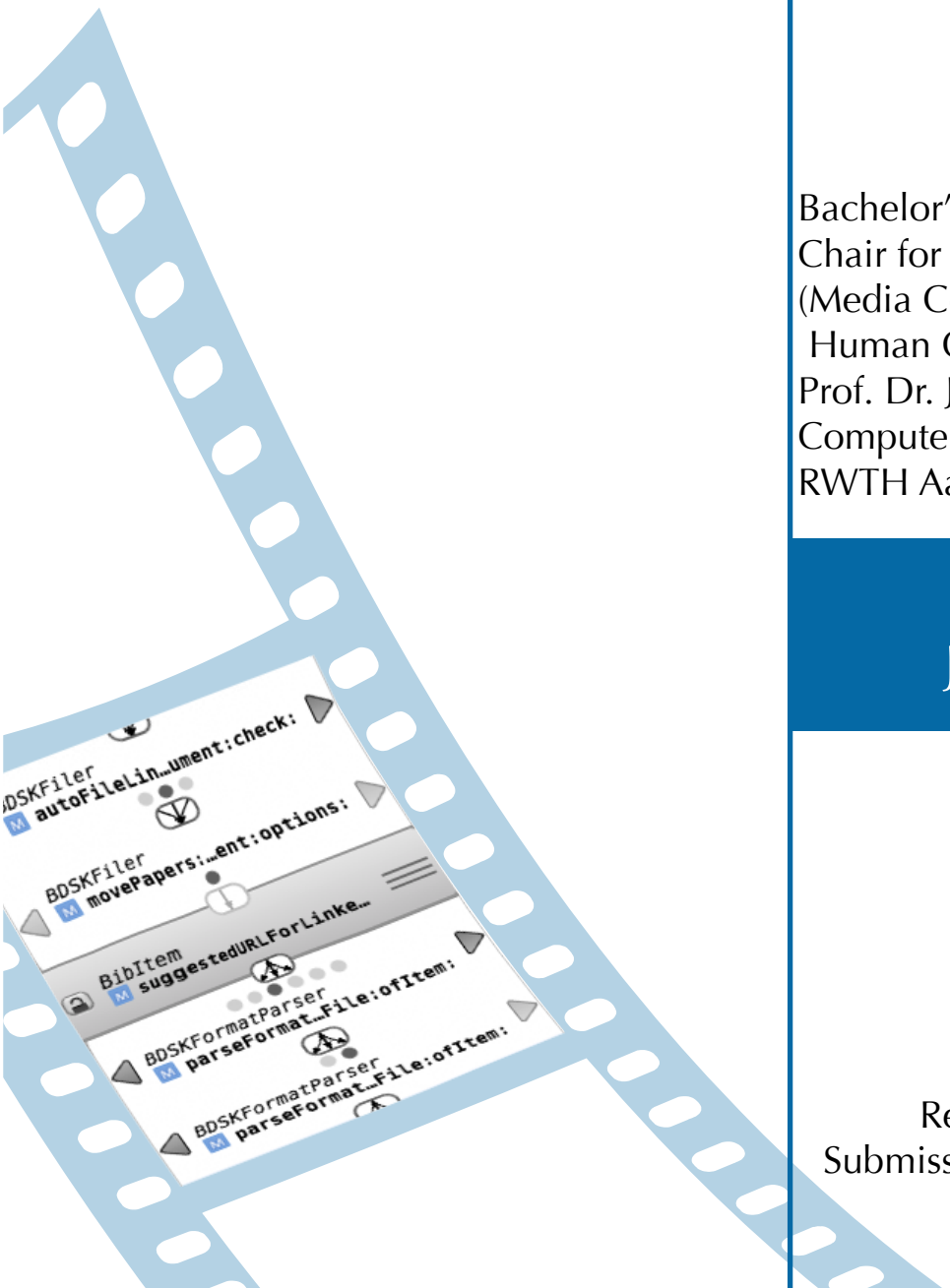
RWTH AACHEN UNIVERSITY

Blaze

Navigating Source Code via Call Stack Contexts



Bachelor's Thesis at the
Chair for Computer Science 10
(Media Computing and
Human Computer Interaction)
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Joachim Kurz

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Bernhard Rumpe

Registration date: June 27th, 2011
Submission date: September 30th, 2011

Parts of the results of this thesis have been used to write a paper submitted to the ACM SIGCHI Conference on Human Factors in Computing Systems 2012. However all the work presented in this thesis is the result of my work and my work alone unless otherwise specified.

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, September 2011
Joachim Kurz

Contents

List of Figures

List of Tables

Abstract

Understanding unknown source code and navigating in it is a common problem for developers. Navigating along the call graph is a particularly important type of such navigation.

We present Blaze, a call graph exploration tool designed to help developers search along paths. Blaze implements established programmer navigation models by first providing additional information scent to developers browsing the source code. This is done by always displaying a path that contains the currently viewed method — the focus method — next to the editor that is used to browse the source code. Then, when the developer found an anchor point to start the exploration from, Blaze can lock this focus method. This prevents it from auto-updating; thus providing a save starting point to backtrack to. Blaze can then be used to explore all paths that contain this anchor point, while always providing a save return node and making it very easy to navigate along the selected path.

We first compared Blaze to a Call Hierarchy view. This kind of call graph exploration tool is used in many modern Integrated Development Environments (IDE), for example, Eclipse or NetBeans. We found that Blaze significantly decreases task completion time in all the tasks we tested. We then used results from a similar earlier study comparing Stacksplorer, another research prototype for call graph exploration, to an IDE without such a tool. These results allowed us to compare all four conditions. We found that call graph exploration tools significantly increase the success rate compared to an IDE without such a tool. We also found that the two research prototypes significantly decrease the task completion time compared to an IDE *with or without* the Call Hierarchy tool.

Überblick

Unbekannten Quellcode zu verstehen und darin zu navigieren ist ein häufiges Problem für Entwickler. Die Navigation entlang des Call-Graphen ist eine besonders wichtige Art der Navigation in solchen Fällen.

In dieser Arbeit stellen wir Blaze vor, ein Call-Graph-Explorationswerkzeug, das entwickelt wurde, um Entwicklern die Suche entlang von Methodenaufrufpfaden zu erleichtern. Blaze setzt etablierte Modelle der Programmierernavigation um, indem es Entwickler zuerst mit zusätzlicher "Informationswitterung" versorgt. Dies wird erreicht, indem immer direkt neben dem Quellcode-Editor ein Pfad angezeigt wird, der durch die aktuell betrachtete Methode — die Fokusmethode — verläuft. Dann, wenn der Entwickler einen günstigen Startpunkt, einen "Ankerpunkt", gefunden hat, kann Blaze die Fokusmethod arretieren. Auf diese Weise wird das automatische Aktualisieren der Fokusmethode verhindert und der Entwickler hat einen sicheren Startpunkt, zu dem er zurückkehren kann. Nun kann Blaze benutzt werden, um alle Pfade durch diesen Ankerpunkt zu erforschen. Blaze macht es dabei sehr einfach zu diesem Ankerpunkt oder auch jeder anderen Methode auf dem Pfad zu springen.

Zuerst haben wir Blaze mit einer Call Hierarchy Anzeige verglichen. Diese Art von Call-Graph-Explorierungswerkzeug wird in vielen modernen Entwicklungsumgebungen benutzt, zum Beispiel Eclipse und NetBeans. Wir konnten zeigen, dass Blaze die Zeit, die für eine Wartungsaufgabe benötigt wird, in allen von uns getesteten Fällen signifikant verringert. Als nächstes haben wir die Ergebnisse einer älteren Studie benutzt, die Stacksplorer, einen weiteren Forschungsprototypen zur Call-Graph-Exploration, mit einer Entwicklungsumgebung ohne ein solches Tool verglichen hat. Mit diesen Ergebnissen konnten wir alle vier Konditionen vergleichen und zeigen, dass solche Call-Graph-Explorierungswerkzeuge die Erfolgsrate gegenüber einer Entwicklungsumgebung ohne solche Werkzeuge signifikant erhöhen. Wir konnten außerdem zeigen, dass die beiden Forschungsprototypen die Zeit, die benötigt wird um eine Aufgabe abzuschließen, im Vergleich zu einer Entwicklungsumgebung *mit oder ohne* ein Call Hierarchy-Werkzeug signifikant verringern.

Acknowledgements

First of all I would like to thank my advisors, Jan-Peter Krämer and Thorsten Karrer, for their support and willingness to answer all kinds of questions. I also want to thank Chatchavan Wacharamanotham for his statistics support. He answered lots of questions about statistics I had and spent probably hours doing so.

Special thanks to Matt Gemmel[a] for the MAAttachedWindow code I used to create overlays. Special thanks also to Thorsten Karrer and Jan-Peter Krämer again for programming a view that resembles the iOS page indicator[b] after I realized that there is no standard UI element that does this on Mac OS X. And special thanks to Fabian Kürten, for helping me tame LaTeX's layout engine.

Also I want to thank all of the participants of my user studies again for taking the time and providing very valuable input.

Last, but not least, I want to thank Leandra for putting up with me having almost no time for anything except this bachelor thesis, especially during the last month.

[a] http://mattgemmell.com/ [b] http://hci.rwth-aachen.de/pageindicator

Conventions

Throughout this thesis we use the following conventions.

To be consistent with the previous Stacksplorer related thesis, we will use similar conventions:

> "[E]stimates about the size of an application will be given in *source lines* of code (SLOC). Measurements are always, except if they are cited, performed using sloccount[a] by David A. Wheeler, which counts each 'line ending in a newline or end-of-file marker, and which contains at least one non-whitespace non-comment character'[b]."[Krämer, 2011]

Diagram Conventions In box plot diagrams the whiskers always extend to the furthest data point within 1.5 times of the interquartile range. The top of the box represents the 75% quartile, the bottom the 25% quartile and the median is shown by a horizontal thicker line inside the box.

Statistic Conventions If nothing else is specified the significance level used is 0.05 and confidence intervals are calculated as 95% confidence intervals.

Text Conventions Source code and implementation symbols are written in `typewriter-style` text except in listings or appendixes.

Definitions of technical terms or short excursus are set off in colored boxes.

[a] http://www.dwheeler.com/sloccount/
[b] http://www.dwheeler.com/sloccount/sloccount.html

Definition:
Excursus

EXCURSUS:
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

The whole thesis is written in American English.

Citations We use the citation styles laid out in Hämäläinen [2006]. Thus, when we take information from a reference and use it in only one sentence in our thesis, we will add the reference before the full stop of the sentence, like so [Hämäläinen, 2006]. However, if we use several pieces of information from the same source in successive sentences we save the reference at the end of each sentence. Instead we add the reference at the end of the paragraph. To indicate that it belongs to the complete paragraph we add the reference *after* the full stop, like so. [Hämäläinen, 2006]

1 Introduction

"We think basically you watch television to turn your brain off, and you work on your computer when you want to turn your brain on."

—Steve Jobs (2004)

1.1 Motivation

Software development does not stop when a product is deployed, instead it is usually necessary to adapt the software product to changing requirements [Sommerville, 2007, p. 489]. This is called maintenance, and costs to do so often make up the majority of the total costs during the software lifecycle [Sommerville, 2007, p. 489]. According to Pressman [2010] maintenance even makes up 70% of the total expenses of a software project. Thus, it is important to support programmers during maintenance tasks.

Maintenance tasks are common.

Since such maintenance tasks usually involve changes in the program code, it is important to understand the program to estimate the effects a change will have and to implement the change in the most effective way. Often, it is not necessary to understand the complete program but just a part of it [Erdös and Sneed, 1998]. According to Erdös and Sneed [1998] one of the basic questions to answer when supporting program understanding is "where is a particular subroutine or procedure invoked?".

Understanding the complete program is not required for a maintenance task.

Also, programmers spend a lot of time navigating in source code, especially during maintenance tasks [Robillard et al., 2004]. According to Ko et al. navigation makes up 35% of the time spent on maintenance [Ko et al., 2006, 2005]. A particularly important type of navigation is navigation along the call graph [Karrer et al., 2011].

Programmers spent a lot of time navigating in source code, often along the call graph.

Definition: *Call Graph*

CALL GRAPH:
The call graph for a piece of software is the graph that is constructed by interpreting each method and function as a node and adding a directed edge between method a and method b iff a calls b.

Especially forwards and backwards navigation is important.

Sherwood [2008] found in their study that 18% of all navigation actions was "forward" navigation, which means that subjects navigated to the declaration of a discovered object. An additional 8% was "backwards" navigation looking for places from which an element is referenced. However, they do not reveal how many of those navigation actions were along the call graph.

Searching along paths is important.

LaToza and Myers [2010b] found that developers ask reachability questions. According to their definition, a "reachability question is a search across all feasible paths through a program for statements matching search criteria" [LaToza and Myers, 2010b]. Making navigation along the call graph easier may help in answering these kinds of questions.

Stacksplorer gives only little support for searching across paths.

Stacksplorer [Krämer, 2011] already answers the question identified by Erdös and Sneed [1998] and supports navigation along the call graph.

However, Stacksplorer only displays the direct successors and predecessors in the call graph, but developers also have questions that they try to answer by searching across whole paths (For more information see section 3.2.2—"Stacksplorer"). Also, programmers do not only search upstream looking for callers of a method (as implied by Erdös). They also search downstream (following method calls), for example, to better understand why a method is called by identifying methods deeper in the call graph that cause (expected) side effects [LaToza and Myers, 2010a]. Stacksplorer does support up- and downstream navigation but only one level at a time.

We developed Blaze to better support navigation and exploration along paths.

To support developers in searching across paths we modified Stacksplorer and developed Blaze to support the visualization of possible call stacks. Blaze is an Xcode plugin as well, but provides a narrow look into the depth of the call graph. This is an orthogonal approach to Stacksplorer's direct-neighbor-focused breadth-first view of the call stack. Figure 1.1 shows

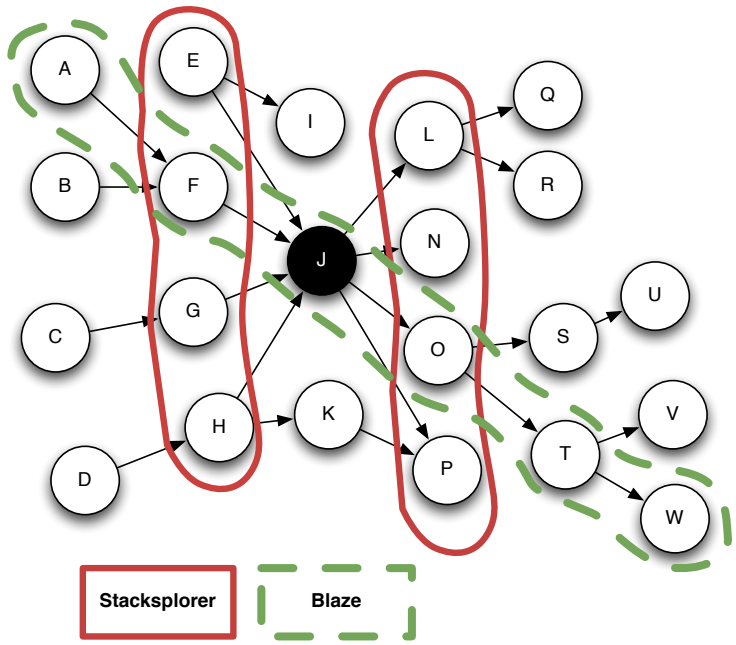


Figure 1.1: *The different parts of the call graph surrounding a focus method (black) visualized by Stacksplorer and Blaze. Circles represent methods, arrows represent calls between methods. Stacksplorer visualizes the complete direct neighborhood of a method, Blaze a path that contains the focus method. The Blaze visualization can be changed to show any of the displayed paths through the focus method, thus it can show every method except for K and I without changing the focus method.*

which parts of a sample call graph Stacksplorer and Blaze would show.

FOCUS METHOD:
The *focus method* is the "method [the user] is currently working on or trying to understand" [Karrer et al., 2011].

Definition:
Focus Method

1.2 Chapter Overview

Chapter 2 In this chapter, we will first give some background information about programmer navigation. We will describe some research results about this kind of navigation.

Chapter 3 In chapter three, we present related work to Blaze.

We will first describe some basic tree visualizations, then describe the current state of call graph exploration tools in IDEs and show some research prototypes that are similar to what we want to achieve with Blaze.

Chapter 4 Our design process for Blaze and the different stages during its design are described in this chapter, including a set of small user interviews we did to verify that our visualization is understood by users.

Chapter 5 Here we describe the final version of the software prototype for Blaze. We also describe our version of the Call Hierarchy plugin that we compared to Blaze.

Chapter 6 We compared Blaze to an implementation of the Eclipse Call Hierarchy. We also used results from an older study to compare these two plugins to Stacksplorer and Xcode without a plugin. The results of these comparisons are described in this chapter.

Chapter 7 In this last chapter we summarize our work and contributions and point out some interesting further research opportunities.

2 Background

"To me programming is more than an important practical art. It is also a gigantic undertaking in the foundations of knowledge."

—Grace Hopper

There have been several studies looking at how programmers navigate both several years or even decades ago and in recent times. Several of those have formulated models for programmer navigation. We will now discuss some of the more recent ones. Older studies have the risk of lacking applicability because they often investigated quite different programming environments. A comparatively small Assembly project is quite a different thing than a Java project with several hundred thousand lines of code.

2.1 Information Foraging Theory

Information foraging is a theory that describes how humans search for information. It uses the concept of *information scent* derived from animal scent arguing that humans searching for information act similar to predators hunting prey [Lawrance et al., 2008]. It has previously been used to predict web user navigation. We will present two similar models that adapt this theory to programmer navigation.

The information foraging theory uses *information* scent to predict navigation.

2.1.1 Programmers Look for an Anchor Point to Start Exploration

Ko et al. [2006] developed a model for programmer navigation in source code and program understanding by observing 31 Java developers. According to this model (see figure

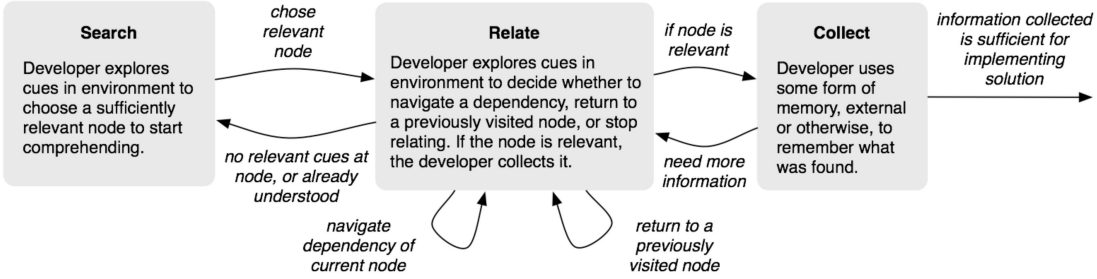


Figure 2.1: *A model of programmer navigation and program understanding developed by Ko et al. [2006]. Diagram reproduced from the same source.*

Programmers look for an anchor point.

2.1), programmers start out by searching for a point in the code or other artifacts to start from. When they have found a convincing starting node, they explore other nodes along relationships (call relationships, inheritance, documentation references, etc.) from this node. They explore these nodes recursively, again exploring relationships and collecting nodes they find relevant. If they do not find other interesting nodes, they backtrack to previously visited nodes until they reach the starting node again. If there are no other nodes to explore from the starting node, they go back to the search phase and look for a new starting point. They stop when they have gathered enough information to accomplish their goal. [Ko et al., 2006]

Programmers often backtrack to previously visited nodes.

Showing relationships for the programmer's current focus should be helpful.

Ko et al. also derived some implications for tool design from their theory. For example, tools should support programmers to search more effectively. One way to do this would be to provide more 'layers' of nodes that can be explored without navigating. This could be accomplished by to not only showing relations to other nodes but also showing their related nodes. Another recommendation is to support programmers in relating information to the currently viewed node more easily, for example, by automatically showing some related nodes based on the current selection (i.e., the programmers focus). [Ko et al., 2006]

2.1.2 Information Scent

Lawrance et al. [2008] use a similar approach to Ko et al. and adapt the information foraging theory to programmer navigation in source code, too. However, they do it a bit differently.

They compare programmers to 'predators' 'hunting' for 'prey',

whereby the prey can be any kind of information programmers look for, for example, how to fix a bug or where to make a specific change. To find prey, developers evaluate cues in the source code, which can be almost anything from simple words in function names or documentation to runtime behavior of the application. These cues are then judged by how likely they will lead to the 'prey'. The likelihood to lead to the prey is the 'information scent': How much the cue smells like the information the developer is looking for. The 'topology' the predator wanders would then be the collection of artifacts in the software (source code, documentation etc.) and the relationships and paths between them. [Lawrance et al., 2008, 2010]

Programmers 'hunt' for information and judge relationships by their 'information scent'.

They tested this theory by calculating the information scent of a cue by calculating the similarity of it (for example a method name) to the bug report the developers were working on and used this to predict the navigation of programmers. This prediction was quite accurate, closely following the aggregated navigation of a group of developers that were tested. It even predicted the navigation of a single developer better than using the navigation of a single other developer as prediction. [Lawrance et al., 2008]

Information scent can predict programmer navigation very well.

2.2 Other

Developers use tabs as breadcrumbs for their navigation paths. Sherwood [2008] tried to improve programmer navigation by changing the Eclipse "one file per tab" model to a "one history per tab" model by saving a history of viewed files on a tab by tab basis, similar to what is currently done in web browsers. They expected developers to take a more breadth-first approach (which was also encouraged during their study) by opening one tab for each possible further direction of exploration to remember to evaluate each possibility. Instead, what they found is that developers only opened one new tab in such branching situations to "save" the position where they took one of several possible exploration paths, thus using the tabs like breadcrumbs to be able to go back to an earlier state in the exploration.

Instead of using tabs for a more breadth-first navigation developers used them as 'breadcrumbs'.

Requirements for Software Exploration Tools. There are a number of papers proposing different sets of requirements for software exploration tools, for example, Schäfer et al. [2006] and Storey et al. [1997]. Often, these are targeted at more inclusive software exploration tools that are meant to stand and be used alone, thus not all of them apply to our approach. But some of the requirements presented in [Storey et al., 1997] are useful; therefore we will describe them here.

Relationships between low-level objects should be shown.

One strategy to understand source code is to start at the smallest units and then create bigger, more abstract, units from them until the whole system is understood. This is called a bottom-up strategy. For this to work, it is important that the tools used show the different relationships between the low-level objects of the software project and thus enable abstraction to higher level objects by grouping the low-level objects together (E1[1]). [Storey et al., 1997]

Delocalization effects should be mitigated.

Another important part is to reduce delocalization effects (E2). Delocalization describes the fact that, in modern software projects, relevant pieces of an implementation of a feature are often split up in many different parts, which are located at different places in the source code. Software exploration tools should try to mitigate the problems that occur from this delocalization. [Storey et al., 1997]

Enabling sequential navigation and providing cues to help the programmer orient themselves is important.

Another requirement that we will try to fulfill is the enabling of sequential navigation, meaning that it should be possible to navigate along the data-flow or control-flow of the software project (E8). But not only should navigation be possible, tools should also provide hints to the developer to help him stay oriented (E11). For example, tools should show the current focus to the developer, show how they got where they are now, and where they could go now. In addition, they should also try to prevent confusion and disorientation by using animations to explain interface changes or providing easy to understand presentations/layouts (E15).

[1] We used the same identifiers for the requirements as Storey et al. [1997] to make the comparison easier.

3 Related work

"All problems in computer science can be solved by another level of indirection.
... But that usually will create another problem."

—David Wheeler

There are many different ways to visualize a program's structure in general. Since we do not plan to provide an overview over the complete code but only over the surroundings of a specific part—in our case the methods in the call graph—we will not focus on complete program visualization approaches but instead look for work which tries to visualize only a part of the program structure. Edge cases in that way are projects like Code Bubbles [Bragdon et al., 2010], which allows the developer to choose which parts of the code they want to focus on and displays the relations between the chosen code segments. We did not want to create a whole new Integrated Development Environment (IDE). Instead we integrated the visualization with Xcode[a], an existing IDE, more like an accessory view in the same way Stacksplorer works.

We will focus our research on partial visualization of program structure, not complete visualization.

Another edge case are projects that use degree of interest functions to select information to display from the context for a currently viewed code segment. An example would be [Jakobsen and Hornbæk, 2009]. These tools are not limited to call relations. We will have a look at those projects to see whether they have useful visualization techniques but will not use their degree of interest functions and instead focus on visualizing related methods by using the call graph.

We will not talk about degree of interest tools and instead concentrate on visualization and more interactive tools.

[a] http://developer.apple.com/Xcode/

3.1 Graph and Tree Visualizations in General

We concentrate on DAG and tree visualization.

A call graph is usually a proper[1] directed graph not just a tree, not even acyclic because of recursion. But by limiting the visualization to one method plus its descendants and ancestors, and replacing recursion (i.e., cycles) by special nodes, the part of the call graph to visualize can be transformed into a directed acyclic graph (DAG) or even a tree. Thus, we will concentrate on DAG and tree visualization.

3.1.1 Basic Tree Visualization

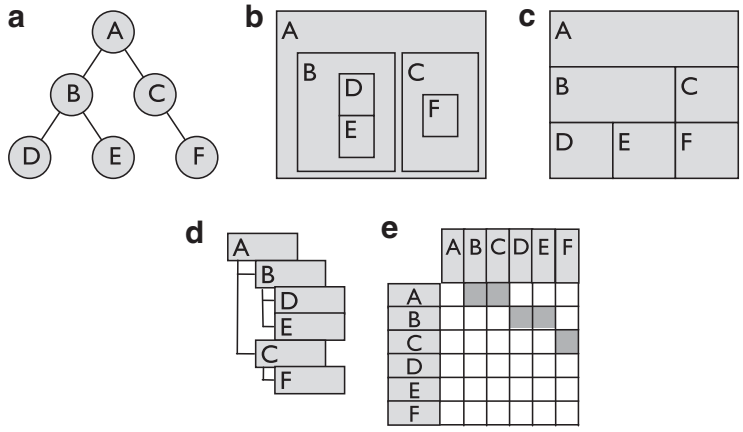


Figure 3.1: *Shown are five different tree representation styles, a) node-link diagram, b) nested diagram, c) adjacency diagram, d) indented list, e) matrix representation. Redrawn from [Graham and Kennedy, 2010].*

Graham and Kennedy [2010] identify 5 types of basic tree visualizations (see Figure 3.1):

a) node-link diagram
b) nested diagram
c) adjacency diagram
d) indented list
e) matrix representation

[1] contains cycles and possibly more than one predecessor per node

The matrix representation is included in their list but not considered in their evaluation of tree visualizations because they find it too complicated and space-inefficient for trees. There are studies showing that the indented list is subjectively preferred by users and other studies implying that this is only due to the familiarity of the users caused by the lists usage in Windows [Graham and Kennedy, 2010]. Andrews and Kasanicka [2007] compare four hierarchy browsers[2], one of them an indented list (which they call tree view), two nested diagrams, and one node-link diagram. They could not find a significant difference in completion time when letting the users explore different hierarchies except for one pair of browsers for one out of eight different tasks. Since there is no clearly favorable visualization, we think there is room for improvement and hope a more focused visualization will perform better for our use case.

There seems to be no big differences between different basic tree visualizations.

3.1.2 Fisheye Views

One graph visualization method respecting the idea of having a focus point in a graph and displaying context of different importance is the concept of fisheye views described by [Furnas, 1986]. Often, they are just applied as an optical effect to existing graph drawings by zooming in on a focus point and distorting nodes that are far away from the focus point (see Figure 3.2). But fisheye views are not limited to optical effects, there are also semantic ones. Jakobsen and Hornbæk [2009] modified the Eclipse IDE to display relevant lines from the currently viewed code file below and above the editor view and could show that their modified editor was adopted and actually used in real-life work.

Fisheye views highlight a focus point and distort distant features.

3.2 Call Graph and Control Flow Graph visualization

There are also programs and plugins focused on visualizing call graphs, not just general graphs. We will review a few of those that are related to Blaze or even inspired its design.

[2] A hierarchy is a tree, thus a hierarchy browser is just another kind of tree

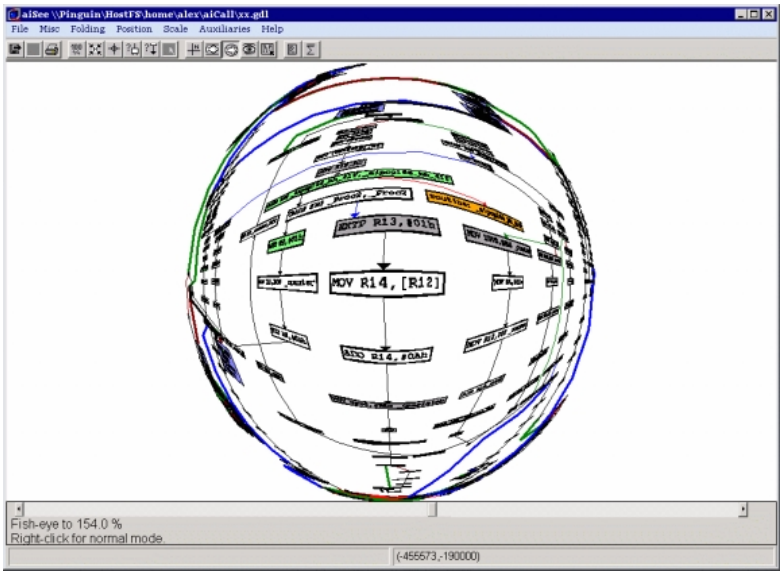


Figure 3.2: *A fisheye view of a control-flow graph. Redrawn from [Evstiougov-Babaev, 2002].*

3.2.1 Current IDEs

Most big IDEs use tree views to show the Call Hierarchy. But they are often not implemented very well.

Several of the more popular IDEs already have call graph visualization features, but they all use different kinds of tree views (i.e., indented lists) to visualize them (see figures 3.3 and 3.4). Eclipse[a], NetBeans[b] and IntelliJ IDEA[c] also only allow the user to view just one side of the call graph (either callees or callers) at a time (for screenshots of NetBeans and IntelliJ see 1 and 2 in appendix 7.2.2—"Other Pictures and Diagrams").

The Visual Studio visualization can be quite confusing.

Microsoft Visual Studio[d] displays a node for callees and one for callers in the same view, but one has to expand this collection of callees/callers before being able to see them. And one has to expand a similar collection on each level down the call hierarchy. Even worse: One can interleave callee and caller relationships in this tree view; thus, at one point the parent-child relation in the tree view might mean "callee" and further down the tree it might mean "caller" (see Figure 3.4 for an example).

visualization and exploration tool.

[a]http://www.eclipse.org/downloads/ [b]http://netbeans.org/
[c]http://www.jetbrains.com/idea/
[d]http://www.microsoft.com/visualstudio/

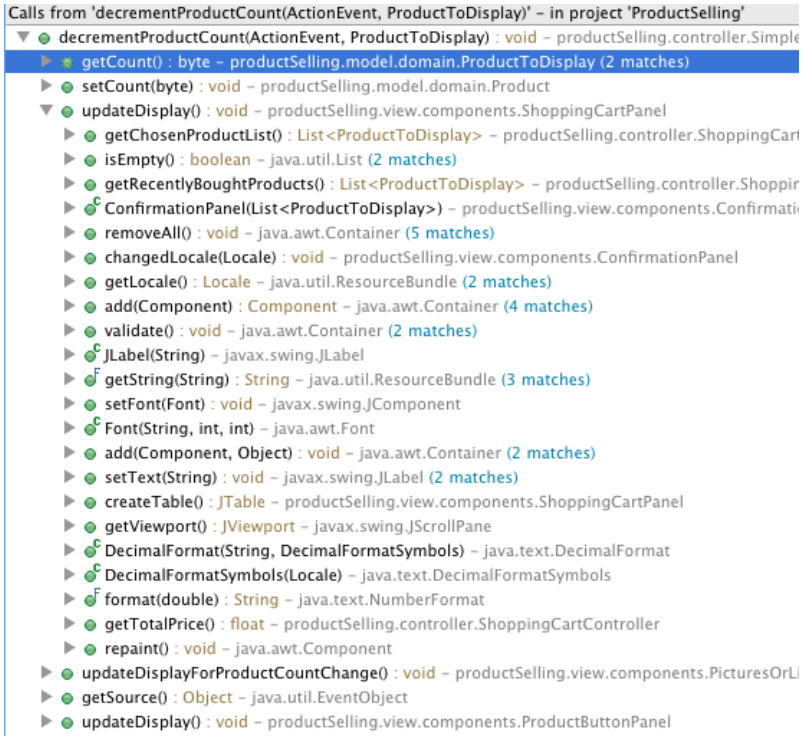


Figure 3.3: *Eclipse Call Hierarchy view: Only one direction (callees or callers) is viewable at a time.*

Also none of the mentioned tree views updates automatically when another method is selected. The user has to invoke it explicitly for each possibly interesting method. Thus, there seems to be room for improvement.

3.2.2 Research

Besides the call graph visualizations used in the "real world", there are also some research prototypes with similar goals.

Stacksplorer

Our prototype, Blaze, was developed as a follow-up to Stacksplorer, which was developed during a diploma thesis at our

[a] http://hci.rwth-aachen.de/stacksplorer

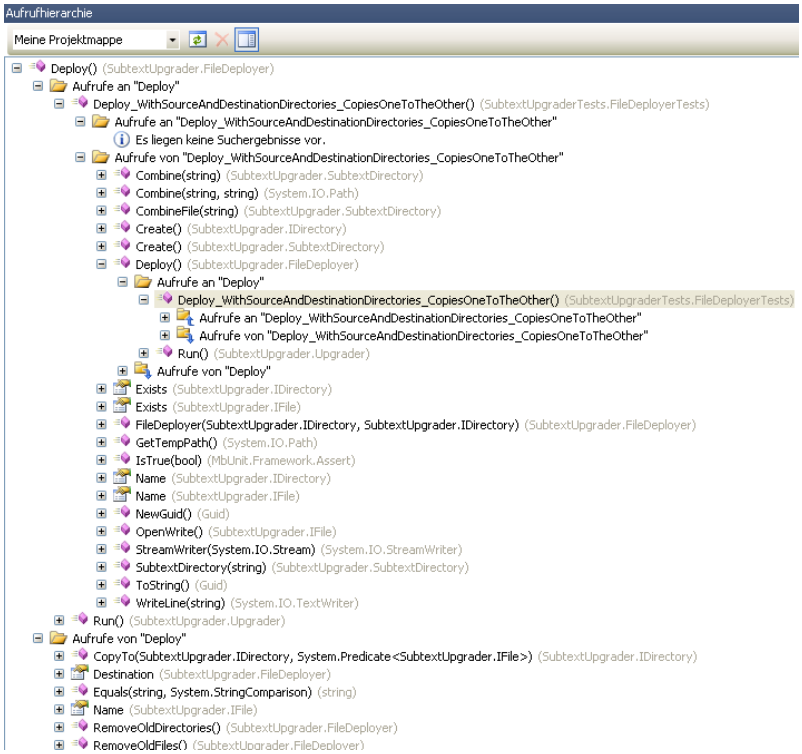


Figure 3.4: *Visual Studio Call Hierarchy view: Both directions can be seen at a time, but directions can be interleaved as well (Caller → Callee → Caller visualized as children).*

chair [Krämer, 2011]. Stacksplorer takes the call graph surrounding the currently selected method in the editor (the focus method) and displays incoming and outgoing calls for this method. Incoming calls (i.e., methods that call the focus method) are displayed to the left of the source code editor, outgoing calls (i.e., methods that are called by the focus method) are displayed to the right of the editor. This way one can follow the execution trace reading from left to right. [Karrer et al., 2011]

Stacksplorer shows direct callers and callees of a method.

Stacksplorer also allows users to navigate to a calling or called method by clicking on it in the left or right view. When a user clicks on a method on the right it slides to the left and will be opened in the editor. The previous focus method slides to the left and will be displayed in the left view as one of the incoming calls. This works the other way round for clicking on a method on the left. [Karrer et al., 2011]

Stacksplorer allows users to navigate along the call graph.

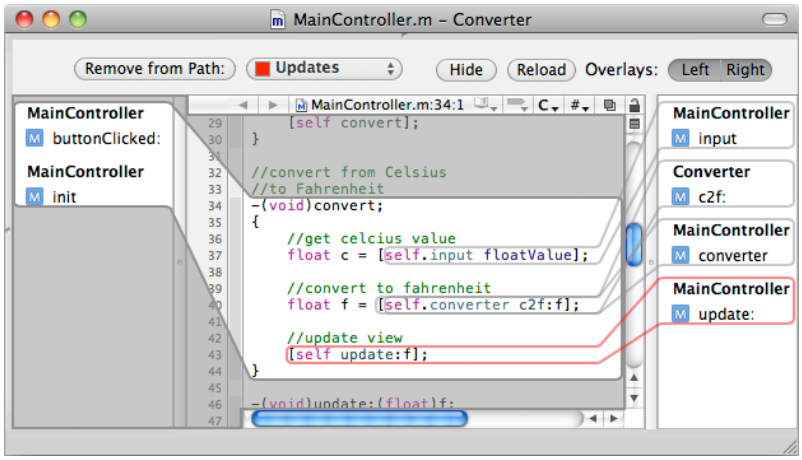


Figure 3.5: *Stacksplorer displays the callees of the current focus method on the left of the editor view and the called methods on the right. Overlays are used to show where the methods on the right are called in the code. Picture from the Stacksplorer Website*[a].

If one interprets Stacksplorer as a window placed on the call graph (hiding everything but the direct neighborhood of a method) clicking on a method represents sliding this window to the new focus method. This concept of a window on the graph and moving the window for exploration was described by Herman et al. [2000]; Huang et al. [1998] called it a "logical frame".

Stacksplorer presents a "logical frame" around a method.

To highlight where methods on the right are called, Stacksplorer optionally displays overlays connecting the method views on the right with their calls in the source code [Karrer et al., 2011]. Thus, Stacksplorer is tightly integrated with the editor showing the source code and even uses the editor as part of the visualization.

Stacksplorer is tightly integrated with the source code editor.

CallStax

Young and Munro [1997] also explored the visualizations of all possible paths through a focus method in a tool called CallStax. It displays all possible stacks in the source code of a piece of software as stacks of 3D blocks, with the blocks representing methods (see figure 3.6). Individual methods can be selected to align all stacks at that method and see all paths that

CallStax displays all possible stacks.

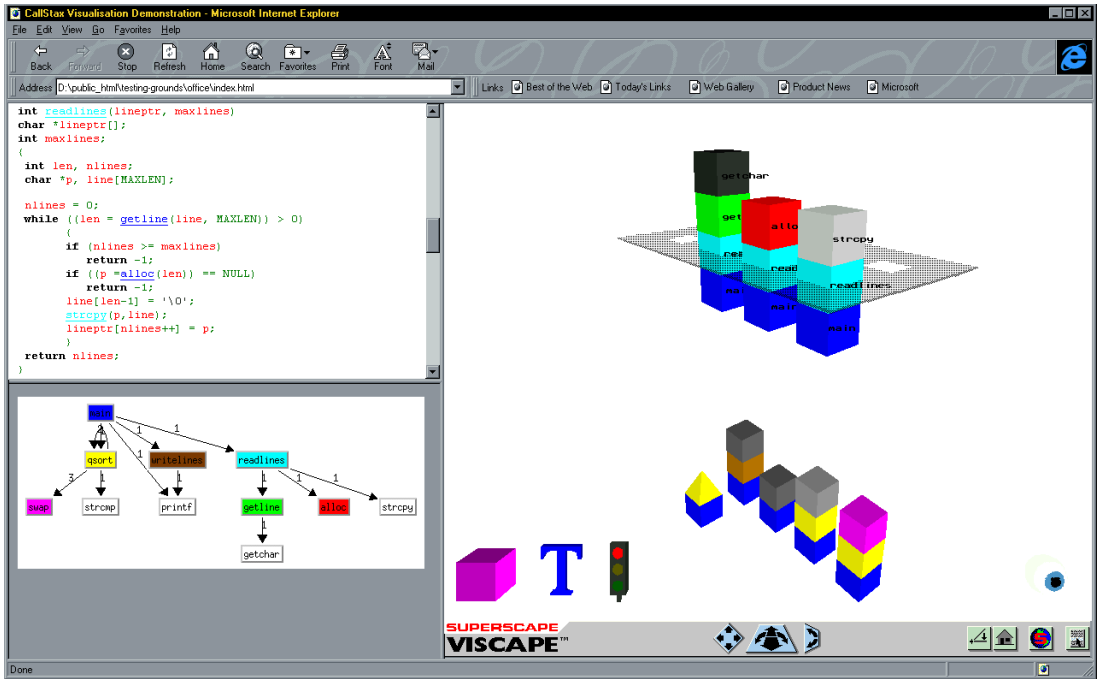


Figure 3.6: *CallStax provides a 3D view of all possible stacks of functions in a software system. Functions are visualized as small cubes stacked onto each other when a method calls another. Users can select a function which causes all stacks to be aligned at that method. Stacks that do not contain the function are moved away and leave empty spaces in the mesh that represent the selection. It can also be integrated with other views such that clicking on a function in CallStax highlights it in the code editor and vice versa. Image reproduced from [Young and Munro, 1997].*

contain this method. Stacks that do not contain this method are moved away to be visually distinctive, but their previous positions are shown as holes in the mesh that represents the selection. [Young and Munro, 1997]

On the one hand, showing all possible stacks removes the problem of crossing edges in a graph visualization, which often makes the representation visually complex. On the other hand, it introduces the problem of the user having to mentally combine all the different representations of one method to understand its role. To help the user identify the different representations of one and the same method, the methods are colored similarly. [Young and Munro, 1997]

It is also possible to zoom in and out of the representation to get a better overview or more details. When the zoom level reaches a certain threshold the individual method visualiza-

tion becomes more complex by displaying more information like metrics or inner structure of the function. CallStax can also be integrated with other visualizations like a simple code editor and allows clicking on a method in CallStax to show its implementation in the editor and vice versa. [Young and Munro, 1997]

Zooming in reveals details of the implementation of a method.

Although the concept of showing all paths is similar to our idea, there are still a lot of differences. CallStax shows all stacks of a complete software system or at least all stacks for one method at the same time, while we want to focus on only one method. It is also a 3D visualization while we think 2D should be enough and it still uses a lot of space.

REACHER

REACHER, a tool developed by LaToza and Myers [2011], also shows only a part of a call graph. But it is not limited to the direct neighborhood of one method. It displays several paths of several selected methods. It can also search through the call graph, evaluates conditions and loops and is overall a much mightier call graph exploration tool than Stacksplorer. However, it also has a much more complex visualization (see figure 3.7).

REACHER is a much mightier and much more complex call graph exploration tool.

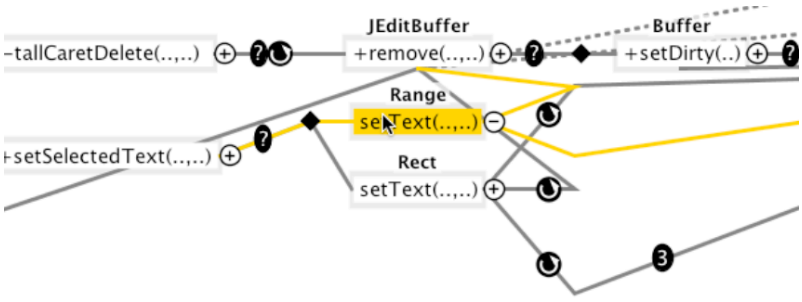


Figure 3.7: *An excerpt from a visualization of a search through the call graph using REACHER. Image reproduced from [LaToza and Myers, 2011].*

REACHER is used to answer reachability questions. A user starts using it by selecting a start method. They can then choose to search upstream, which finds all calls that might have hap-

pened before this method[3] or downstream, which finds all methods that are indirectly called by the start method. [LaToza and Myers, 2011]

REACHER allows users to search for methods in the call graph and build the visualization incrementally.

This list of methods can be filtered by entering a string into a search field. Users can add methods from this list permanently to the current call graph visualization by double-clicking them. Calls between method nodes are represented by lines, although by default any methods that have not been added explicitly are hidden. Paths from one method to another that contain hidden methods are represented by a dashed line. Hidden outgoing paths are represented by a circled plus, which can be used to reveal those other paths. [LaToza and Myers, 2011]

Lots of different icons convey different bits of information.

Icons on the call lines are used to give hints about the context of the call. A question mark represents a conditional guarding the call, a circled arrows represents a loop containing the call, a number n means the method is called n times from the implementation of the calling method, a diamond is used to split up a path that could take different directions due to overridden methods.

REACHER can also be used for navigation.

Users can click on any of the call lines between method nodes to navigate to the location of this call in the code. Thus, REACHER already implements the idea of searching and navigating along call paths but is a lot more complex than we envisioned our tool to be.

Other Program Exploration Tools

More inclusive software exploration tools are often too complex to be comparable.

There are a lot of other program exploration tools but many of those strive for an even wider applicability than REACHER and incorporate even more features. This limits their applicability to our problem and the comparability to the tool we developed, since we wanted to design a tool that is focused on call graph navigation, uses only as little space as possible and is still concise.

[3] Either because the found method is (indirectly) calling the start method or because it is (indirectly) called by a method which (indirectly) calls the start method later.

4 Design

"Good design is also an act of communication between the designer and the user, except that all the communication has to come about by the appearance of the device itself. The device must explain itself."

—Donald Norman

Starting from the idea of trying the orthogonal approach to Stacksplorer of visualizing paths, we designed a new call graph visualization. We will now describe the basic idea behind it, the changes we made later in the design process, and the final design.

4.1 Basic Idea

Stacksplorer only shows the direct predecessors and the direct successors of the focus method, but it shows all of those successors and predecessors at the same time. We wanted to explore the orthogonal approach of visualizing only one path at a time, thus visualizing several depth levels of the call graph from the current focus method. As a tradeoff, we would only show one of the possible paths compared to Stacksplorer, which shows all possible incoming and outgoing methods. A graphical description of the different parts of the call graph that are shown by Stacksplorer and Blaze was shown in figure 1.1.

We wanted to only visualize one path through the focus method at a time.

To visualize only one path at a time but make it possible to switch between different paths, we decided to use a combination lock metaphor. The different methods on a path would be stacked on top of each other and one could exchange each method for another method by clicking on an arrow button next to it (see figure 4.1). This would move the clicked method out and another alternative in.

We wanted to use a combination lock metaphor.

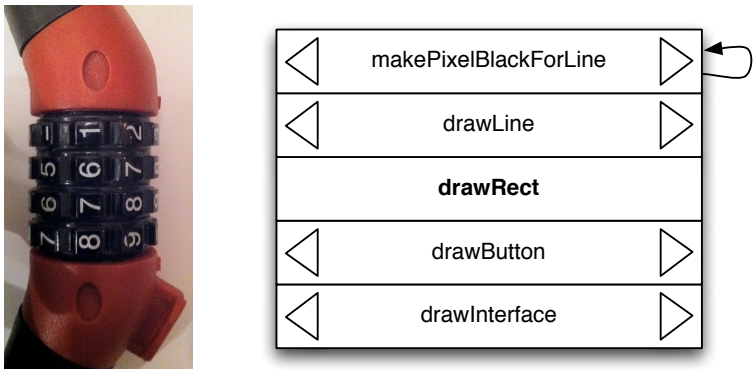


Figure 4.1: *An early sketch of how to keep the visualization simple by visualizing only one path at a time but still give users the possibility to switch between different paths. We wanted to use a combination lock metaphor to enable the user to adjust the path to their liking. The method in the middle is the focus method and can not be changed through this view, but methods in the incoming and outgoing path can be exchanged for other alternatives by using the arrow buttons.*

The difference to a combination lock is that we have one fixed point, the focus method, which can not be changed. Also all the methods/dials that are further away from the focus method than the method the user changes, change as well since they depend on the changed method. From now on we will refer to this kind of combination-lock-like user interface element as *combination lock view*.

We tried to design an auto-updating, clear an concise accessory view.

Although we wanted to change the part of the call graph that is shown, we still wanted to keep some of the properties of Stacksplorer. We wanted Blaze to be a small (in relation to the editor view) accessory view so users can still see and work with the code, since reading code is still an important part of program understanding [Schäfer et al., 2006]. And again, similar to Stacksplorer, it should be auto-updating to provide additional exploration cues to the developer without requiring interaction; thus increasing the amount of information scent the developer has access to. The visualization should also be concise and rather leave some information out than overwhelm the developer.

Different Conceptual Ideas With these basic principles in mind, we came up with three different coarse concepts for the visualization. In the beginning, we wanted to integrate Blaze

into Stacksplorer and try to visualize both sets of nodes. Our sketches for these ideas can be found in the appendix under 2—"Design", we will only briefly describe them here.

We explored three different concepts.

The first of these ideas was quite close to Stacksplorer itself. We wanted to keep the views on the left and right of the editor (see figure 3.5) and modify the Stacksplorer method views to be able to expand them to show a combination lock view for the outgoing/incoming path below/above that method (see figure 3). However, we quickly realized that this would be quite confusing since method A being below method B could mean that A and B are siblings (in the Stacksplorer part) or that B calls A (in the combination lock view).

The first idea was to make the Stacksplorer method views expand a combination lock view.

A second idea was to use Stacksplorer's approach of using the editor view as focus method and making it part of the visualization. Since the idea was orthogonal we decided to use an orthogonal visualization as well and display an outgoing path *below* the editor and an incoming path *above* the editor (see figure 4). We would still keep the Stacksplorer visualization on the left and right of the editor and one would select an outgoing/incoming path by clicking on the corresponding outgoing/incoming method. Thus, the editor would now be framed by one additional view on each side.

A second idea integrated the editor view as part of the combination lock view.

The third concept is a modification of the second one. In addition to the visualization in the second concept, one could click on each of the methods in an incoming/outgoing path to expand it into showing the implementation of this method in an editor view (see figure 5).

The third concept featured multiple editor views.

But both the second and the third concept take up a lot of screen space and are still somewhat complex. In the third idea, each editor view would probably be quite small. Therefore, we decided to take another route and develop a simpler concept.

All concepts seemed to complex.

Final concept The idea that we decided to build upon was a lot simpler. It is almost only the combination lock view introduced above, displayed in a view on the right of the editor window. We would leave out the Stacksplorer view and try just the new visualization to make it easier to understand. Also, this has the benefit of needing just half as much screen real estate as Stacksplorer.

We decided to stay close to standard UI elements and to try to create a clear, concise and space-saving visualization. Further, we wanted to leave out much of the additional functionality of Stacksplorer, like filtering for framework methods or tagging paths, to see whether it was really needed or helpful.

4.2 First complete design

We designed a first complete prototype.

We then built upon this basic concept and tried to flesh out a first complete design. This should already provide answers to questions like 'how is recursion visualized?' or 'how exactly does navigation work?'. First, we wanted to create a paper prototype to test some basic interaction with it and see whether users understand this kind of visualization.

We used OmniGraffle to create a partially interactive prototype.

But then, we realized that we would not be able to perform all the changes, which would be required after a user "clicked" on a method, quickly enough to create an experience close to reality. But this was necessary to let users try out the system to get some early feedback on which parts of the interaction and visualization are clear and which are unclear. Therefore, we decided to create a somewhat interactive software prototype using OmniGraffle[a]. We will now show and describe the first complete design we implemented as an OmniGraffle drawing.

4.2.1 Visualization and Navigation

We added several little details to the visualization but stayed close to the basic concept.

As can be seen from figure 4.2, our first design was still quite close to the basic idea of a combination lock view. We added some more features, though, to give users additional hints about the state of the combination lock. First, there are numbers inside the arrow buttons (3) giving the number of alternative methods that can be reached by using this button. Second, there are bars with arrows between the methods (2), acting like a connection between the methods. They act similarly to scrollbars in that they become smaller the more alternative methods there are, and their position indicates the position of the currently selected method in the complete list of methods. However, the arrow bars are only indicators, they are not

[a] http://www.omnigroup.com/products/omnigraffle/

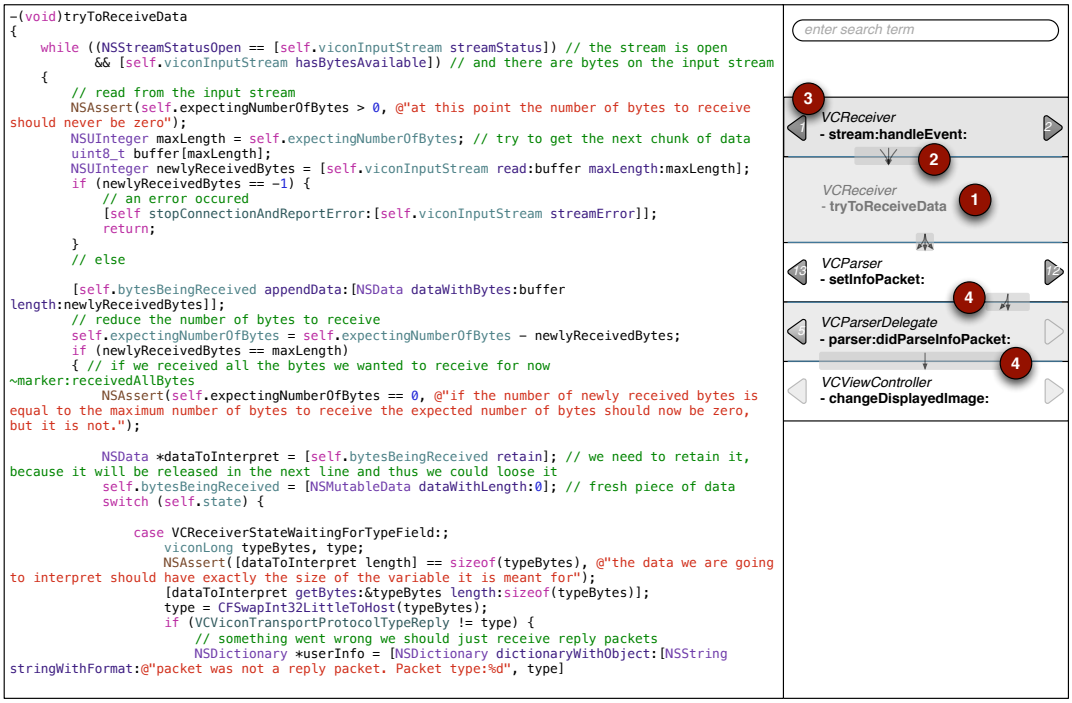


Figure 4.2: *First version of the OmniGraffle prototype used for user interviews. It displays a path with tryToReceiveData as focus method (1). In addition to the combination lock view, we have scroll-bar-like arrow bars (2) displaying the position of the currently select method in the list of alternatives and giving the calling direction (downwards). We also added numbers to the arrow buttons to show how many alternatives are left on each "side". The arrow bars have a special appearance for the edge case of the selected method being the only choice (lower 4) or being at the beginning or end of the list of alternatives (upper 4).*

meant to be dragged; thus, they are partially transparent and flat. Their appearance is meant to weaken the affordance to click them.

Icons on the arrow bars indicate the calling direction.

The arrow bars fulfill several other roles. In addition to their scroll bar like function, they also give an indication of the calling direction. We included this since we were afraid that users would not know whether a method A that appears above method B is called by B or is calling B. The reading direction is downwards, indicating that A calls B (A before B). But if users thought of this UI as a call stack, they might use the mental model of a stack; thus stacking old methods onto new methods, which would mean that B calls A (A was added to the stack after B). We decided to use a downward calling-direction, and therefore the arrows point downwards.

Arrow bars show the dependency direction.

A third piece of information the arrow bars show is the dependency direction of the methods. The root method, which defines all the other methods that occur in the combination lock view, is the focus method. If this method changes, a complete new path will be displayed and thus all other methods change. From there on outwards, each method depends on the method that is closer to the focus method. It has to change if the previous method changes since the path after the changed method changes as well.

Each method in the combination lock view depends on another method, except for the focus method.

Therefore, above the focus method the displayed callers will change if their callee changes, and below the focus method the displayed callees will change if their caller changes. In figure 4.2 `parser:didParseInfoPacket:` depends on `setInfoPacket:`; thus, if `setInfoPacket:` changes, all the methods below it will change. A new outgoing path, starting from the method that was selected instead of `setInfoPacket:`, will be calculated. This could mean that instead of `parser:didParseInfoPacket:` another method is displayed or no other method is displayed since the new path stops at the previous method. The arrow bars show this dependency by their branching; the branching part of the arrows is always at the side of the dependent method. This is consistent with the model of two trees, one for the *calls* and one for the *called-by* relationship, both starting with the focus method as the root. The dependent method is also the one which determines the position and size of the arrow by the number of alternatives and the position of the selected method in this list of alternatives.

Arrow bars can be clicked to show a menu with alternative methods.

A fourth function of the arrow bars is to act as an expert shortcut for selecting specific methods from the list of alternatives. A user can click on an arrow bar to invoke a context-menu that will show all possible alternatives for the dependent method. From this menu, they can select one of the alternatives to quickly display its corresponding path without having to use the left and right arrow buttons several times. This way, users basically click on the connection of two methods to see which other methods are also *called by* or *call* the definitive[1] of the two methods.

[1] the one that is closer to the focus method and the other method depends on

There are two special states for the arrow bars (see 4 in figure 4.2). The first one indicates that one end of the list of alternatives has been reached by displaying only two of the three branching arrows. The second one is used when there is only one alternative. In that case only one arrow is displayed, the button spans the whole width, and no menu can be invoked by clicking on it.

Arrow bars indicate when the end of the list is reached.

Another element we added to the combination lock view was a search field. This was meant to filter the displayed paths by the entered search term; thus, only paths that contained a method that matched the search term would be displayed.

The possible paths can be searched.

Instead of just showing the methods we decided to follow Stacksplorer's example and display the method name and the name of the class the method is declared in. In contrast to Stacksplorer we decided to make the method name bold, not the class name, since we thought the method name is more relevant.

Method names are displayed in bold.

Other minor changes include making the focus method bigger to make it more prominent This also visually splits the two parts of the path (incoming and outgoing path) that have different dependency directions (Gestalt Law of Proximity [Johnson, 2010]). As in the basic combination lock view we differentiate methods from each other by drawing a box around them. This also makes sure that users perceive the arrow buttons as belonging to the displayed method (Gestalt Law of Closure [Johnson, 2010]). But we also group methods by class by only alternating the background color of one method view if the class changes. This way we can make methods that belong to the same class appear as belonging together (again: Law of Closure).

We used the Gestalt Laws to guide our interface decisions.

4.2.2 Handling recursion

Another problem we explored with this prototype was how to handle recursion. Recursion means that one method either calls itself (in this case we will call this method *direct recursive*) or calls at least one other method that (indirectly) calls the first method (in this case we will call this method *indirect recursive*). A method is *recursive* if it is *direct recursive* or *indirect recursive*.

Recursion requires special treatment.

Thus, it means the method is part of a circle in the call graph. If a method is not recursive we call it *non-recursive*.

We differentiate several kinds of recursion.

If a method is recursive and there exists a path that starts in this method and ends in a non-recursive method we will call this method *may-recursive*. The opposite case is that there exists no such path, that is, all paths starting in this method end in a recursive method. We call such methods *only-recursive*. A small sample call graph with different kinds of recursion is shown in figure 4.3.

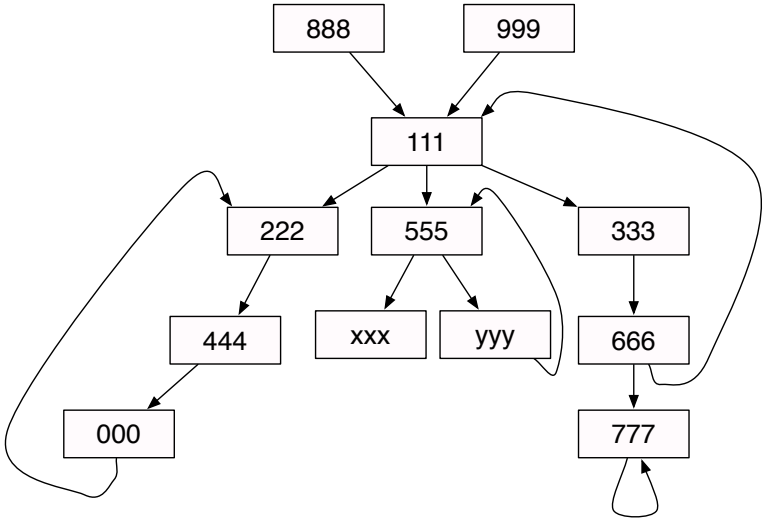


Figure 4.3: *An example call graph showing different kinds of recursion.*
777 calls itself and nothing else; thus 777 is direct only-recursive.
555 calls yyy which calls 555, thus 555 is indirect may-recursive since 555 also calls xxx which is a non-recursive method.
222 only calls 444 which only calls 000 which only calls 222. Thus, 222 is indirect only-recursive since no other non-recursive method is reachable.

The existence of recursion means that it might be possible that one method on our path calls another method that already appeared in the path before this method. We see three ways to visualize recursion in our case.

First, we could simply display an icon marking a recursive call as recursive and stop the path there. This saves space and visual clutter. However, it has the disadvantage of not showing which method is called recursively.

Another way to visualize this would be to just display some connection (for example an arrow) from the method further down the path back to the method were the recursive call starts again. For one-step recursion this would just be an arrow pointing back to the calling method itself (an example for direct recursion can be seen in figure 4.1). This has the advantage of not duplicating recursive methods but introduces visual clutter through additional arrows connecting methods in the combination lock view. Another disadvantage is that paths that include a circle can not be displayed properly. For example, in figure 4.3 if we had 555 as focus method we would see yyy below it and than an arrow back from yyy to 555 to show the recursive call. However, a possible path would be to start in 555, then call yyy, then 555, then xxx. This path could not be displayed using this visualization. The visualization still indicates the existence of this path, but it does not show it explicitly.

Extra arrows could be used to visualize recursive calls.

A third way to visualize the recursion is by *not* visualizing it in a special way and simply continue the path through recursive calls thus potentially making the displayed path a lot longer. Obviously we can not do this strictly and have to stop at some point, otherwise we would have infinite paths. It also has the disadvantage of possibly showing the same method several times, which may confuse users. But this way we can explicitly visualize paths with circles.

One could also just display recursive method calls repeatedly.

The Chosen Way We decided to use the third way with some modifications since it gives users the ability to 'unwind' several recursive calls into one path thus making understanding easier. Also, it is closer to what the user would see on the call stack in the debugger. We hope that this way of visualizing the possible paths can save some trips to the debugger by just manually and statically analyzing the potential paths.

We combine the approaches.

The modifications we made to the third way are that we always try to find a non-recursive path. Thus, for the call graph in figure 4.3 with 111 as focus method we would show the outgoing path 111-555-xxx. If the user then chooses yyy instead of xxx Blaze would show 111-**555**-yyy-**555**-xxx. Also, the occurrences of 555 would be highlighted to notify the user that this is a recursive call with 555 appearing several times. The user could then choose yyy instead of xxx again.

We always try to find a non-recursive path.

For only-recursive methods we display the first method in the cycle twice and all others only once.

This is only possible in the case of may-recursive methods. If there is an only-recursive method we differentiate between direct recursive and indirect recursive methods. For direct recursive methods we will just display a simple icon at this method view indicating that it calls itself. Thus, for the displayed call graph with 333 as a focus method, the outgoing path would be 333-666-777 with an icon indicating the recursion shown at 777[2]. For indirect recursive methods, we would show all nodes in the circle at least once and the first node in the circle a second time at the end of the path. For this last method, we would show an icon again, but use another icon than in the direct recursive case to differentiate the two cases. Again, Blaze would highlight methods that appear multiple times to show the recursion.

4.3 User Interviews

With this first complete prototype, we wanted to conduct a small user study to verify that users understand our design. We added some interactive elements to make it easier for them to imagine what the final version would look and feel like.

We made some elements of the prototype interactive.

For the prototype itself, we made the focus method itself and one other method clickable to simulate the 'clicking on a method and navigating to the corresponding implementation'-behavior. Clicking on any of the two interactive methods would display their implementation in the editor view on the left. The other element we made interactive was one of the arrow bars. Thus, one could click on it to display the menu with alternatives (not shown in figure 4.2).

We created an abstracted but more interactive version to test the recursion cases.

We also created a set of views to test the switching of methods with the combination lock view and the visualization of recursion. These views were a lot more basic and just had the left-/right arrows with numbers and some downwards-pointing arrows to indicate the calling direction between the methods (see figure 4.4). They also had the highlights for methods that appeared several times due to recursion and two icons for indicating one-step and multiple-step recursion.

[2]This would only happen if there is no edge between 666 and 111, otherwise the path 333-666-111-555-xxx would be taken, so the example is not completely correct.

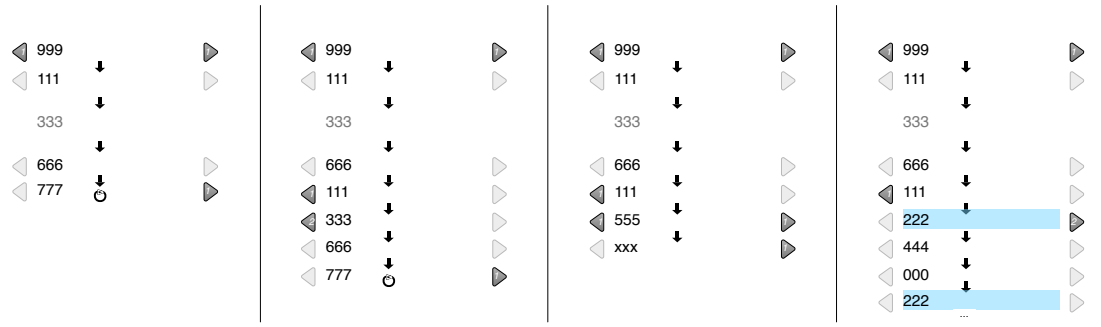


Figure 4.4: *A sequence of states of the combination lock view resulting from using the left and right arrow buttons. 333 is the focus method (the one without left/right arrows), the underlaying call graph whose methods are displayed is shown in figure 4.3. The transition of the views is as follows: Start left, click on right arrow near 777 to reach the second state. Click on the left arrow near the second 333 to reach the third state. Click on the left arrow near the second 111 to reach the fourth state.*

This combination lock view implementation could show most of the switching of methods possible in the call graph in figure 4.3 (we did not include arbitrarily long circles for example). A sample sequence of states of the combination lock view after clicking on different arrow buttons is shown in figure 4.4.

4.3.1 Study Design

Due to the limited interactivity we decided to test our prototype using an interview-style study. We showed the prototype to users and explained as little as possible about it. All we explained was that we were working on a software visualization and navigation tool and that the view on the right side of figure 4.2 was our prototype and the left view represented the code editor of Xcode. Then we started with basic questions like "what does the view on the right show" and then slowly increased the specificity of the questions, explaining more about the prototype when necessary.

We conducted an interview-style study.

We first asked questions about what different parts of the user interface mean and then asked the users to describe what they would do to reach certain goals (e.g., to navigate to the implementation of a method). In the end we showed them the simplified interface for visualizing recursion and asked them to switch the methods in a way to display certain stacks. This

We first asked questions and then let them try out the recursion visualization.

was meant to let us observe them using the combination lock interface to see whether there were any problems.

The questions used during the interviews can be found in the appendix under 5—"User Interviews". The prototype used is shown in figure 4.2. The prototype used for the recursion questions is shown in figure 4.4 and displayed the call graph shown in figure 4.3. We performed these interviews with 5 fellow male computer science students. During the interviews we took notes and recorded the conversation.

4.3.2 Results

From the answers to our questions and from the reactions to our prototype we learned a few things.

Answers From the First Three Participants For example, the numbers in the arrow buttons were not well understood. One participants guessed that it would give the number of the line im which the method is called. Another participant first thought that the number in the arrows would be the index of the method a click on the arrow would lead to. Thus, when a 12 was displayed in the right arrow it would mean that a click on this arrow would show the method with index 12. Another participant first thought the number would give the number of calls to this method, but then thought about it and came to the correct conclusion.

Numbers in the arrow buttons are not clear.

One participant also had difficulties recognizing that `tryToReceiveData` is the focus method and later said that they would expect a more explicit connection between the code and the path view. He said, he did not recognize that the method displayed in the editor was the same as the grayed out method (which represents the focus method).

It was not easy to recognize the focus method.

None of the first three participants had problems with the behavior of our visualization when encountering recursion. But one of them did not understand our icon for one-step recursion (a circular arrow).

All of the first three participants understood that the arrow bar position represented the position in the list of alternatives.

One participant thought the width of the arrow bar would represent the size of the implementation of the method, the other two (correctly) assumed that it would represent the length of the list of alternatives (the smaller the longer the list).

The arrow bar position was correctly understood.

When asked what they would do to invoke a menu to show all alternatives, two participants said they would click and hold on the method view. One participant said they would right click on the method view. None of them even thought about clicking on the arrow bar. Even after this question, when asked what they think might happen when they do click on the arrow bar, none of them thought of the possibility that it might invoke the alternatives menu. One participant thought it would just scroll in the list of alternatives (since the bar looked similar to a scroll bar).

None of the participants guessed correctly how to invoke the alternatives menu.

Changes to the prototype Since the interviews with the first three participants confirmed a few things that had already been pointed out to us as potentially problematic by other researchers, we decided to change the prototype. The changed prototype is displayed in figure 4.5 and 6.

Visually the biggest change is the more prominent display of the focus method. It now has a metallic background and is thus darker than the other methods and splits the complete view in half. We decided to give the focus method a fixed position (the position can be changed but it won't change automatically as it would have before) to let users make use of their spatial memory. This way, they can always know that the focus method is in the middle and the outgoing path is below and the incoming one above it.

We made the focus method a lot more prominent.

Since none of our users understood that one could click on the arrow bar, we made it look more like a button now. To remove some of the functional overload on this UI element we no longer used it for displaying the position in the list of alternatives.

We removed some functional overload from the arrow bar.

Instead we introduced another UI element for displaying this information. It was inspired by the dots on the iPhone's home screen, which show what page one is looking at, but in contrast to it, always uses the full width available (see figure 4.5). This UI element is displayed near the arrow bar, which invokes the corresponding alternatives menu (using the Gestalt

We use an iOS page indicator instead of a scroll bar.

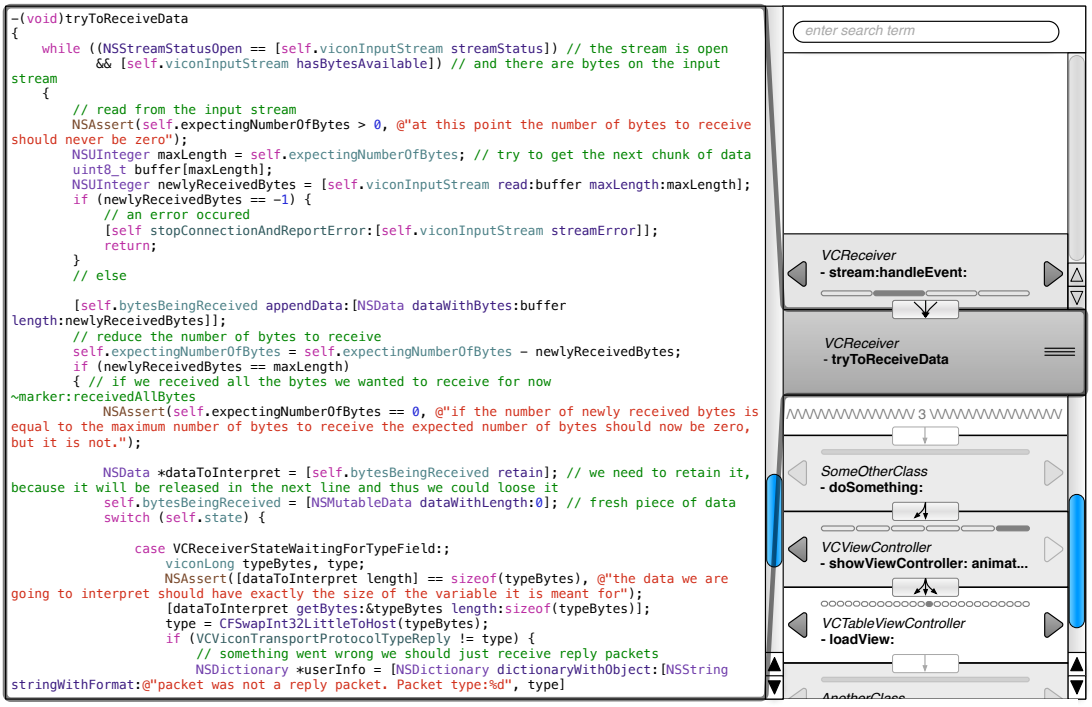


Figure 4.5: *This is a new prototype that was designed after the first few user interviews and after some discussion with fellow researchers. The visually biggest change is the more prominent display of the focus method. It now has a metallic background and is thus darker than the other methods and splits the complete view half. The outgoing path is still below the focus method and the incoming one is still above the focus method but now the position of the focus method can be changed to distribute the available space between the two halts of the path differently. Also the arrow bars between the methods are no longer used like scrollbars and instead they are now always in the middle and have a more button-like (raised) appearance to encourage users to click on them. The zigzag-like drawing with the 3 below the focus method shows that three of methods on the outgoing path are scrolled below the focus method and are not shown.*

Law of Proximity [Johnson, 2010]). This leads to it being placed at the bottom for incoming methods and at the top for outgoing methods. This has another helpful effect: It makes incoming method views appear different from outgoing ones and thus separates them into two distinct groups (Gestalt Law of Similarity [Johnson, 2010]).

We introduced a view that indicates when methods are hidden.

Since the incoming and the outgoing path can now be scrolled independently, it might happen that some methods are scrolled "under" the focus method. To make sure no user is confused and makes incorrect assumptions about the connection of methods, we display another view in that case. It is a zigzag-like

view, which is meant to represent the "folding-away" of methods and gives the number of currently hidden methods.

We also changed the menu that would have been displayed when one clicks on a button into a popover since we thought this would fit the purpose of the button better. The popover can be seen in the appendix in figure 6.

More Interviews We then conducted two more interviews using our improved prototype. We changed some of the questions to also ask for the drag handle and the zigzag-view. We also decided to leave out the part about the recursion since we felt we did not learn much from the previous participants. The changed set of questions can again be found in appendix 5—"Questions for new prototype".

We conducted two more interviews.

More Results When asked how they would navigate to a method's implementation, all five participants said they would simply click on the method so this seems to be the correct decision.

When asked about the sorting of the alternative methods, participants expected the outgoing methods to be sorted according to their appearance in the calling method (i.e., if A is called before B, A should be sorted before B). For the incoming methods most expected alphabetic sorting, since there is no sensible sorting by the code structure.

Incoming methods should be sorted alphabetically, outgoing ones according to their call order.

Two of the five participants assumed the calling direction was upwards (stack-like) *although* there were the arrows pointing downwards. However, both later discovered the arrows on the buttons and concluded that the calling direction is downwards. But due to the different initial assumptions, we plan to add a preference in a later version of Blaze. This would allow users to adjust the calling direction to their liking.

Some participants assumed an upwards calling direction.

Another part of the UI that was not well understood was the two-arrow icon of the arrow-bar button. Two of the five participants did not understand what it meant although they did understand why it sometimes had three and sometimes just one arrow in it.

After the change of the arrow bar design both participants

The arrow bar was still not well understood.

said they would think this UI element was clickable but one said he had no idea what would happen if it was clicked. The other one guessed (correctly) that somehow all the alternatives would be displayed, but instead of a menu they would have liked a graph like display. Both participants understood that the zigzag view meant that some methods were omitted but neither of them liked its design.

5 Software Prototypes

"What a computer is to me is the most remarkable tool that we have ever come up with. It's the equivalent of a bicycle for our minds."

—Steve Jobs (1991)

We created software prototypes of two different tools to be able to compare them to each other. The first one is our new tool, Blaze, the second one is an implementation of the Eclipse Call Hierarchy.

5.1 Blaze

For the final design, we tried to apply as much as possible from what we learned during the interviews and then implemented a software prototype to evaluate the concept in a more realistic scenario. Figure 5.1 shows the prototype in action, being used by one of our user study participants.

We simplified the UI again for the software prototype.

For the implementation, we removed the special case of the arrow bar with just two arrows since it was not understood. We also just used an unmodified page indicator view as found on the iPhone since we thought that the differing width of the indicator dots we used in the design introduced additional visual clutter. For the arrow bar buttons we decided to make them a bit smaller to not take away to much attention from the more important parts of the UI. For the same reason we used a flat button design.

We left out some small features due to time constraints.

We were not able to implement the zigzag-view and the exact version of our alternatives popover from the design due to time constraints. The bar at the top of the editor view is a relict from Stacksplorer, which was used to tag methods. In our implementation, it does not have any functionality, but we

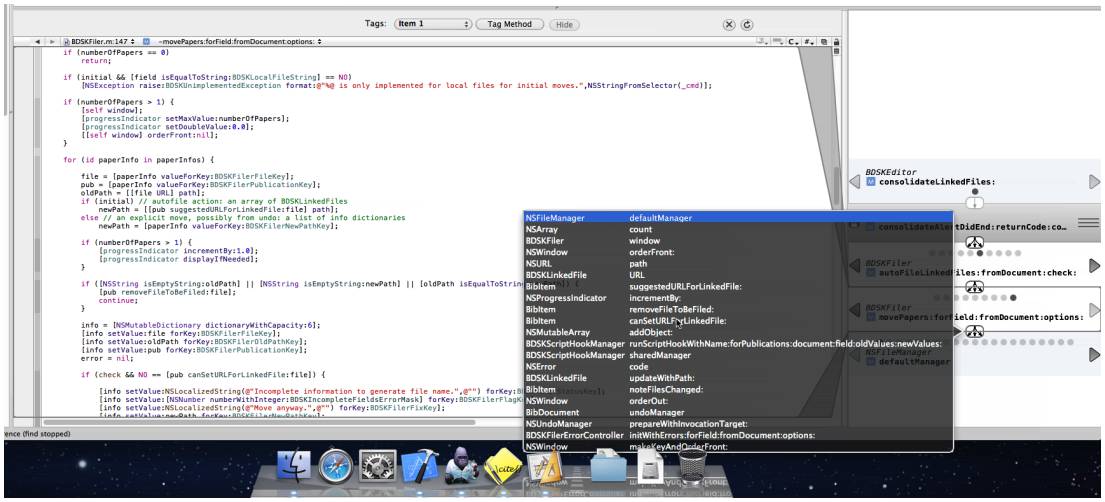


Figure 5.1: *A screenshot from one of our user study participants using the software prototype. The participant first looked around exploring the source code and then found the method* `consolidateAlertDidEnd:returnCode:contextInfo:`. *There they decided this would be a good starting point and locked it. Then they explored the outgoing paths from this method further. In the given screenshot they have looked at the code of the method* `movePaper:forField:fromDocument:options:` *and are now looking at that method's callees using the alternatives popover invoked by clicking on the button below the* `movePapers...`*-method. The bar at the top with the "Tag Method" button is a relict from Stacksplorer.*

decided to keep it to be more comparable to Stacksplorer and to have the possibility to later include the tagging feature as well.

The focus method can be locked to prevent it from changing automatically.

One feature that was not included in the design (but already planned) was the possibility to lock the focus method. By default, the focus method changes automatically when the selection in the editor changes. This provides additional context and cues to the developer and increases the access to information scent while they are searching for a starting point. But when they found such a starting point they usually wanted to be able to explore all relationships from this node and most importantly be able to backtrack to this starting point. To make this possible the focus method has to be transformed into a fixed point that can not be easily changed and is a safe haven to return to (see the model of programmer navigation introduced in 2.1.1—"Programmers Look for an Anchor Point to Start Exploration"). We thus introduced a lock-button to the focus method.

By default, the focus method still automatically updates. But

as soon as it is locked, it will no longer update and will always stay the same while the programmer can safely explore all the surrounding methods in the call graph. Of course, the overlay showing which method is selected in the editor still updates (see figure 5.1).

A locked focus method provides an anchor point for the programmer.

5.2 Call Hierarchy

In the Stacksplorer study, Stacksplorer was compared to plain Xcode. However, we felt this might not be a fair comparison since it seems to be easy to develop a tool that will somehow improve the experience compared to not using such a tool. The important question is whether our new tool is better than existing ones. For this reason, we decided to compare Blaze to the Eclipse Call Hierarchy (see figure 3.3). But it would be almost impossible to find a way to fairly compare Eclipse with the Call Hierarchy to Xcode with Blaze since both IDEs are designed completely differently and are also designed for different languages. Therefore, most of the differences we would find would most likely be due to differences other than the call graph exploration tools.

We wanted to compare Blaze to the Eclipse Call Hierarchy.

Instead we decided to implement our own version of the Call Hierarchy. It would run as an Xcode plugin and use the same parser and other backend code as Stacksplorer and Blaze. This plugin was developed by one of my advisors, Jan-Peter Krämer, with my help and according to the requirements laid down by me. This plugin can be seen in appendix 7.

We developed a Call Hierarchy Xcode plugin.

The Call Hierarchy makes use of the fact that the subgraph of incoming (indirect) calls can be transformed into a tree. The same is true for outgoing calls. Following these inherent properties of the underlying data, the Call Hierarchy has two views, one for the incoming tree (caller view) and one for the outgoing one (callee view). Consequently, a tree view is used to display the corresponding tree in each of these views. A tree view is basically an indented list (see 3.1.1—"Basic Tree Visualization") with a small arrow in front of each element that the user can click on to expand or collapse the children of this element.

The Call Hierarchy uses tree views to display to display parts of the call graph.

The Call Hierarchy displays either the caller view or the callee

view never both at the same time. The user can switch between the two views with two buttons in the lower left.

The Call Hierarchy does not update automatically.

Choosing a method to be displayed as root node in the Call Hierarchy works by right-clicking on it to invoke a context-menu and then choosing the "Show Call Hierarchy..." menu item. In contrast to Stacksplorer and Blaze, just selecting another method in the editor will *not* change what is displayed in the Call Hierarchy.

Clicking on a method navigates to the method implementation of the clicked method or its parent.

When the Call Hierarchy displays methods, the user can click on each of these methods to jump to the location in code that represents the connection of the clicked method to its parent in the tree view. Thus, in the *callee* view a click on the method will jump to the parents implementation and show the call to the clicked method. In the *caller* view, a click on the method will jump to the clicked methods implementation and show the call to the parent[1] method. In each case, the call will be highlighted in the editor.

If a user just wants to go to a method implementation they can right click on that method in the Call Hierarchy and choose the "Open" menu item. If they want to make a method displayed in the Call Hierarchy the new root method (focus node) they can right click it and choose "Focus on this node". The complete described behavior is consistent with the Eclipse Call Hierarchy.

5.3 Backend

Both plugins use the same backend as Stacksplorer.

We did not change the backend that parses the code and calculates the call graph that is displayed. Therefore, Blaze, the Call Hierarchy, and Stacksplorer all use the same underlying parser. Implementation details and information about how to write Xcode plugins can be found in Krämer [2011].

[1] The parent according to the tree view.

6 Evaluation

"Facts are meaningless. You could use facts to prove anything that's even remotely true!"

—Homer Simpson

To test how well Blaze works for programmers compared to existing tools, we ran a user test in which we compared it to another plugin, which works similarly to the Eclipse Call Hierarchy. We also used the results from a previous study to compare both tools to another similar research prototype and a default Xcode installation. During this chapter, we will describe the experiments we did, the results of those experiments, and whether they confirm our hypotheses.

We compare Blaze, the Call Hierarchy, Stacksplorer, and plain Xcode.

6.1 Experimental Setup

We designed our study to be similar to the study used to evaluate the Stacksplorer Xcode plugin developed previously. In fact, we tried to be as close as possible to the previous study by Krämer [2011] to be able to compare our results to this previous study. We will now describe the setup of this study again and describe the differences to the old study. More detailed reasons for the study design can be found in Krämer [2011].

We use the same study design as in the study comparing Stacksplorer to Xcode.

6.1.1 Conditions and Tasks

The previous study used a within-groups design with two sets of tasks. Subjects would solve one set of tasks with Stacksplorer and one without. However, later analysis showed that the tasks were not as equal as one had hoped [Krämer, 2011]. Also, working on two tasks increased the time required from each participant to nearly two hours. For these two reasons, we

We do a between groups study.

decided to do a between-groups study using just the first set of tasks, which had shown differences between participants more reliably [Krämer, 2011]. A between-groups study means that each participant will work under only one condition. That is, our participants would either use the Call Hierarchy or Blaze. If we just use the data for the first set of tasks from the previous study, it will also be a between-groups study since each participant completed the task either with or without using Stacksplorer.

Tasks were to be done in the source code of BibDesk.

Tasks All tasks were concerned with solving problems in the BibDesk[a] source code. BibDesk is an open-source reference-management application that can be used to create .bib files for LaTeX-documents. It contains 88 000 SLOC and roughly 400 files [Krämer, 2011]. We used the same revision as in the last study, revision 17029. We also included the same small changes that were meant to make a task in the second set more interesting.

The first task focuses on navigation and exploration.

The first task of the chosen set was designed to test programmer navigation and code exploration. We explained a specific feature to participants, which moves PDF files and sorts them into folders. We asked them to change the implementation of this feature such that for every file that was moved by BibDesk the string "TRIAL" would be added in front of the filename of the moved PDF. We told them that the class `BDSKLinkedFile` is the class that is used to represent those PDFs. To save some time, we spared participants from writing code and instead told them it would be enough to tell us which method they would modify.

The second task focuses on finding side effects.

The second task was designed to make participants look for side effects. In this task we proposed a solution to task 1. The solution included changing the return value of a method. We then asked them what side effects this change could have. To limit the task to a sensible time we told them we are only interested in side effects that have a visible effect in the graphical user interface (GUI). If participants presented a side effect, we asked them if the side effects they named were all side effects on the GUI or if they wanted to continue searching.

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

The actual task descriptions that we gave to users can be found in appendix 6.1—"Study Setup". The conditions (independent variable) were Xcode with Blaze and Xcode with the Call Hierarchy. A detailed description of Blaze can be found in 4—"Design" and in 5.1—"Blaze". A detailed description of the Call Hierarchy can be found in 5.2—"Call Hierarchy".

6.1.2 Participants

We again recruited graduate and undergraduate students who had at least some basic experience with Xcode and Objective-C. By hiring only students we hoped to reduce the impact of different levels of programming experience and capabilities. According to Bragdon et al. [2010] experience levels between students are less varying then those between professional programmers. In contrast to the previous study we did not include professional software developers this time. However, the professional software developers did not perform noticeably faster or better than other participants in the last study so this should not be a problem either.

We recruited student developers with experience in Objective-C and Xcode.

In the last study, only programmers that had experience with programming for Mac OS X were included. However, Xcode and Objective-C are also used to develop for iOS and thus we opened the study up for iOS developers as well. Programming for the Mac and iOS is similar in many ways, although the framework differs in some ways. Our tasks did not require specific Mac knowledge so we were confident that this would not influence the performance greatly. Since our participants were programmers who were mainly programming in Objective-C, most of them did not know Eclipse or other IDEs enough to know a plugin similar to our Call Hierarchy.

We employed Mac and iOS developers.

Statistics In the study, we tested 18 participants, 17 male, 1 female. All of the participants were computer science students, out of which 3 had already graduated and were now working on their PhDs. The average experience with Xcode and Objective-C was 2.9 years ($SD = 1.9$), and on average they were spending 11.5 hours per week programming ($SD =$

10.8)[1]. 7 out of our 18 participants said they used BibDesk before but none of them had seen the source code before the test.

6.1.3 Methodology

Participants were not allowed to analyze a running instance of BibDesk through trace statements, etc.

As with the complete study design, the methodology was the same as in the last study. Participants were not allowed to use any tools to analyze a running instance of BibDesk (like a debugger or inserting logging statements into the code and then compiling and running it to log information.). In fact, they were not even allowed to compile the code. To make up for this, a compiled version of BibDesk was provided so participants could investigate the functionality of the program. All other tools Xcode provides could be used.

We explained the plugin participants used (either Blaze or the Call Hierarchy) using a small sample project. We then allowed participants to play around with the sample project and the tool until they had no more questions. We explained all the features of the plugin that were described in chapter 5—"Software Prototypes".

Due to a bug in the parser, when clicking on a method both plugins would sometimes display the declaration of the method in the header file instead of the implementation. To overcome this bug we told participants that they could double-click a method while holding the command key down to jump to a method's implementation. This is a standard feature of Xcode.

We asked participants to think aloud.

Before participants started working on the task, we asked them to think aloud so we could better understand what they were thinking and doing and where they had problems. However, we did not remind them to think aloud while working on the tasks to make sure we did not distract them. Questions about the BibDesk source code would not be answered, but questions about Cocoa and Xcode would be answered (to the best knowledge of the experimenter).

[1] Participants were quite unsure how much time they spent on programming and said it varied a lot. They often gave a range like 5-10 hours. In such cases we used the average (7.5 in this case) to calculate the mean and standard deviation.

We measured the time participants took until they found a solution and whether this solution was correct (the dependent variables). We recorded the screen content, the participants voice and their upper body (the latter two using an iSight camera attached to the top of the display) using Silverback[a]. Two participants asked to not have their voice and face recorded, thus we just recorded the screen.

We recorded the screen, participants' voice, and upper body.

We used a Mac Pro computer comparable to the one used in the last study. Again participants used a 23" screen with a 1920×1680 resolution.

6.1.4 Postsession Questionnaire

After participants had finished the tasks, we wanted to find out their subjective opinion of the tool they used. To measure their satisfaction with the respective plugin[2], we used the System Usability Scale (SUS) by Brooke [1996] combined with 6 additional questions specifically targeted at the plugin use cases. Again, we used the identical questionnaire as used in the Stacksplorer study, even up to the name "Stacksplorer" for the plugin. We just called whichever plugin the participant used "Stacksplorer" to make sure they were not influenced by any differences in the questions. We also did not tell them that the Call Hierarchy version was not designed by us but a copy of the Eclipse version and instead told them it was our design to prevent them from rating the "new" tool or "our" tool any better. The full questionnaire can be found in appendix 6.1—"Post-Session Questionnaire".

The postession questionnaire consisted of the SUS with 6 additional questions.

6.1.5 Differences Between the Two Studies

Although we tried to make the second study reflect the first study as closely as possible there were some differences in the two studies. There was also one difference that was by design.

The difference that was by design of the study was in the rating and how we let participants work on task 2. We noticed

[2] remember that each participant only used one of the plugins

[a] http://silverbackapp.com/

that it took participants on average only 6 minutes in the Stacksplorer condition to complete the task. Even without plugins using just Xcode it took them only 9 minutes on average out of the 15 minutes allocated for this task. So there was some time left, which we decided to use.

We rated task 2 slightly differently than in the last study.

In Task 2 participants were asked to find side effects of a given change on the graphical user interface (GUI). During the first study, the task was considered to be finished when they found one side effect in the GUI. In our new study we decided to ask participants to find *all* side effects on the GUI and asked them to stop only when they thought they found *all* side effects. This way we could observe them using the given plugin more intensively. We also felt this was a more natural situation and the tools were up for the task since both tools make it easy to explore several paths from a given starting method.

Task descriptions were the same, though.

However, it is important to point out that task descriptions were exactly the same. The question in Task 2 is "Which effects would this have in the UI". So it is asking for the plural, and the same question was used in the last study. The difference is the reaction of the experimenter to the first solution the users present. In the last study, when the participant presented a side effect, the task was stopped. During the new study we asked them whether they were sure they found all side effects.

To be able to compare the results to the previous study we recorded two times: The time until they found the first side effect on the GUI and the time until they were convinced they found all side effects on the GUI. We used the first time in the following comparison to be consistent with the previous study and used the second time for the within-study comparison in the previous section.

The different task rating also led to rating side effects.

This difference in rating leads to a side effect in the rating of the correctness of the solution: One participant of the Call Hierarchy condition first found a correct side effect on the GUI which made him have a correct solution for the comparison between the two studies. However, he later found side effects that did not exist, leaving him with an incorrect solution. Thus, the Call Hierarchy has a lower success rate in the within-study comparison than in the between-studies comparison.

Not by design was the fact that the experiments were conducted by a different person (the author of this thesis) and

in a different room. However, we made sure the setup was the same and we presented the questions in the same way and gave the same kind of support to participants. To do this, we talked to the previous experimenter and reviewed the notes of the study design. Of course, there might still be differences in how the experimenter interacted with the participants simply because they were not the same person. There are also slight differences in the participants we recruited. This is described in detail in section 6.1.2—"Participants".

The study was conducted by a different experimenter.

6.2 Quantitative Results

In this section, we will describe our quantitative results in detail, explain which ones were significant, which ones were not, and what they tell us. Since we rated Task 2 a bit differently than in the last study and there are more threats to validity when combining the results of both studies, we split our evaluation of the results in two parts.

First, we will discuss the results of the comparison of the Blaze plugin and the Call Hierarchy plugin without referring to the old study. In the second part, we use the results obtained from applying the rating criteria of the old study to our new study and compare these to the results of the old study to compare the four conditions of Xcode without plugins, Xcode with the Stacksplorer plugin, Xcode with the Call Hierarchy plugin, and Xcode with the Blaze plugin.

We first compare just the results from our study.

Wording Since we did not allow participants to implement their proposed changes and test them, or even change or debug the program in any way, it would not be realistic to say our plugin helps programmers to solve maintenance tasks faster. Solving such a task involves a lot more than just reading source code and thinking about a solution. However, the plugins help participants to form a hypothesis of how to solve such a task; thus, instead of calling the time the participants needed to give an answer "time to solution", we will call it "time to hypothesis". If this hypothesis turns out to be correct we call it "time to correct hypothesis".

We use *time to hypothesis* instead of *time to solution*.

We excluded two participants due to technical difficulties.

Excluding Two Participants The results of two participants were excluded from the quantitative evaluation. In the first case, the experimenter made a mistake and accidentally gave out some information about the tasks during the introductory code reading. This led the participant to already investigate the features that should be explored. Thus there was an unfair advantage for this participant and we decided to exclude all the quantitative results of this participant from the evaluation. In another case, the parser had not parsed the call graph correctly for task 2. In this case, we only excluded the data from Task 2 from the evaluation, since there were no problems during Task 1. In the first case, Blaze was used, in the second case the Call Hierarchy was used.

6.2.1 Results of the New Study

In our study we wanted to explore the following three hypotheses:

H1 More programmers come to a correct hypothesis of how to solve a task that requires browsing and understanding previously unknown source code with a time-constraint using Blaze than using the Call Hierarchy.

H2 Programmers can solve tasks that require browsing and understanding previously unknown source code more quickly using Blaze than using the Call Hierarchy.

H3 Programmers can identify side effects of changes made to unknown code more quickly using Blaze than using the Call Hierarchy.

The first hypothesis is concerned with the rate of correct hypothesis for maintenance tasks in general. The second and third hypothesis are aimed at how quickly programmers can solve these tasks. The second one concentrates more on the code exploration and understanding which is required more in Task 1, the third one is focusing on side effects of changes, something that was specifically tested in Task 2.

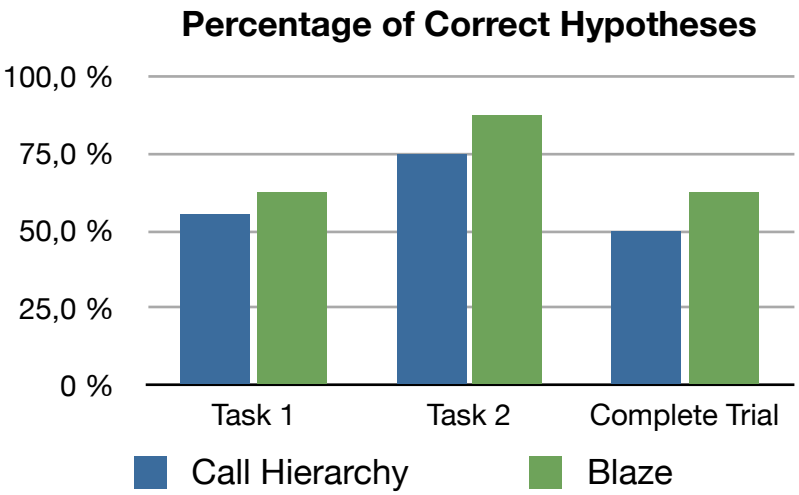


Figure 6.1: *The percentage of correct solutions grouped by the tasks and the plugin used.*

Task Success Rates

Figure 6.1 shows the percentage of participants who found a correct solution during the given time frame for each of the subtasks and the complete task. The complete task was considered to be solved correctly if both subtasks were solved correctly. Although the success rate was higher with our Blaze plugin, the difference for each of the tasks is small and a Fisher's Exact Test did not show any significant differences (see table 1).

There were no significant differences in success rate.

Task Completion Times

We subdivided the task completion times in two (non-disjoint) groups. First, we compared all task completion times ("time to hypothesis"). Then, we compared only the times of the participants which solved the task correctly ("time to correct hypothesis").

We assume that the underlying population for the "times to hypothesis'' and "times to correct hypothesis" is normally distributed and will use t-tests to compare the results. To confirm this assumption, we did Shapiro-Wilk tests for each of the samples. The results of those are shown in table 2. Only

We assume a normal distribution and test this assumption with a Shapiro-Wilk test.

the times to hypothesis for Task 1 in the Call Hierarchy condition are significantly different from a normal distribution. Although a Kolmogorov-Smirnov test shows no significant difference ($D = 0.232, p = 0.719$) to a normal distribution with the same mean and standard deviation, we will do a Mann-Whitney's U test for this case, in addition to the t-test, since this test does not assume normality.

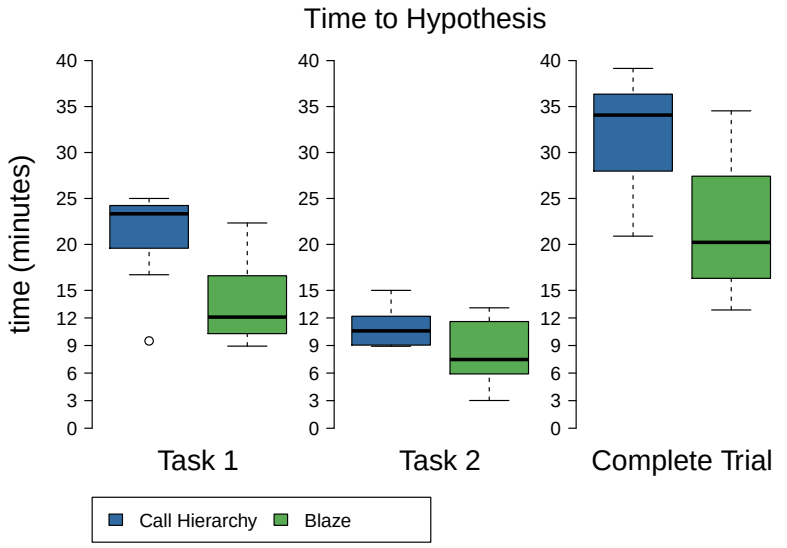


Figure 6.2: *The average time to hypotheses by task and condition in minutes. Participants using our plugin are several minutes faster in each task.*

Blaze significantly decreases the time to hypothesis.

Time to Hypothesis Figure 6.2 shows that participants had a hypothesis several minutes earlier with Blaze than with the Call Hierarchy. The median time for task 1 for Blaze is even roughly half the median time of the Call Hierarchy. Using a one-sided Welch's t-test we found a significant difference in the time to hypothesis in each of the subtasks and the task as a whole. The exact results of the t-tests can be found in table 6.1.

We also performed a Mann-Whitney's U test.

Since a Shapiro-Wilk test showed a significant difference between the times of the Call Hierarchy condition in Task 1 and a normal distribution we did an additional Mann-Whitney's U test which does not assume the normality. The results of this

Task	t	df	p	Cohen's d
Task 1	3.12	15.0	**0.003**	1.51
Task 2	1.82	11.5	**0.048**	0.91
Complete Trial	2.98	13.5	**0.005**	1.49

Table 6.1: *Results of one-sided Welch's t-tests comparing the time to hypothesis of the Call Hierarchy to that of Blaze for each task and the complete trial. Given are the t-value, the degree of freedoms (df), the p-value and the effect size as Cohen's d. The results show significant differences and large effects for each task.*

test confirmed the significance of the difference ($U = 61.5, Z = 2.46, p = 0.006$).

We also calculated the effect size (Cohen's d, see table 6.1). A value of 0.8 or higher of Cohen's d is usually considered to be a large effect size, so our results show a large effect for each of the cases.

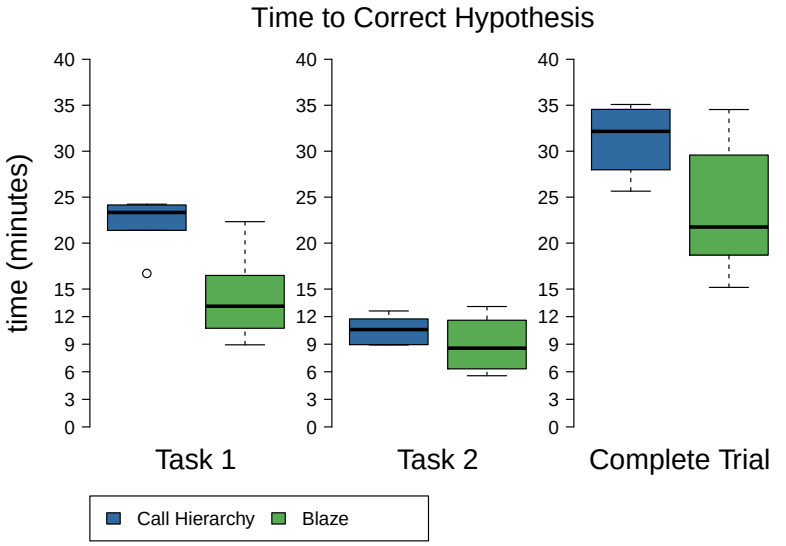


Figure 6.3: *The average time to a* correct *hypothesis by task and condition in minutes. Participants using our plugin are several minutes faster in each task.*

Time to Correct Hypothesis Figure 6.3 shows that not only the time to hypothesis is smaller with our plugin but also the

The time to correct hypothesis is only significantly faster for Task 1.

time to a *correct* hypothesis is smaller. However, using one-sided Welch's t-tests again, we could only confirm a significant effect for Task 1 with Blaze outperforming the Call Hierarchy. The exact results of the t-tests can be found in table 6.2.

Task	t	df	p	Cohen's d
Task 1	2.77	6.5	**0.015**	1.75
Task 2	1.17	9.1	0.14	0.62
Complete Trial	1.76	6.3	0.063	1.10

Table 6.2: *Results of one-sided Welch's t-tests comparing the time to* correct *hypothesis of the Call Hierarchy to that of Blaze for each task and the complete trial. Given are the t-value, the degree of freedoms (df), the p-value and the effect size as Cohen's d. The results show significant differences for Task 1 and large effects for each task.*

Although non-significant, the p-value of the t-test for the complete trial is close to 0.05 and thus indicates that developers had a correct hypothesis earlier using Blaze. The results for Task 2 are inconclusive. Again, the effect size is large for the complete trial and Task 1 but only medium for Task 2.

Discussion

We think the non-significant results for time to correct hypothesis are due to the small sample size.

Both tools seem equally well suited to find a correct hypothesis, so we have to reject H1. Comparing the time to hypothesis for both tools confirmed H2 and H3, but the results for the time to correct hypothesis only supported H2. We think the non-significant results are probably due to two to causes. First, the differences between the conditions are actually smaller than in the time to hypothesis comparison, but not a lot. The bigger problem for the significance is probably the reduced sample size (only the correct answers) since the significance is dependent on the sample size. But we cannot be sure until we have conducted further tests to confirm the results.

We can confirm H2 and H3.

However, even forming an incorrect theory more quickly will probably improve the overall time to solve a maintenance task since the programmer can then test this hypothesis earlier. Therefore, we think we can confirm H2 and H3 overall. But the support for H3 is weaker. This is understandable since the differences between the Call Hierarchy and Blaze were smaller

in task 2. We think this indicates that the Call Hierarchy is also helpful for finding side effects but less helpful for exploring and navigating in unknown source code.

6.2.2 Comparing the Results of the Old and the New Study

From the results in the previous section we already know, that programmers could find a hypothesis significantly faster with Blaze than with the Call Hierarchy. From the previous study we know that programmers could solve tasks with a higher success rate using Stacksplorer compared to plain Xcode. They were also able to find a hypothesis significantly faster for several tasks using Stacksplorer compared to using plain Xcode.

We use planned contrast ANOVA to compare all four conditions.

But of these two studies looked at the results independently. We will now try to combine the results of both studies to compare the 4 conditions. To do this, we will use a planned contrast analysis of variance (planned contrast ANOVA, see [Field, 2009, pp. 360–369]) to show the differences between the plugins and a default Xcode installation.

Since Xcode was the control condition in the previous study and the Call Hierarchy was the control condition in the new study of this thesis, we will put them together in the first contrast (C1) and compare them to both of our research prototypes, Stacksplorer and Blaze, combined. We do this because we believe the differences between our research prototypes are small, but the differences between our research prototypes compared to Xcode and to the Call Hierarchy are big. We also think that the differences between Xcode and the Call Hierarchy are not as big as the differences between the Call Hierarchy and our research prototypes.

We group Xcode and the Call Hierarchy together, as well as Blaze and Stacksplorer.

One could say that it is unfair to put Xcode and the Call Hierarchy in one group and compare them to our research prototype because "clearly" Xcode will be worse than the Call Hierarchy. Thus, a potentially significant difference between Xcode and the Call Hierarchy compared to our research prototypes would simply be due to Xcode not being up to the task, although the Call Hierarchy alone would be as good as our research prototypes. To check this possibility we do a second contrast (C2), comparing Xcode without plugins and the Call

We also test the differences in each of the groups.

Hierarchy to each other. After that, we will do a third contrast (C3), comparing Stacksplorer and Blaze to see whether there are differences between the two research prototypes.

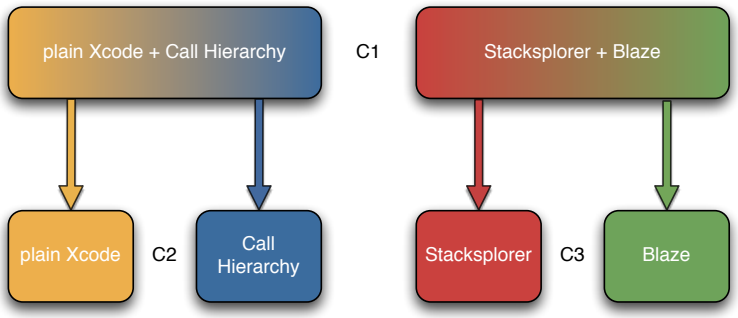


Figure 6.4: *A diagram showing the groups we plan to compare using planned contrast ANOVA to find difference between the plugins.*

Figure 6.4 gives an overview of the contrasts we plan to do. The selected contrasts are orthogonal. Using these contrasts we will explore similar hypotheses as in the individual studies:

H2 Programmers can solve tasks that require browsing and understanding previously unknown source code more quickly using an auto-updating call graph exploration tool that is visually interlinked with the source code than using a non-auto-updating non-interlinked call graph exploration tool or even no such tool.

H3 Programmers can identify side effects of changes made to unknown code more quickly using an auto-updating call graph exploration tool that is visually interlinked with the source code than using a non-auto-updating non-interlinked call graph exploration tool or even no such tool.

We do the first contrast (C1) to find the differences and the other two contrast (C2 and C3) to check whether our hypotheses are too broad and we have to narrow them down.

However, we can only do a planned contrast comparison with ANOVA, but the data for the success rate is binary data (either a participant got it right or wrong). Therefore, we cannot use

an ANOVA test to compare the conditions. Also, we expect the differences in success rate to be higher between Xcode and the Call Hierarchy plugin. For this reason we will do a Fisher's Exact test to test the following hypothesis:

We still use a Fisher's Exact test for the success rate.

H1 More programmers come to a correct hypothesis of how to solve a task that requires browsing and understanding previously unknown source code with a time-constraint using a call graph exploration tool than using plain Xcode.

Task Success Rates

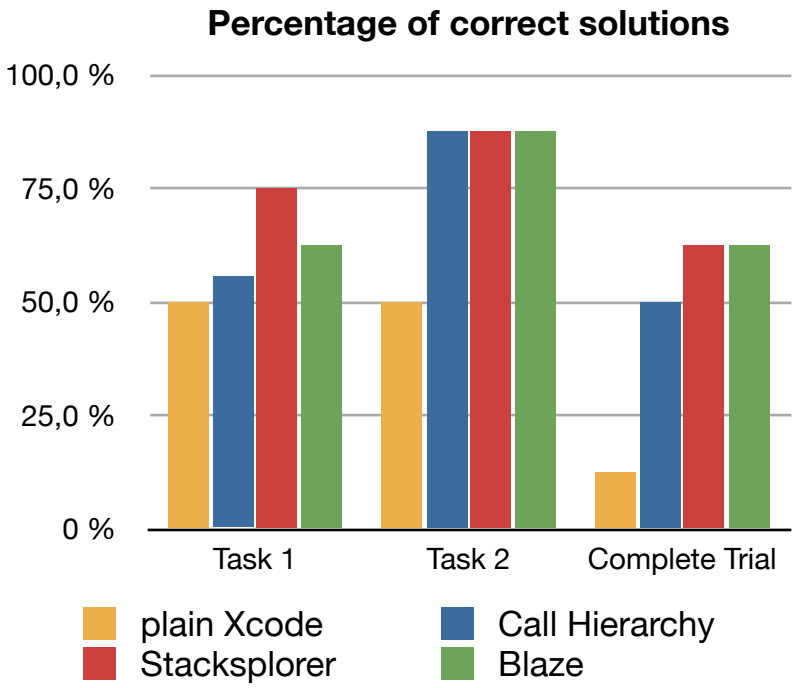


Figure 6.5: *The percentage of correct solutions grouped by the tasks and the plugin used. Note that for Task 1 of the Call Hierarchy we have 9 participants in total but 8 in all other cases.*

Figure 6.5 shows the percentage of participants who found a correct solution during the given time frame for each of the tasks and the complete trial. The complete trial was considered to be solved correctly if both tasks were solved correctly. All tasks were solved relatively more often with a call graph exploration tool than with a default Xcode installation. If there

Differences between the call graph exploration tools are small.

were differences in the success rate between the different tools, our research prototypes had a higher success rate than the Call Hierarchy with Stacksplorer having a higher success rate than Blaze in Task 1.

Task	p	odds ratio	confidence interval
Task 1	0.381	0.573	$[0, 2.968]$
Task 2	**0.047**	0.155	$[0, 0.973]$
Complete Trial	**0.030**	0.109	$[0, 0.827]$

Table 6.3: *The results of one-sided Fisher's Exact tests for the tasks comparing the success rate of Xcode participants to those using any of the three call graph exploration tools. The difference in success rate for Task 2 and the complete trial is significant.*

Call graph exploration tools increase the success rate compared to plain Xcode.

We performed one-sided Fisher's Exact tests comparing the success rate of Xcode to the success rate of the participants using any of the three plugins. The results of those tests can be seen in table 6.3. We found significant differences for Task 2 and the complete trial. However, we did not find significant differences for Task 1. To check whether there are significant differences between the individual conditions, we also performed two-sided Fisher's Exact tests with all four conditions (not grouped). But the results were not significant (Task 1: $p = 0.865$, Task 2: $p = 0.743$, Complete Trial: $p = 0.171$).

Task Completion Times

The division in "time to 'hypothesis" and "time to correct hypothesis" is the same as before, but this time we do this for all three contrasts. Before that, we discuss the data in general.

Figure 6.6 shows that the time to hypothesis for Blaze and Stacksplorer was almost equal or at least similar, with the difference between the two tools being less than a minute. But the difference to plain Xcode is a lot bigger and both tools shorten the time to hypothesis by several minutes on average. They also lead to a shorter average time to hypothesis compared to the Call Hierarchy with several minutes difference in Task 1 and the complete trial but only a slight difference in Task 2. This supports our decision to compare our research prototypes to Xcode and the Call Hierarchy combined.

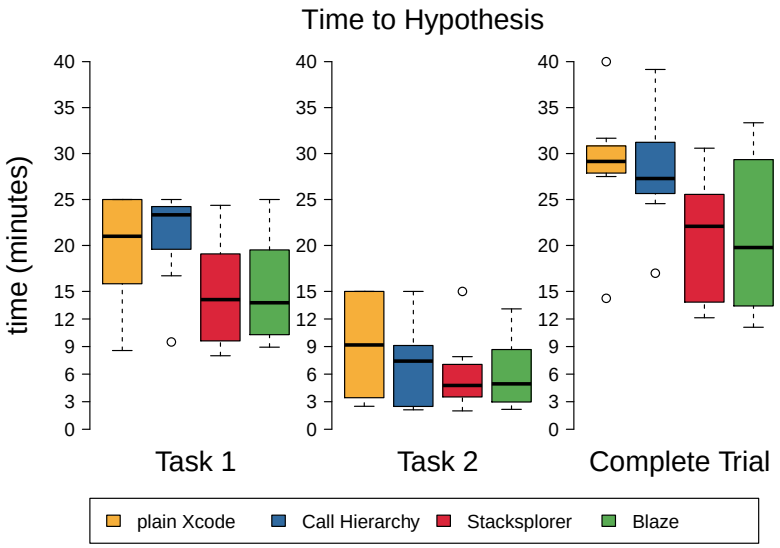


Figure 6.6: *The average time to hypotheses by task and condition in minutes. Participants using our research prototypes were several minutes faster in each task compared to plain Xcode. They were also faster using our research prototypes than using the Call Hierarchy. The difference between the two research prototypes is small.*

The case for the time to *correct* hypothesis is different, as shown in figure 6.6. In this case, Xcode. which had a long time to hypothesis, suddenly had a very short time to *correct* hypothesis in all three tasks. The differences between the other tools are similar although the difference between Blaze and Stacksplorer is bigger in this case and for Task 2 the Call Hierarchy is on average actually quicker than Blaze.

ANOVA assumptions Similar to a t-test one of the assumptions of ANOVA is that the underlying population is normally distributed. As before we assume that but also test it using Shapiro-Wilk tests. The results of those tests can be found in table 3. There are three cases in which a Shapiro-Wilk test found a significant difference to a normal distribution (Task 1 Xcode and Call Hierarchy and Task 2 Xcode). But a Kolmogorov-Smirnov test for these cases did not find a significant difference to the normal distribution and ANOVA is robust against non-normality[Field, 2009, pp. 359f], so we were convinced that we could use it nevertheless. Note that in the time to cor-

Again we assume the normality and test it with Shapiro-Wilk tests.

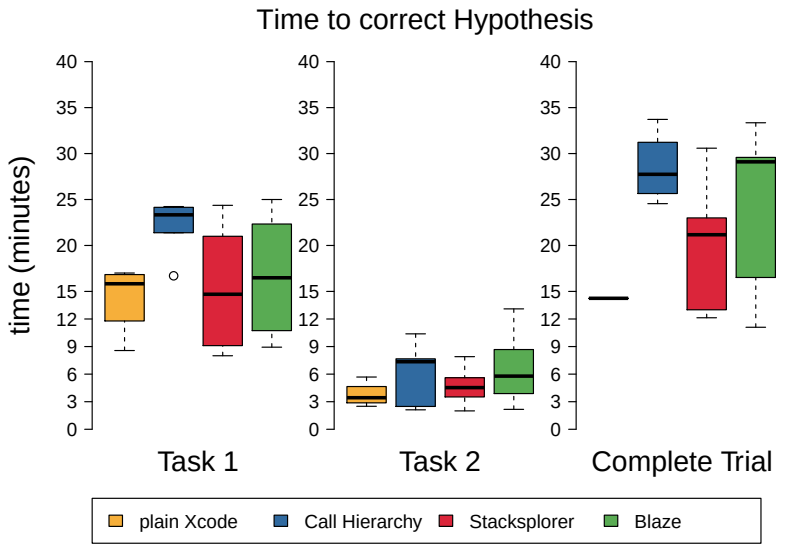


Figure 6.7: *The average time to a* correct *hypotheses by task and condition in minutes. Interestingly Xcode is faster than any of the plugins in all of the tasks. But it is also the one with the fewest participants who actually had a correct hypothesis so it might be that only the good programmers found a correct hypothesis.*

rect hypothesis for the complete trial the Xcode condition was not checked since it only includes one participant and it does not make sense to check a sample of size 1 for normality.

We also check the other ANOVA assumptions.

Another assumption is the homogeneity of variances of the samples. We tested this with a Bartlett's test (see table 4) and did not find a violation of this assumption in any of the tasks.

A third assumption is that the sample sizes of the different conditions are equal. This is the case for the time to hypothesis, but it is not always the case for the time to *correct* hypothesis. Especially the sample sizes of the correct answers for the complete trial differ a lot; thus, we will not conduct an ANOVA test in this case.

Comparing Our Research Prototypes to Xcode and the Call Hierarchy As explained before, our first contrast compares our control conditions (Xcode and Xcode with the Call

Hierarchy plugin) to our research prototypes. The results for this contrast can be found in table 6.4

Task	degree of freedom (contrast/total)	F	p
Time to hypothesis			
Task 1	1/29	7.12	**0.012**
Task 2	1/28	1.45	0.238
Complete Trial	1/28	8.29	**0.008**
Time to correct hypothesis			
Task 1	1/16	1.08	0.314
Task 2	1/21	0.25	0.619
Complete Trial	not compared due to sample size differences		

Table 6.4: *The results of a one-way planned contrast ANOVA comparing the time to (correct) hypothesis of Xcode and the Call Hierarchy to our research prototypes. The differences in the time to hypothesis for Task 1 and the complete trial are significant.*

We found significant differences in the time to hypothesis for task 1 and the complete trial with our research prototypes outperforming Xcode and the Call Hierarchy, but we did not find differences for Task 2. We did not find any significant difference in the time to *correct* hypothesis and thus can not support H2 nor H3.

Comparing the Control Conditions We wanted to compare the two control conditions to each other to make sure that the significant differences are not due to only one of the control groups. As table 5 shows there were no significant differences for any of the tasks, which indicates that the differences are not due to just one of the conditions. The only case close to being significant is the time to *correct* hypothesis for Task 1 with Xcode outperforming the Call Hierarchy.

We do not find differences between the control conditions.

Comparing Our Research Prototypes Last, we wanted to compare our two research prototypes to see whether any performed significantly better than the other. Again, we did not find any significant differences (see table 6) for any of the tasks.

We do not find differences between the research prototypes.

Post-Hoc Tests To see whether there were further differences between the tools, we also performed Welch's t-tests with Holm correction for each pair of conditions. To find as many differences as possible, we first ordered the conditions for each task by mean and then performed one-sided t-tests. This ordering might increase the experiment-wide Type I error rate; thus, we might find more significant differences than there actually are. But we did not find any significant difference for any pair of conditions for any task. The lowest p-value we got as a result was $p \approx 0.10$.

Discussion

We could confirm H1.

We could confirm H1 for the complete trial and Task 2, but were not able to find a difference for Task 1. We think this indicates that the tools do not increase the success rate for browsing focused tasks much. We could not find any differences between the tools so we conclude that any call graph exploration tool increases the success rate compared to an IDE without such a tool and that it does not matter what kind of tool is used.

Good performance of Xcode participants in the time to correct hypothesis is probably due to a filtering effect.

Regarding the time to (correct) hypothesis we assume that the good performance of the Xcode participants is due to a filtering process. Slower/less-skilled participants probably did not have a chance to find a correct solution with a default Xcode installation and thus only the skilled-programmers are included in the average time to correct hypothesis. The other tools might have allowed less-skilled participants to come to a correct hypothesis although they needed more time, thus increasing the average time.

We found significant differences for the time to hypothesis in Task 1 and the complete trial.

We could show significant differences in the time to hypothesis for the complete trial and Task 1. We were not able to show differences for Task 2 or the time to *correct* hypothesis. Thus, we have to reject H3, but we think we can confirm H2 since a shorter time to hypothesis will likely shorten the time to a solution, no matter whether this solution is correct. It would only be bad if our tools increased the probability for finding an incorrect hypothesis, which is not the case.

Our additional contrasts (C2 and C3) did not show any differences between our research prototypes or the control con-

ditions respectively. This means that our hypotheses were not too broad and did not combine unequal conditions. It also indicates that each of our research prototypes decreases the time to hypothesis compared to each of Xcode with and without the Call Hierarchy. However, we were not able to show these pairwise differences using post-hoc test, therefore more data is required to confirm or reject our hypotheses.

We think the better performance of our research prototypes is due to their auto-updating nature and their tight integration with the source code editor since this is one of the few commonalities between Blaze and Stacksplorer and a difference to the Call Hierarchy. However, the three tools are too different to be sure of the cause of these differences. Further research in this direction should certainly be interesting.

6.3 Qualitative Results

In this section we will look at the qualitative results of our study. We will first discuss the System Usability Scale rating we gathered during the study, then we will show the results to some further Likert-scale questions we asked the participants to answer, and in the end talk about some more general observations we made.

6.3.1 Postsession Questionnaire

A description of the postsession questionnaire was given in 6.1.4—"Postsession Questionnaire". Therefore, we will only discuss the results now.

System Usability Scale

The average SUS scores as well as the Usability and Learnability scores identified by Lewis and Sauro [2009] are shown in figure 6.8. All scores are quite close, with Stacksplorer having the best SUS score (median 87.5) followed by Blaze (median 85) and the Call Hierarchy (median 82.5). All scores would

All tools received excellent SUS scores.

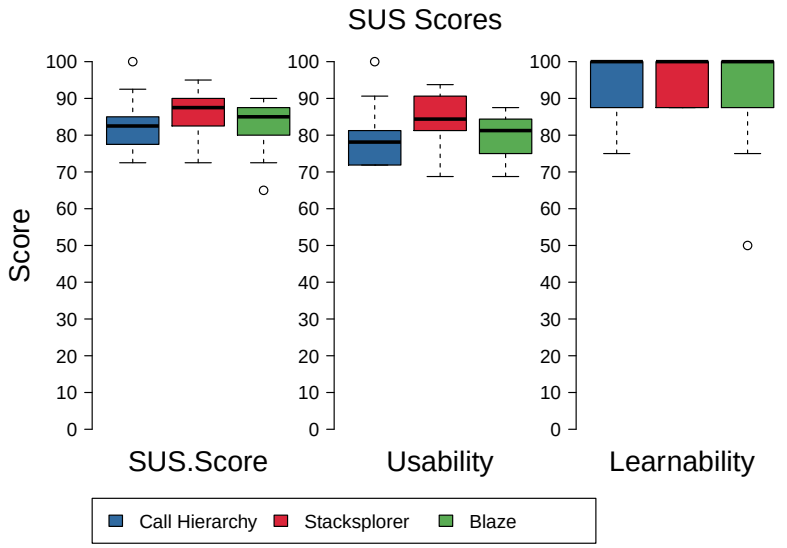


Figure 6.8: *The SUS Scores and the respective score for Learnability and Usability for Blaze, Stacksplorer and the Call Hierarchy. Higher is better.*

be translated as "excellent" by an interpretation of the SUS by Bangor et al. [2008].

Stacksplorer also has higher Usability (median 84.375) and Learnability (median 100) ratings, again followed by Blaze (Usability median: 81.25; Learnability median: 100) and the Call Hierarchy (Usability median: 78.125; Learnability median: 100).

The differences in SUS scores are not significant.

However, Kruskal-Wallis tests for the three scores did not reveal a significant difference in any of the scores (SUS: $\chi^2(2) = 2.40, p = 0.301$; Usability: $\chi^2(2) = 2.10, p = 0.349$; Learnability: $\chi^2(2) = 0.09, p = 0.955$).

Slower Call Hierarchy performance was not reflected in the SUS score.

Discussion Interestingly the significantly slower performance of participants using the Call Hierarchy plugin is not reflected in their subjective SUS rating. This is likely due to most participants not being used to any call graph exploration tool so their general excitement (resulting in the excellent scores) about having such a tool at hand hides the individual differences between the tools.

The achieved scores are quite good considering that they were rate research prototypes that still had a few bugs. However,

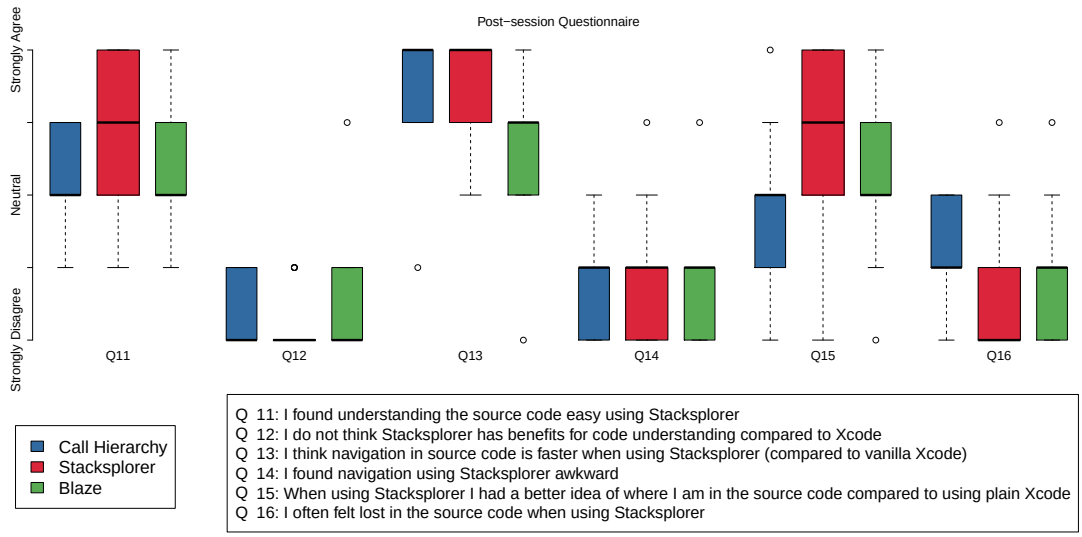


Figure 6.9: *The answers to our six additional questions for the plugins Blaze, Stacksplorer and the Call Hierarchy. The larger width of the Stacksplorer plot indicates the larger sample size. The Stacksplorer study had 17 participants filling out the questionnaire, our new study had 18, but those were split up between Blaze and the Call Hierarchy. The questions can be grouped into pairs, each concerned with one aspect of the respective plugin: Q11+Q12 (code understanding), Q13+Q14 (navigation) and Q15+Q16 (orientation in source code).*

we did not manage to design systems that were liked significantly more than the existing plugins. This did not happen so we can only say our "users generally liked the system[s equally well]". The Learnability rating was higher than the overall SUS score for each of the plugins, indicating that all of the plugins are easy to learn and understand.

6.3.2 Additional Post-Session Questionnaire Questions

The agreement to our six additional statements has to be evaluated independently from the SUS. Figure 6.9 shows box plots with the results. The 6 questions were designed to be grouped into 3 pairs each concerned with evaluating one aspect of the analyzed plugin.

Participants found source code understanding still difficult.

The first two questions were about source code understanding with the used plugin. Almost all participants agreed that the plugin they used had benefits for code understanding (Q12),

however, many still found code understanding difficult (Q11). Several participants actually said they thought understanding source code was *easier* but not *easy*.

Participants agree that the plugins improve the navigation.

The next two statements where concerned with the ease and speed of navigation using the plugin. More than half of participants strongly agreed that Stacksplorer as well as the Call Hierarchy made the navigation faster. For Blaze, agreement is less strong; still, most participants agree but not strongly. Most of the participants found the navigation not awkward (Q14).

The last two statements were concerned with the sense of location and orientation in the source code when using the plugin. This is where we got the least positive results. For Stacksplorer, the majority of participants agreed that they had a better idea of where they are in the source code when using the plugin (Q15). But quite a few were neutral or even disagreed. For Blaze the majority agreed or was neutral but again some disagreed. For the Call Hierarchy, more people disagreed than agreed to the statement with a third of them being neutral.

Participants do not know where they are, but do not feel lost either.

In our new study several users explained their reasons to disagree even without being asked to do so. Two of them independently came up with the same example: a satnav. They said, one reaches the destination safely and most of the time without problems but has no idea of where one has been during the journey or where that destination even is. Some other users gave similar reasons for their disagreement.

Although many users did not agree to Q15, the majority said they did not feel lost in the source code using the plugin (Q16), which fits the satnav example of not knowing where one is but not feeling lost either. Again, Stacksplorer gets the best results followed by Blaze and the Call Hierarchy.

A big flaw in the Blaze implementation was that it did not highlight where the succeeding method is called in the implementation.

Discussion We think less enthusiastic agreement to the question whether Blaze makes navigation faster, is due to a flaw in the Blaze implementation, which we only noticed after the user tests began: It does show which method calls another method and it allows users to jump to the implementation of a displayed method, but there is no way to find out *where* the succeeding method is called in the currently displayed implementation. Both the Call Hierarchy and Stacksplorer have this functionality and we noticed that participants were using it

during the study and missed it in Blaze (see 6.3.3—"Observations and User Comments").

Regarding the orientation of the programmer in the source code, there seems to be room for improvement for all the plugins. We think the differences between the plugins are due to two reasons. First, Stacksplorer lets users only do one step in the call graph at a time, the Call Hierarchy and Blaze show several steps into the call graph and let users jump directly between methods that are several calls apart from each other. Thus, it is easier using one of the latter plugins to loose ones starting point or forget which way one took to get where one is now.

We think the reason that Blaze gets a slightly better rating than the Call Hierarchy is that Blaze highlights the relationship of the currently displayed code to the displayed part of the call graph better. The Call Hierarchy only shows which method was clicked on last, but does not change the highlight if the user navigates to another method without using the plugin. Blaze highlights the connection of the currently displayed method to the stack with a big overlay, which updates automatically when the user navigates to another method, no matter how he or she got there. This way, Blaze shows at least the current position in the selected subset of the call graph as accurate as possible.

Blaze is better integrated with the editor then the Call Hierarchy.

6.3.3 Observations and User Comments

In general users, liked the plugin (no matter which one of them) as we have already seen from the SUS results and wondered why Xcode does not provide such a functionality by default. But they also had some problems with each of them and some critique.

Call Hierarchy One of the biggest complaints about the Call Hierarchy was that it was too difficult to invoke. Three users wanted it to update automatically when they switched to another method in the editor. We also observed that the non-updating Call Hierarchy was a problem in some cases. For example one of our participants looked at the Call Hierarchy after they found the correct starting method. But they

Users want the Call Hierarchy to update automatically.

did not remember to invoke the Call Hierarchy again for this method, so the call tree they were looking at belonged to a completely different method. Since they did not realize that the information was out of date, they interpreted the call tree that was displayed as being the call tree for the method in the editor, which was not the case. Another participant said they would like to have a highlight in the Call Hierarchy to show which of the methods in the Call Hierarchy is currently shown in the editor.

Using the context-menu to invoke the Call Hierarchy is less than ideal.

Three users also complained that invoking the Call Hierarchy using the context menu was "awkward" and "annoying" and they thought the functionality was "hidden" in the context menu. We also observed another user choosing a neighboring option accidentally and several users seemed to have to search a bit for the right option after invoking the context menu.

The tree view was often confusing.

Several participants also did not like the tree view in itself, especially the version of the caller view. Two participants said that it was confusing that if method A calls method B, A would appear as a child of B in the caller view, which is exactly the opposite of their mental model (B follows A). Two others found it difficult to discern which node is on which level.

Caller and callee view were difficult to distinguish.

Another point of criticism was that it was difficult to distinguish the callee and caller view. Since there is no difference in how they are displayed, the only way to know which one is displayed is the little icon at the bottom. But about these icons, three of our participants said they found them confusing, "non-intuitive", "meaningless" and "too small".

Users search for the location of a call in the code, not just a method implementation.

Blaze We think the biggest problem of Blaze was the missing highlight of a succeeding method in the currently displayed implementation. We noticed that many users clicked on a method to navigate to its implementation and then looked for the place in code where the method that is displayed as the next method (and which is called from the current method) was called.

This happened when navigating downstream, but more often when navigating upstream to find out under which conditions the method they came from is called. Also three users said in the post-session discussion or during the study that they would like to have such a highlight functionality.

Three users also complained about the overlay showing which method in the path is currently displayed in the editor. They generally liked the idea, but often they found it too strong and intrusive. Stacksplorer has the option to activate and deactivate overlays, but in Stacksplorer's case there are a lot more overlays which are probably more disturbing. We thought that only one overlay that does not obscure any of the current method's implementation would not disturb the user very much. However, we seem to have misjudged this and underestimated how often users want to read methods that are implemented in the code directly before or after the currently displayed method.

The overlay is too intrusive.

Another request, that came from three different users independently, was functionality to filter the displayed path. All of them wanted to be able to hide the calls to framework methods to reduce the number of alternatives displayed. Some would like to filter the possible methods by method and class names or even their belonging to a specific framework.

Filtering the path is another popular request.

Other requests include keyboard navigation and using Blaze to find a path between two methods. Another one was to display previously visited methods in the outgoing or incoming path with a higher priority. This way, if a user first visits a method A and then navigates to B without using Blaze and thus B is the focus method having A in one of its incoming paths the path containing A should be displayed instead of the default incoming path for B. This way it would be easier for users to find relationships between methods they have visited.

Comparing Blaze to the Call Hierarchy user's comments we notice that several of the complaints there were things we tried to fix with Blaze. For example, Blaze is auto-updating its display to correspond to the users selection in the editor by default. But if an important starting point was found, which should not be lost, it can can locked; thus hopefully invalidating the fears of those, who were afraid auto-updating could make them loose interesting call trees.

Blaze is auto-updating, which Call Hierarchy users wished for.

Another thing we fixed is the confusing tree view for the Caller View in which calling methods were displayed as children of the called methods. In our case, a method that is calling another method will always be displayed above this method. And we hope we also eliminated the confusion whether one is

Blaze is less confusing than the Call Hierarchy.

currently looking at the Caller or the Callee view. We made the focus method clearly identifiable; callers are always displayed above the focus method and callees below the focus method. Also, this position will not change without user interaction, so if users did not move the focus method, they can always look at the lower right of the screen to find the outgoing path and at the upper right screen to find the incoming path; thus using their spatial memory.

6.4 Improvements to Blaze

Add a highlight and make the overlay less intrusive.

Following the results from our evaluation we plan to make a few changes to Blaze. First, we want to add the highlighting of the call to the next method in the implementation of the currently viewed method (as described above). Second, we want to make the overlay less intrusive, possibly user-hideable.

Help users recognize the scroll position and enable filtering of the possible paths.

We also want to implement the zigzag-view mentioned in the design to notify users that some methods are hidden behind the focus method. We saw some users not recognizing that methods are hidden, despite the existence of the scroll bar on the side. Something else that was already planned in the design is the search field. We realized during the evaluation that users would like to have it to filter the possible paths. Some bigger possible changes are discussed in 7.2.1—"Blaze Improvements".

7 Summary and Future Work

"The future cannot be predicted, but futures can be invented."

—Dennis Gabor

The work in this thesis complements other previous work in creating software—and specifically call graph—exploration tools (see chapter 3—"Related work"). We will summarize our work and point out interesting future work on our prototype and in the context of call graph exploration in general.

7.1 Summary and Contributions

We presented Blaze, a novel way to visualize and explore the call graph context of a method and the call graph of a software project in general using a one-path-at-at-time visualization. Blaze displays a combination lock interface on the right hand side of the code editor showing an incoming and an outgoing path to the method currently selected in the editor, the focus method. It thus takes an orthogonal approach compared to Stacksplorer and displays only a subset of the information usually displayed by call graph exploration tools in today's IDEs.

Blaze allows navigating through the call graph by displaying possible paths.

Blaze supports normal programmer navigation behavior by first providing context to the currently selected method while the programmer browses the source code searching for an anchor point. When an anchor point is found, Blaze is able to lock the focus method; thus making the anchoring explicit. It then allows for the exploration of the surrounding call graph from this method and provides easy backtracking when a chosen path does not lead to the desired information.

Blaze supports established programmer navigation models.

Blaze was implemented as an Xcode plugin and can index arbitrary Objective-C Xcode projects in theory[1]. It can then be used as a navigation and exploration help. Blaze's user interface has been developed over several iterations and achieved excellent SUS Scores (mean: 81.9).

Call graph exploration tools increase the success rate, but only our research prototypes decrease the task completion time.

We compared Blaze to another research prototype, Stacksplorer, to the Eclipse Call Hierarchy as an example of a commonly used call graph exploration tool, and an IDE without a call graph exploration tool, represented by Xcode. We found that a call graph exploration tool increases the success rate. Further, we found that using one of the research prototypes significantly decreases the time to a hypothesis when working on maintenance tasks compared to Xcode with or without the support of the Call Hierarchy. We think this is due to the tight coupling to the source editor of both research prototypes and their auto-updating nature.

Parts of the results of this thesis, especially the ones presented in chapter 6—"Evaluation", were used in a paper submitted to the ACM SIGCHI Conference on Human Factors in Computing Systems 2012.

7.2 Future Work

Besides the smaller and bigger flaws of Blaze we recognized during the user study (see 6.4—"Improvements to Blaze"), we found some more general improvement possibilities to Blaze which also lead to some interesting further research.

7.2.1 Blaze Improvements

First, we will describe some more general possible improvements to Blaze.

[1] Due to the research prototype nature of Blaze we did not test it with a lot of different projects so there may still be bugs that prevent it from working in some constellations.

More Relationships

Although navigation along the call graph is an important part of programmer navigation, caller/callee-relationships are certainly not the only kind of relationship programmers navigate along and use for finding the required information or locations. Thus, it would be great if Blaze could be extended to support a wider range of relationships. Schäfer et al. [2006] even argue that software exploration tools should be end-user extendable with regard to what kind of relationships they support. We are doubtful as to how well Blaze's simple way of visualizing paths scales to more or even arbitrary relationships, but it would certainly be interesting to explore these possibilities.

Integrating more relationships might be useful but is difficult.

More Accurate Call Graph Parsing

Currently, the parser of Blaze, which creates the call graph, is not particularly smart. For example, it will not recognize that a call to a method x of class A could also be a call to the method x' of class B which is a subclass of A and overrides x with x'. Also, we often have paths that end in some framework method or start at some method that is called by the framework since the parser is just using static analysis to create the call graph.

The Blaze parser is not particularly smart.

This hides potentially interesting relationships, for example, if a controller object tells a table view to update its information (with the table view being implemented by the framework) and the table view then calls a method on a model object to get the updated information. Blaze would currently display two independent paths in this case and would not find a connection although there clearly is one.

Several interesting paths will not be found.

For this reason, we would like to use information from tracing application runs to integrate paths into the call graph that can not be found statically. However, we want to combine the static and the runtime analysis since the runtime analysis alone is not complete either, since certain executable paths might not occur in a specific run of the application [LaToza and Myers, 2011].

Integrating information from runtime traces could be a solution.

Better Initial Path Selection

Often, the initial path selected by Blaze is not very helpful.

We think Blaze could be improved a lot by having a heuristic to select paths for the user if they did not select one yet. At the moment, their is no such heuristic and if the user looks at a new[2] method it just displays the first path in the list. This often leads to the outgoing paths ending quickly in a framework method although there may be more interesting paths to the developer. This, in turn limits the usefulness of the information displayed by Blaze when browsing source code and looking for interesting methods.

Algorithms of recommender tools might be integrated.

There has already been a lot of research in different kinds of recommender systems [Singer et al., 2005, DeLine et al., 2005, Čubranić and Murphy, 2003]. However, these are usually not specialized on recommending paths. Nevertheless some of this research might be used to find a good recommendation of paths for Blaze.

7.2.2 Open Research Questions

We also identified two areas in which further research might be interesting and helpful to call graph exploration tools.

Degree of Interest of Paths Based on the Call Graph Structure

Recommender systems use navigation history as metric.

As mentioned above, Blaze could be improved with a better initial path selection. We already mentioned recommender systems as a way to recommend a path. But the recommender systems usually calculate the degree of interest of an artifact based on previous programmer navigation or similar wear-based algorithms.

Maybe there are inherent properties of a subgraph that make it more interesting.

We think it might be interesting to see whether there are properties of a call graph that allow predictions about the degree of interest of a specific node to a programmer. For example: Are nodes with a larger number of outgoing edges more interesting to a programmer than those with a low number? What

[2] if the user looks at a previously visited method Blaze will display the last

about incoming edges? Are nodes that are part of a cycle in the call graph more or less interesting to the programmer? Is the depth of the subtree starting in a node a good prediction of the importance of this node to the programmer?

We could not find much research that looks into this relationship of the structure of the call graph to the relevance for understanding the program. Some researchers used Strahler numbers[3] as a relevance rating for nodes in directed acyclic graphs. Auber [2002] used it to prioritize graph drawing in huge graphs and thus provided a preview of the final graph to the user while the complete graph was being drawn. Herman et al. [1998] used it to highlight branches with a high Strahler number in a tree visualization to give users hints which parts of the tree are worth exploring. However, neither of them evaluated their approach to see whether the preview or highlighting actually showed the parts of the graph users were interested in.

There are some graph metrics, but they have not been evaluated, yet.

Are Auto-Updating Tightly Integrated Exploration Tools Really Better?

In our study we found that the two tools which were tightly integrated with the source code editor and auto-updated to show information relevant to the currently selected method performed better than Xcode with or without the Call Hierarchy, which is not that tightly integrated and does not update automatically. We think this difference is due to the tight integration and the auto-updating but obviously there are a lot of other differences between the two research prototypes and the Call Hierarchy, thus we can not be sure.

Our results indicate an advantage of auto-updating, tighly-integrated software exploration tools.

On the one hand one could think that tools with these features are obviously better than tools which are less-integrated and require the user to actively interact with it to display relevant information. On the other hand none of the call graph exploration tools we found in current IDEs (See section 3.2.1—"Current IDEs") has these features. Also, none of the research prototypes presented in section 3.2.2—"Research" has these

But many other tools are not auto-updating. Why?

path the user saw which contained this method

[3] see [Auber, 2002] for an explanation

features[4]. Therefore, we think further research into this question should be interesting to either confirm our hypothesis and include these features in future exploration tools or explain why the existing tools are better without these features.

[4]We are not sure whether CallStax (see 3.2.2—"CallStax") updates when the user clicks somewhere in the code or only when he specifically tells CallStax to update. We could not find this information in the description of the referenced paper.

Bibliography

Keith Andrews and Janka Kasanicka. A Comparative Study of Four Hierarchy Browsers using the Hierarchical Visualisation Testing Environment (HVTE). In *2007 11th International Conference Information Visualization (IV '07)*, pages 81–86. IEEE, 2007.

David Auber. Using Strahler numbers for real time visual exploration of huge graphs. In *International Conference on Computer Vision and Graphics*, pages 56–69, September 2002.

Aaron Bangor, Philip T Kortum, and James T Miller. An Empirical Evaluation of the System Usability Scale. *International Journal of Human-Computer Interaction*, 24(6): 574–594, July 2008.

Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the 28th international conference on Human factors in computing systems*, pages 2503–2512, New York, NY, USA, 2010. ACM.

J. Brooke. SUS: A quick and dirty usability scale. In P. W. Jordan, B. Weerdmeester, A. Thomas, and I. L. Mclelland, editors, *Usability evaluation in industry*. Taylor and Francis, London, 1996.

Davor Čubranić and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 408–418, 2003.

Robert DeLine, Mary Czerwinski, and George Robertson. Easing program comprehension by sharing navigation data. *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, pages 241–248, 2005.

Katalin Erdös and Harry M. Sneed. Partial comprehension of complex programs (enough to perform maintenance). In *6th International Workshop on Program Comprehension. IWPC'98*, pages 98–105. SES Software Eng. Service, Budapest, IEEE Comput. Soc, June 1998.

Alexander A. Evstiougov-Babaev. Call graph and control flow graph visualization

for developers of embedded applications. In *Graph Drawing*, pages 337–346. AbsInt Angew Informat GmbH, Saarbrucken, Germany, 2002.

Andy Field. *Discovering Statistics Using SPSS*. ISM Introducing Statistical Methods. Sage Publications Ltd, London, 3rd edition, 2009.

G. W. Furnas. Generalized fisheye views. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 16–23, New York, NY, USA, 1986. ACM.

Martin Graham and Jessie Kennedy. A Survey of Multiple Tree Visualisation. *Information Visualization*, 9(4):235–252, December 2010.

Wilhelmiina Hämäläinen. Writing Scientific English, September 2006. URL `http://www.cs.joensuu.fi/pages/whamalai/sciwri/sciwri.pdf`.

I Herman, G Melancon, and M S Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

Ivan Herman, Maylis Delest, and Guy Melancon. Tree visualisation and navigation clues for information visualisation. *Computer Graphics Forum*, 17(2):153–165, June 1998.

Mao Lin Huang, Peter Eades, and Junhu Wang. Online Animated Graph Drawing using a Modified Spring Algorithm. *Journal of Visual Languages and Computing*, 9: 17–28, 1998.

Mikkel Rønne Jakobsen and Kasper Hornbæk. Fisheyes in the field: using method triangulation to study the adoption and use of a source code visualization. In *Proceedings of the 27th international conference on Human factors in computing systems*, pages 1579–1588. ACM, April 2009.

Jeff Johnson. Our Vision is Optimized to See Structure. In *Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules*, pages 11–24. Morgan Kaufmann, Burlington (USA), 2010.

Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan Borchers. Stacksplorer: Call Graph Navigation Helps Increasing Code Maintenance Efficiency. In *Proceedings of the ACM UIST 2011 Symposium on User Interface Software and Technology*, 2011.

Andrew J Ko, Htet Htet Aung, and Brad A Myers. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 126–135, 2005.

Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet 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, December 2006.

Jan-Peter Krämer. Stacksplorer Understanding Dynamic Program Behavior. Master's thesis, RWTH Aachen University, January 2011.

Thomas D. LaToza and Brad A Myers. *Searching across paths*. ACM, May 2010a.

Thomas D. LaToza and Brad A Myers. *Developers ask reachability questions*. ACM, May 2010b.

Thomas D. LaToza and Brad A Myers. Visualizing Call Graphs. *Visual Languages and Human-Centric Computing (VL/HCC)*, 2011.

Joseph Lawrance, Rachel Bellamy, Margaret Burnett, and Kyle Rector. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*. ACM Request Permissions, April 2008.

Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D. Fleming. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *Software Engineering, IEEE Transactions on*, PP(99):1, 2010.

James Lewis and Jeff Sauro. The Factor Structure of the System Usability Scale. In Masaaki Kurosu, editor, *Lecture Notes in Computer Science*, pages 94–103. Springer Berlin / Heidelberg, 2009.

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

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): 889–903, December 2004.

Thorsten Schäfer, Michael Eichberg, Michael Haupt, and Mira Mezini. The SEXTANT Software Exploration Tool. *IEEE Transactions on Software Engineering*, 32(9):753–768, September 2006.

Kaitlin Duck Sherwood. Path exploration during code navigation. Master's thesis, The University of British Columbia (Vancouver), August 2008.

J Singer, R Elves, and M.-A Storey. NavTracks: supporting navigation in software maintenance. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 325–334, 2005.

Ian Sommerville. *Software Engineering*. Pearson Education Limited, Essex, England, 8 edition, 2007.

Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software visualization. In *Program Comprehension, 1997. IWPC '97. Proceedings., Fifth Iternational Workshop on*, pages 17–28. IEEE, 1997.

Peter Young and Malcolm Munro. A New View of Call Graphs for Visualising Code Structures. Technical report, January 1997.

Other Pictures and Diagrams

In here we present additional pictures, diagrams and figures that might be interesting or helpful to the engaged reader.

1 Related Work

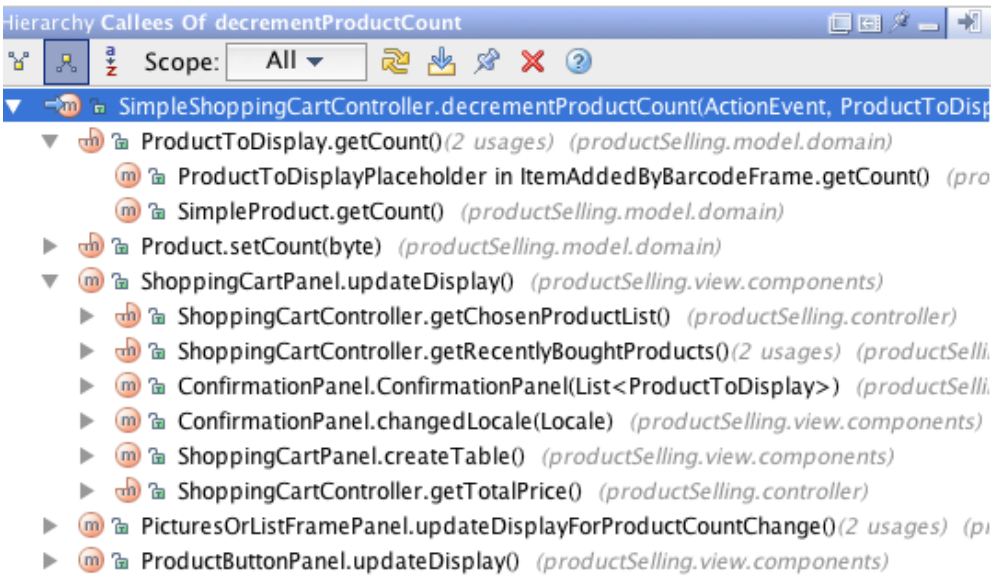


Figure 1: *Call Hierarchy view of IntelliJ: Only one direction (callees or callers) is viewable at a time.*

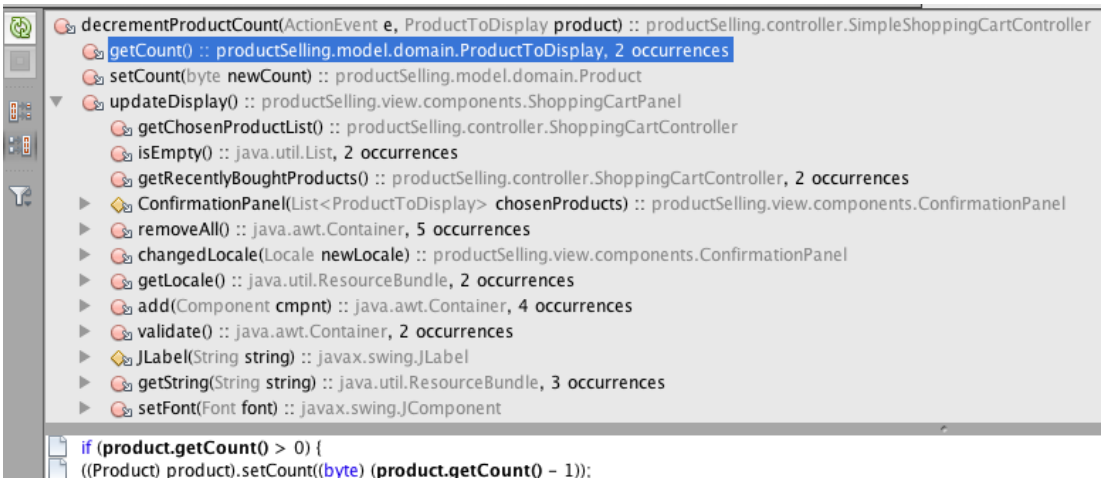


Figure 2: *Call Hierarchy view of NetBeans: Only one direction (callees or callers) is viewable at a time.*

2 Design

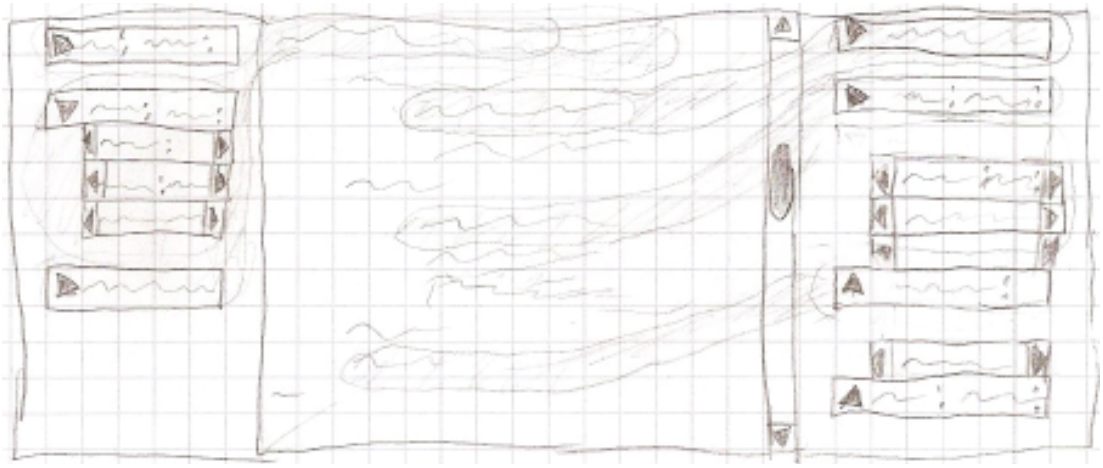


Figure 3: *A first concept of how to integrate the information that Blaze provides into the existing Stacksplorer implementation. We modified the Stacksplorer view to add little arrows to the method view cells. Those could be used to expand an outgoing/incoming path for this method that could then be adjusted with a combination lock view.*

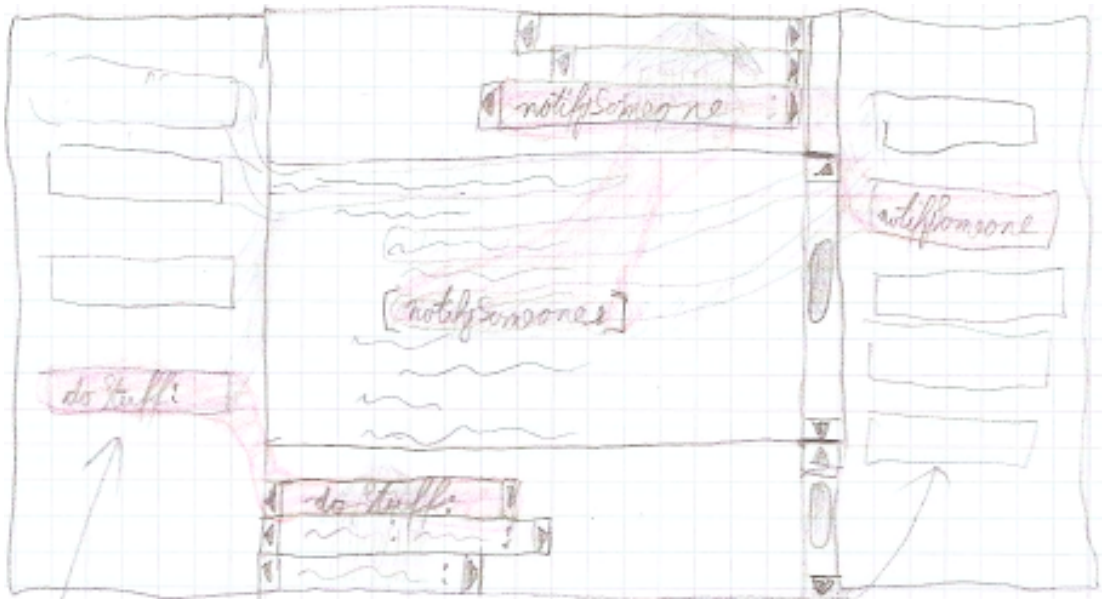


Figure 4: *A second concept of how to integrate the information that Blaze provides into the existing Stacksplorer implementation. We added a view below and above the editor to show an outgoing and an incoming path, again using combination lock views for modifying those. Clicking on the method views on the side would chose the corresponding path for this method. The method whose path is currently selected is shown by an overlay connecting the method on the side with the path above/below the editor.*

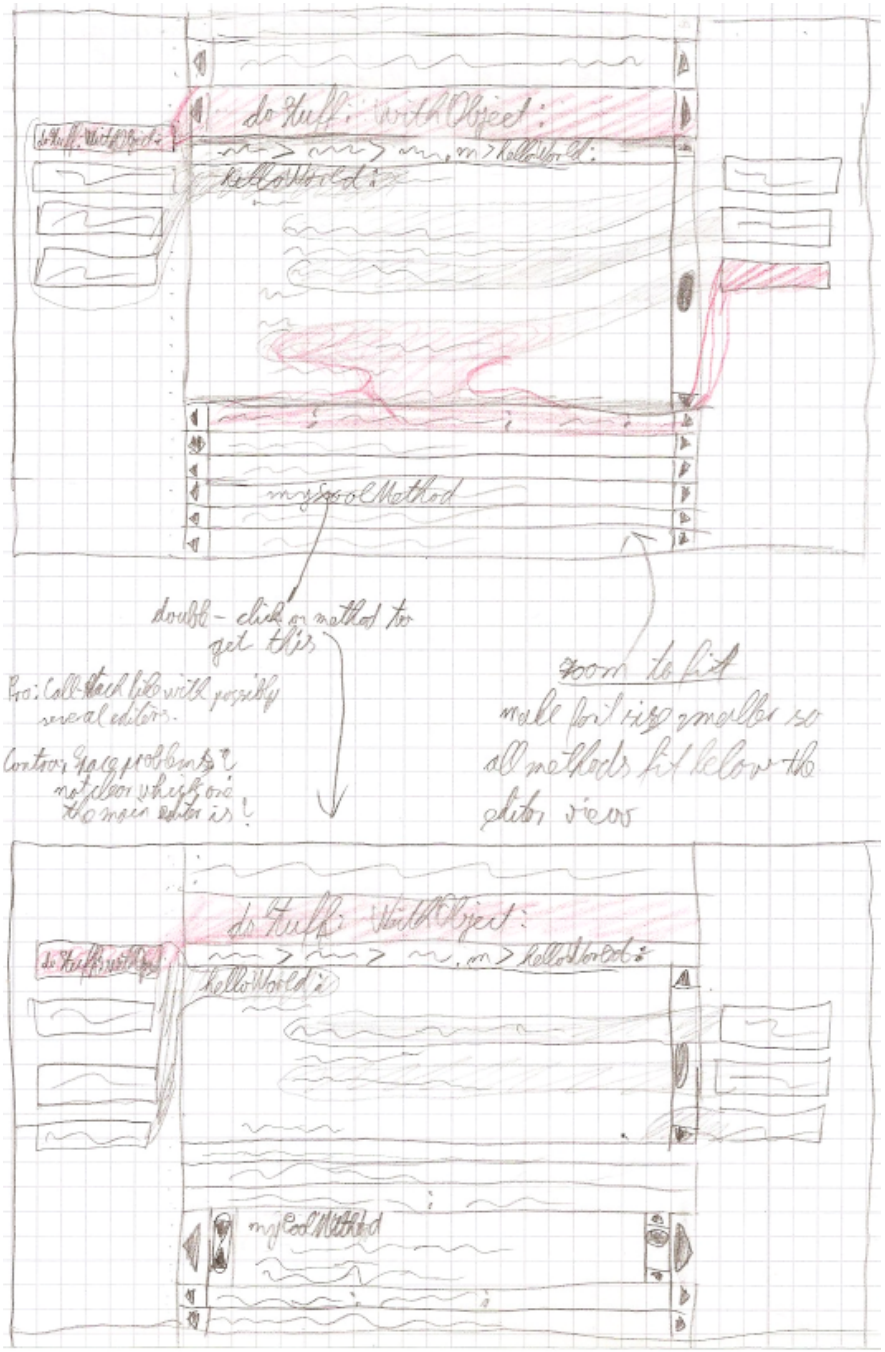


Figure 5: *A third concept of how to integrate the information that Blaze provides into the existing Stacksplorer implementation. The difference to the second one is that the user can click on methods in the combination lock view to expand them into an editor view that shows the implementation of the method.*

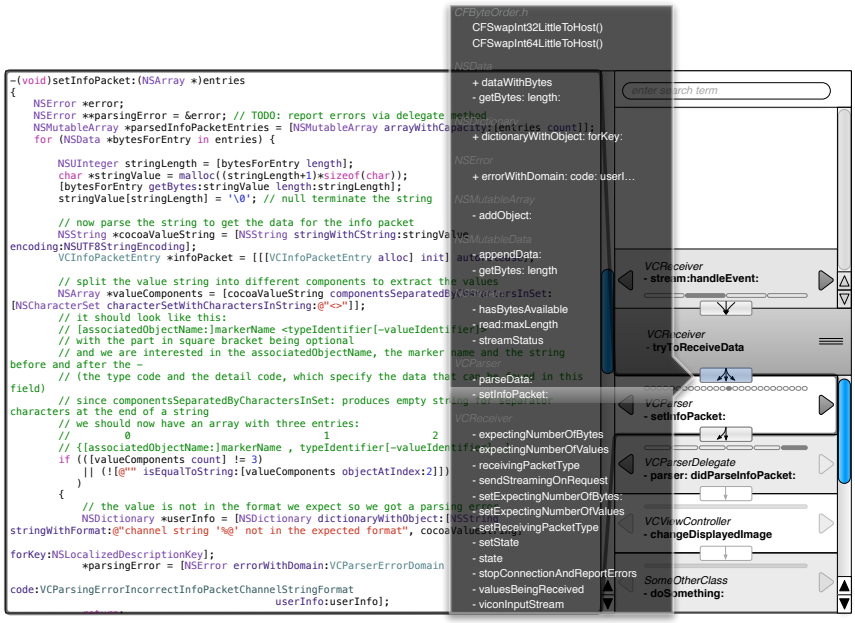


Figure 6: *Another picture of the new prototype that was designed after the first few user interviews and after some discussion with fellow researchers. In addition to the changes explained for figure 4.5, one can see the new popover used to show all the alternative methods. Also, the currently shown method in the editor is not the focus method in this case, which is shown by the changed overlay.*

Figure 7: *A screenshot of one of our user study participants using our implementation of the Eclipse Call Hierarchy. They have invoked the Call Hierarchy on the method* `parseFormat:forField:linkedFile:ofItem:suggestion` *and are now exploring the caller view to look for callers. In the screenshot they just clicked on* `updateFormatPresetUI` *which calls the parent node* `updateFormatPreviewUI`*. Thus, the call to* `updateFormatPreviewUI` *is highlighted in the editor.*

User Study Results

3 New Study

Task	p	odds ratio	confidence interval
Task 1	1	1.311	$[0.132, 14.3]$
Task 2	1	2.214	$[0.093, 156.8]$
Complete Trial	1	1.614	$[0.156, 18.7]$

Table 1: *The results of two-sided Fisher's Exact Tests for the tasks comparing the success rate of Call Hierarchy participants to Blaze participants. None of the results is significant.*

Task	**Condition**	**W**	**p**
Time to Hypothesis			
Task 1	Call Hierarchy	0.802	0.021
Task 1	Blaze	0.896	0.268
Task 2	Call Hierarchy	0.885	0.211
Task 2	Blaze	0.943	0.638
Complete Trial	Call Hierarchy	0.919	0.425
Complete Trial	Blaze	0.954	0.748
Time to Correct Hypothesis			
Task 1	Call Hierarchy	0.808	0.094
Task 1	Blaze	0.945	0.699
Task 2	Call Hierarchy	0.893	0.336
Task 2	Blaze	0.889	0.271
Complete Trial	Call Hierarchy	0.923	0.556
Complete Trial	Blaze	0.948	0.724

Table 2: *The results of Shapiro-Wilk tests for the different samples in the study.*

4 Comparing the Results from the Old and the New Study

Task	Condition	W	p
Time to hypothesis			
Task 1	plain Xcode	0.820	0.047
Task 1	Call Hierarchy	0.802	0.022
Task 1	Stacksplorer	0.935	0.566
Task 1	Blaze	0.882	0.196
Task 2	plain Xcode	0.789	0.022
Task 2	Call Hierarchy	0.896	0.268
Task 2	Stacksplorer	0.827	0.055
Task 2	Blaze	0.872	0.158
Complete Trial	plain Xcode	0.853	0.101
Complete Trial	Call Hierarchy	0.949	0.690
Complete Trial	Stacksplorer	0.930	0.512
Complete Trial	Blaze	0.899	0.284
Time to correct hypothesis			
Task 1	plain Xcode	0.797	0.098
Task 1	Call Hierarchy	0.808	0.094
Task 1	Stacksplorer	0.932	0.596
Task 1	Blaze	0.924	0.556
Task 2	plain Xcode	0.908	0.474
Task 2	Call Hierarchy	0.850	0.122
Task 2	Stacksplorer	0.985	0.981
Task 2	Blaze	0.902	0.344
Complete Trial	Call Hierarchy	0.957	0.762
Complete Trial	Stacksplorer	0.923	0.552
Complete Trial	Blaze	0.877	0.297

Table 3: *The results of Shapiro-Wilk tests for the different conditions.*

Task	df	χ^2	p
Time to hypothesis			
Task 1	3	0.31	0.956
Task 2	3	1.24	0.743
Complete Trial	3	0.72	0.868
Time to correct hypothesis			
Task 1	3	2.93	0.402
Task 2	3	4.96	0.175
Complete Trial	2	2.03	0.363

Table 4: *The results of Bartlett's tests to check the homogeneity of variances in the conditions of each task.*

Task	degree of freedom (contrast/total)	F	p
Time to hypothesis			
Task 1	1/29	0.22	0.642
Task 2	1/28	0.89	0.354
Complete Trial	1/28	0.04	0.852
Time to correct hypothesis			
Task 1	1/16	4.15	0.059
Task 2	1/21	1.09	0.309
Complete Trial	not compared due to sample size differences		

Table 5: *The results of a one-way planned contrast ANOVA comparing the time to (correct) hypothesis of Xcode to those of the Call Hierarchy. Theses tests did not reveal a significant difference.*

Task	degree of freedom (contrast/total)	F	p
Time to hypothesis			
Task 1	1/29	0.02	0.895
Task 2	1/28	< 0.01	0.968
Complete Trial	1/28	0.02	0.896
Time to correct hypothesis			
Task 1	1/16	0.17	0.688
Task 2	1/21	1.42	0.246
Complete Trial	not compared due to sample size differences		

Table 6: *The results of a one-way planned contrast ANOVA comparing the time to (correct) hypothesis of our two research prototypes to each other. Theses tests did not reveal a significant difference.*

User Study Material

5 User Interviews

These are the questions used during the first set of user interviews:

1. Was stellt die Ansicht an der Seite dar?
2. In welcher Beziehung stehen "-stream:handleEvent:" und "tryToReceiveData"?
3. In welcher Beziehung stehen "tryToReceiveData:" und "parser:didParseInfoPacket:"?
4. Wozu sind die Pfeile an der Seite der Methoden da?
5. Welche Methode wird angezeigt, wenn man auf den linken Pfeil bei "setInfoPacket:" klickt? In welcher Beziehung steht diese zu "setInfoPacket" und wovon hängt ab ,welche Methode angezeigt wird, wenn man auf den Pfeil klickt?
6. Warum ist der rechte Pfeil bei "parser:didParserInfoPacket:" ausgegraut?
7. Warum ist "tryToReceiveData" grau und hat keine Pfeile?
8. Was bedeuten die kleinen Zahlen in den Pfeilen?
9. Du möchtest nun gerne zu der Stelle im Code springen, an der eine bestimmte Methode implementiert wird. Was würdest du tun?
10. Du würdest nun gerne eine Übersicht über alle Alternativen zu einer Methode bekommen. Was würdest du tun? Was würdest du erwarten, dass dann passiert?
11. Was nimmst du an, dass passiert, wenn du auf eine der Methoden in der Seitenansicht klickst?
12. Was nimmst du an, dass passiert, wenn du auf den Bereich zwischen den Methoden (den Balken selbst oder daneben) klickst?
13. Wenn du auf den Balken zwischen den Methoden klickst, erscheint ein Menü mit der Übersicht der alternativen Methoden. Wonach sollte das Menü zur Anzeige

der alternativen Methoden sortiert sein? Aufrufreihenfolge? Alphabet? Klassen und dann Alphabet? Etwas anderes?

And for the recursion visualization:

1. Was bedeutet das Symbol unten an der 777?
2. Was bedeutet das Symbol unten an der 222?
3. Probiere nun selbst etwas damit herum. Nun stelle bitte den Stack 999, 111, 333F, 666, 111, 555, yyy, 555, yyy, 555, xxx ein.
4. Nun stelle bitte den Stack 999, 111, 333F, 666, 111, 222, 444, 000, 222, 444, 000, 222 ein.
5. Nun stelle bitte den Stack 999, 111, 333, 666, 111, 333F, 666, 777 ein.
6. Nun sieh dir bitte den unterliegenden Call-Graphen an und probiere noch einmal ein wenig mit dem Stack herum. Erscheint dir die Darstellung und das Verhalten des Stacks sinnvoll und konsistent?

Questions for new prototype These questions were used for the changed prototype:

1. Was stellt die Ansicht an der Seite dar?
2. In welcher Beziehung stehen "-stream:handleEvent:" und "tryToReceiveData"?
3. In welcher Beziehung stehen "tryToReceiveData:" und "parser:didParseInfoPacket:"?
4. Wozu sind die Pfeile an der Seite der Methoden da?
5. Welche Methode wird angezeigt wenn man auf den linken Pfeil bei "setInfoPacket:" klickt? In welcher Beziehung steht diese zu "setInfoPacket" und wovon hängt ab welche Methode angezeigt wird, wenn man auf den Pfeil klickt?
6. Warum ist der rechte Pfeil bei "parser:didParserInfoPacket:" ausgegraut?
7. Warum ist "tryToReceiveData" grau und hat keine Pfeile?
8. Was bedeuten die Gruppen von Ovalen am unteren bzw. oberen Rand der Zelle
9. Wozu sind die Balken zwischen den einzelnen Zellen da? Was bedeuten sie?
10. Warum sieht tryToReceiveData anders aus als die anderen Methoden und hat keine Pfeile?
11. Was bedeuten die drei Striche an der Seite von tryToReceiveData?
12. Du möchtest nun gerne zu der Stelle im Code springen an der eine bestimmte Methode implementiert wird. Was würdest du tun?
13. Du würdest nun gerne eine Übersicht über alle Alternativen zu einer Methode bekommen. Was würdest du tun? Was würdest du erwarten, dass dann passiert?
14. Was nimmst du an, dass passiert, wenn du auf eine der Methoden in der Seitenansicht klickst? (Ändert sich die Fokusmethode?)
15. Was nimmst du an, dass passiert, wenn du auf die Balken zwischen den Methoden klickst?
16. Wenn du auf den Balken zwischen den Methoden klickst, erscheint ein Menü mit der Übersicht der alternativen Methoden. Erscheint dir das sinnvoll?
17. Wonach sollte das Menü zur Anzeige der alternativen Methoden sortiert sein? Aufrufreihenfolge? Alphabet? Klassen und dann Alphabet? Etwas anderes?

18. *klick auf setInfoPacket:* Welche Bedeutung haben in dieser Ansicht nun tryToReceiveData und setInfoPacket:? Welche Rolle spielen sie?

19. Ich scrolle jetzt ein wenig im Stack. *klick auf tryToReceiveData und unteren Teil der Scrollbar um einen Stack mit Auslassungen zu bekommen* Schau dir die Zelle unter tryToReceiveData an. Was bedeutet sie? Welche Information kannst du entnehmen?

20. Weitere Kommentare?

6 Evaluation

6.1 Study Setup

Task 1 For a (hypothetical) trial version of Bibdesk, you want to add a limitation. This should add "TRIAL" in front of every paper's file name when using the "Autofile" feature. Where would you implement this change?

Hint: The BDSKLinkedFile class is used to represent linked files.

Task 2 One of your colleagues suggests implementing the change from 1 by adapting the parseFormat:forField:linkedFile:ofItem:suggestion: method in the BDSKFormatParser class. Which effects would this have in the UI?

Hint: The Autofile feature operates mainly in the background. The only part of the UI that is dedicated to the Autofile feature is the associated preference screen.

Post-Session Questionnaire

Participant ID:

	Strongly Disagree				**Strongly Agree**
1. I think that I would like to use this system frequently					
2. I found the system unnecessarily complex					
3. I thought the system was easy to use					
4. I think that I would need the support of a technical person to be able to use this system					
5. I found the various functions in this system were well integrated					
6. I thought there was too much inconsistency in this system					
7. I would imagine that most people would learn to use this system very quickly					
8. I found the system very awkward to use					
9. I felt very confident using the system					
10. I needed to learn a lot of things before I could get going with this system					
11. I found understanding the source code easy using Stacksplorer					
12. I do not think Stacksplorer has benefits for code understanding compared to Xcode					
13. I think navigation in source code is faster when using Stacksplorer (compared to vanilla Xcode)					
14. I found navigation using Stacksplorer awkward					
15. When using Stacksplorer I had a better idea of where I am in the source code compared to using plain Xcode					
16. I often felt lost in the source code when using Stacksplorer					

Index

Typeset September 29, 2011