Stacksplorer

Understanding Dynamic Program Behavior

Diploma Thesis at the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Jan-Peter Krämer

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Björn Hartmann

Registration date: Jul 23th, 2010
Submission date: Jan 24th, 2011

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, January 2011
Jan-Peter Krämer

Contents

List of Figures

Abstract

Even when using modern programming environments, comprehending source code is still a major problem for developers. Nevertheless, code comprehension is required to perform successful software maintenance. In the first part of this thesis, we analyze how developers use available tools to cope with this problem. We complement the results of previous studies to show that current IDEs often do not fulfill developers' requirements. The semantic of the source code is insufficiently visualized and not used in order to provide guidance for developers when navigating through the code.

In the second part of the thesis, we present Stacksplorer, a new tool to support source code navigation and comprehension. Stacksplorer visualizes potential call stacks in an application and allows to navigate along them. Thus, Stacksplorer exploits a semantic aspect of source code to suggest relevant methods for exploration. Information displayed in Stacksplorer is visually linked to the source code, the medium developers are most familiar with. A prototype of Stacksplorer was implemented as a fully functional IDE plug-in.

A user study showed that software maintenance tasks in a large open-source application could be completed significantly faster with Stacksplorer than without it. Participants reported that they were highly satisfied with the plug-in and would like to use it for real world projects.

Überblick

Trotz der Verfügbarkeit moderner Software-Entwicklungsumgebungen ist das Verstehen von Programmquellcode immer noch ein großes Problem für Softwareentwickler. Das Verständnis des Programmquellcodes ist allerdings eine notwendige Voraussetzung, um erfolgreich bestehende Software zu pflegen und zu korrigieren. Im ersten Teil der vorliegenden Arbeit stellen wir eine Studie vor, in der wir analysiert haben, wie Softwareentwickler vorhandene Werkzeuge nutzen, um mit dem Problem des Quellcodeverständnisses umzugehen. Wir erweitern damit die Erkenntnisse früherer Untersuchungen und zeigen, dass die heute verfügbaren Werkzeuge die Anforderungen von Entwicklern oft insbesondere deshalb nicht erfüllen können, weil die Semantik von Programmquellcode unzureichend visualisiert und nicht dafür genutzt wird, dem Programmierer Hilfestellung bei der Navigation anzubieten.

Im zweiten Teil der Arbeit stellen wir „Stacksplorer“ vor, ein neuartiges Werkzeug, das das Verstehen von und das Navigieren durch Programmquellcode erleichtern soll. Stacksplorer visualisiert mögliche “Call Stacks” in einer Applikation und erlaubt es, entlang der “Call Stacks” zu navigieren. Es nutzt also einen bestimmten Aspekt der Semantik des Quellcodes, um Stellen im Programm zu identifizieren, die möglicherweise für den Programmierer relevant sind. Die Informationen, die Stacksplorer anzeigt, sind visuell mit dem Quellcode, also dem Medium, das Programmierer am besten kennen, verbunden. Umgesetzt wurde Stacksplorer als voll funktionsfähiges Plug-in für eine Entwicklungsumgebung.

Eine Benutzerstudie konnte zeigen, dass Wartungsaufgaben in einem großen Open-Source Projekt, wenn Stacksplorer genutzt wird, signifikant schneller erledigt werden können als ohne Stacksplorer. Außerdem berichteten Teilnehmer der Studie, dass sie hochzufrieden mit dem Plug-in waren und dass sie Stacksplorer gerne bei ihrer alltäglichen Arbeit benutzen würden.

Acknowledgements

First of all, I want to thank everyone working at the Media Computing Group for their support. In particular, I want to thank Prof. Dr. Jan Borchers and my advisor Thorsten Karrer for their valuable help during the thesis. You and everyone else at the Media Computing Group contributed many helpful comments and advice. Thanks also to Jonathan Diehl, who supported me since I first joined the Media Computing Group in 2008, and to all other research assistants and long-term members of the group. All of you are great colleagues.

I also received very valuable input for the thesis from my second examiner, Prof. Dr. Björn Hartmann, who always answered questions via email elaborately and nearly immediately. Your expertise was very helpful for this thesis, thank you!

Special thanks to Prof. Dr. Leif Kobbelt and again Prof. Dr. Jan Borchers, who agreed to fund multiple trips to conferences, which were invaluable experiences.

Furthermore, I want to thank all participants of my studies again. Without you I would not have obtained any result; thanks for your time!

I also want to thank Bianca for her emotional encouragement. Last but not least, I am very grateful to my parents, who have always supported me.

Conventions

Throughout this thesis, we use the following conventions.

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

The whole thesis is written in American English.

In boxplot diagrams, the whiskers always extend to the maximal and minimal values in the dataset; the left box boundary marks the 25% percentile; the right boundary marks the 75% percentile. The thick line in the box indicates the median.

Throughout the thesis, estimates 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[1] 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"[2].

[1]http://www.dwheeler.com/sloccount/
[2]http://www.dwheeler.com/sloccount/sloccount.html

Chapter 1

Introduction

"The important thing is not to stop questioning;
curiosity has its own reason for existing."

—Albert Einstein

Source code comprehension is required to perform software maintenance.

Up to 70% of the total expenses of a software project are spend on software maintenance [Lientz, 1980], i.e., on adding new features, fixing bugs or performing refactoring after the software has been shipped. In order to work on these maintenance tasks, programmers have to understand the existing source code first [Boehm, 1976]. Code comprehension is also necessary if new developers join a team, a developer gets responsible for a new feature, or even if a developer has to come back to his own code after a while. Interestingly, LaToza *et al.* [2006] found that software development engineers at Microsoft considered understanding source code their biggest problem. As it also requires detailed understanding of existing source code, developers found it similarly problematic to be aware of side effects when implementing changes. To make things worse, the maintenance effort of source code is increasing. Maintenance effort is correlated to size, even for object-oriented software [Li and Henry, 1993]. Embedded software, for example, is doubling in size every two years [Ommering et al., 2000].

Nowadays, object-oriented programming languages are widely used.

The source code developers have to deal with nowadays is mostly object-oriented. Based on the Transparent Language Popularity Index[1] from December 2010, seven of the Top 10 most popular compiled programming languages are object-oriented, which account for more than 50% of the popularity of all compiled programing languages. On TIOBE's Programming Community index[2], among the Top 10 programming languages in 2010, only object-oriented ones could gain popularity in comparison to 2009's measurements. This indicates that the trend towards object-oriented languages is ongoing and these languages can be expected to become even more widespread in the near future.

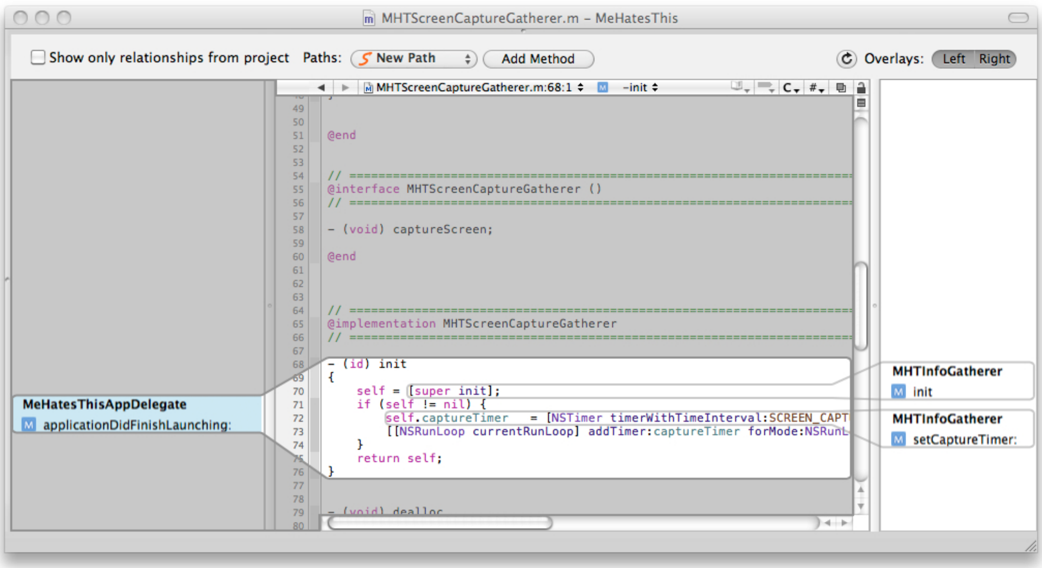
A lot of effort is put into the improvement of tools for software developers.

To support developers of object-oriented languages, integrated development environments (IDEs) are provided as tools. Over the last years, a lot of effort has been put in improving these IDEs. For example, new major versions of the popular Eclipse IDE[3] are released annually. In the Eclipse Marketplace[4], more than 900 plug-ins are available (at the time of writing) to extend the Eclipse IDE. Enhancements of development tools to make them more useful for programmers are also widely discussed in academia. Throughout the last 10 years, papers addressing this topic have been presented on every *Conference on Human Factors in Computing Systems* (CHI), the premier conference on Human-Computer Interaction. The *IEEE Conference on Program Comprehension*, whose 19th edition happens in 2011, is exclusively concerned with investigating how developers understand software and with tools supporting this activity.

Source code is mostly organized in multiple files.

Many popular IDEs, such as Eclipse (with a market share of over 50% in 2004[5]) or Microsoft's Visual Studio[6] (which is believed to be the most adopted IDE in enterprise development[7]), still primarily structure source code in sepa-

[1]http://lang-index.sourceforge.net/
[2]http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[3]http://www.eclipse.org
[4]http://marketplace.eclipse.org
[5]http://www.jboss.com/pdf/bzresearch$_s$*tudy.pdf*
[6]http://www.microsoft.com/visualstudio/en-us/
[7]http://adtmag.com/articles/2010/06/23/eclipse-5th-release-train-helios.aspx


MHTScreenCaptureGatherer.m – MeHatesThis
Show only relationships from project
Paths:
New Path
Add Method
Overlays:
Left
Right
MHTScreenCaptureGatherer.m:68:1
-init
MeHatesThisAppDelegate
applicationDidFinishLaunching:
MHTInfoGatherer
init
MHTInfoGatherer
setCaptureTimer:

```
@end

// ==========================================================================
@interface MHTScreenCaptureGatherer ()
// ==========================================================================

- (void) captureScreen;

@end



// ==========================================================================
@implementation MHTScreenCaptureGatherer
// ==========================================================================

- (id) init
{
    self = [super init];
    if (self != nil) {
        self.captureTimer   = [NSTimer timerWithTimeInterval:SCREEN_CAPT
        [[NSRunLoop currentRunLoop] addTimer:captureTimer forMode:NSRunL
    }
    return self;
}


- (void) dealloc
```

Figure 1.1: Stacksplorer provides information about potential call stacks including the currently edited method in two side columns next to the source code, and offers navigation along this information.

rate files. This representation does not reveal a lot about the source code's semantics. Hence, code comprehension is not supported well by a file-based representation. Previous research, which is introduced in depth in chapter 2.2 – "Programmers' Work Practices", and a study conducted by us (5 – "Navigation Behavior") have shown that this lack of support with program understanding is, in fact, problematic for developers.

Developers rarely consult documentation to understand source code.

The goal of this thesis is to design a new tool for developers to support code comprehension. Robillard *et al.* [2004] found that developers, who want to successfully modify source code they are not familiar with, have to follow structural dependencies when navigating. Furthermore, developers are often not willing to put effort in reading or updating documentation or similar documents; source code is always consulted first to solve a task [LaToza et al., 2006]. For the tool we created, this implies three requirements:

1. The tool should support browsing along structural dependencies.

2. The tool should work fully automatic, requiring no effort from the user.

3. The tool should work in conjunction or interlinked with a source code editor.

Stacksplorer visualizes information about potential call stacks in an application.

In the second part of this thesis, we present Stacksplorer (see Figure 1.1), an IDE plug-in fulfilling these requirements. Stacksplorer is a novel visualization technique providing information about possible call stacks in an application. For a *focus method*, which is marked by the cursor in a source code editor, Stacksplorer shows callers on the left side of the source code editor and methods called from the focus method on the right side of the editor. Users can navigate to methods shown in the side columns by clicking them.

Stacksplorer fulfills developers' requirements.

In the preliminary study we conducted (5 – "Navigation Behavior"), we showed that the call stack is one of the most important structural aspects of source code. Hence, Stacksplorer fulfills the first requirement. Furthermore, the side columns are updated automatically, which conforms to the second requirement. Optional overlays that connect entries in the side columns with method calls in the source code implement the third requirement.

In a user study Stacksplorer was evaluated with positive results.

We tested Stacksplorer's effectiveness for developers in a quantitative user study (7 – "Evaluation") with positive results. We could show that Stacksplorer was not only subjectively considered beneficial but it could also significantly decrease the time required to perform software maintenance tasks.

1.1 Chapter Overview

Chapter 2 In chapter two, we introduce some fundamental aspects of object-oriented development as well as the existing research about work practices of programmers and their use of current IDEs. These topics establish the theoretical foundation upon which Stacksplorer was invented.

Chapter 3 Related work in the field of tool support for developers is introduced in chapter three. We focus on tools that are related to Stacksplorer: Either they automatically determine which locations in source code are relevant for developers, or they introduce clever visualizations of relevant information.

Chapter 4 Because of the immense variety of programing languages and toolkits, we found it impractical to consider all of them in our studies and the design of the Stacksplorer prototype. Hence, we concentrated on developers working with Objective-C and the Cocoa framework. The peculiarities of this language and toolkit are introduced in chapter four.

Chapter 5 In an observational study, we confirmed that developers working with Objective-C and Cocoa exhibit the same navigational behavior as found in previous research. The results of this study, which complement previous work, are presented in chapter five.

Chapter 6 In chapter six, we introduce the software prototype of Stacksplorer. We describe in detail the design choices we made and explain interesting aspects of the implementation.

Chapter 7 The software prototype of Stacksplorer was evaluated in a user study, in which we compared the performance of developers working with and without Stacksplorer. We could show that participants using Stacksplorer were able to solve maintenance tasks faster. The design and the results of this second user study are described in chapter seven.

Chapter 8 During the user test, we noticed some potential for improvement of the first version of Stacksplorer. In chapter eight, we explain these issues and how we fixed them.

Chapter 9 In the last chapter, we summarize the contributions of this thesis and point to interesting research questions that were revealed while working on and evaluating Stacksplorer.

Chapter 2

Theory

"If someone claims to have the perfect programming language, he is either a fool or a salesman or both."

—Bjarne Stroustrup

Our work is rooted in two different theories, which are introduced in this chapter. Firstly, the goal of IDEs and improvements for them is to support the creation of good, (by now nearly exclusively) object-oriented software. Stacksplorer is also designed for object-oriented source code. We present the fundamental properties of object-oriented software development in the first section of this chapter. Secondly, users of Stacksplorer are programmers. In the second section of this chapter, we explain how programmers work with and think about source code. In the last section, we summarize previous studies analyzing how developers utilize existing tools that should support them with object-oriented development.

2.1 Object-oriented Software Development

Object-oriented development helps structuring source code.

Software quality can be assessed based on different criteria, e.g., portability, maintainability, understandability, or consistency [Boehm, 1978]. Most of these properties are positively correlated with a proper software architecture. Although generally no particular development technique is required to create good software architectures, today most software conforms to an object-oriented architecture, which has been proven to model many application domains sufficiently well [Meyer, 1997].

Objects are instances of classes, which are structured hierarchically.

In object-oriented software, application behavior is defined by interactions of *objects* at runtime. Each object is an instance of a *class*, which defines the set of properties the object has and the methods the object can perform. The class to which an object belongs is sometimes also called the object's *type*. Objects and classes often represent real world entities or groups of them. For example, in a school's management application, teachers might be represented by instances of a class `Teacher`. Classes are structured in a hierarchy of one ore more trees, which is called the *static object hierarchy* or *inheritance hierarchy*. Each class can declare properties and methods itself and inherits the properties and methods of its parent class. In some languages, e.g., in C++, a class might have more than one parent class, resulting in a more complicated inheritance hierarchy. In the school's management application example, instances of a class `MathTeacher` might inherit all properties and methods declared in a class `Teacher`, but in addition instances of `MathTeacher` are able to perform more complex algebraic operations. This implicit structure of object-oriented software can make software easier to manage.

Object-oriented languages are tools for object-oriented development.

Object-oriented software can be implemented in any programming language. However, modern object-oriented programming languages have been developed as tools for object-oriented software development. They allow easily defining classes, creating objects as instances of these classes, and they provide a syntax to call methods on these objects.

There are manifold ways to define useful classes and objects for an application. Real world entities can often be easily mapped to objects. Meyer [1997] called classes representing real world entities *analysis classes*. For more abstract parts of applications, e.g., a video post processing engine, other mappings have to be used. One typical approach is to encapsulate a single responsibility into a class. Meyer called theses classes *design classes*. For example, in a video processing engine that, at some point, needs to scale the video to half its size, there might be one class dedicated solely to video scaling. If real world analogies are missing, defining useful borders between object's responsibilities can become complicated and ambiguous. Because of these difficulties, the practical benefits of object-oriented development have been widely discussed (e.g., [Lewis et al., 1991; Potok et al., 1999]), with some researchers even suggesting that there is no benefit in using object-oriented architectures at all [Potok et al., 1999].

A class typically encapsulates a single responsibility.

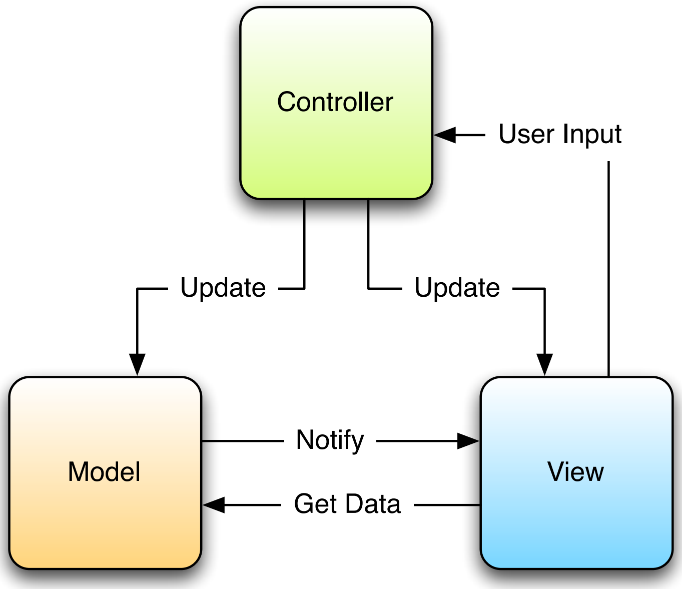


Figure 2.1: The MVC model suggests three categories of classes for applications with a graphical user interface: views, models, and controllers. Image adopted from [Apple, 2010a].

Modern user interface toolkits are object-oriented and use the MVC paradigm to split responsibilities among classes.

Today, many modern user interface toolkits, such as Microsoft's Windows Presentation Foundation[1] (part of .NET 3.0 and above), Apple's Cocoa[2], or Nokias's QT[3], are implemented using an object-oriented language. In these toolkits each individual user interface component is represented by an object. The popular *Model-View-Controller* (MVC) paradigm [Krasner and Pope, 1988] suggests a structure how responsibilities in applications with a graphical user interface can be split into classes. The *view* defines how the application's interface looks like and how data is presented. The application's data and the application logic are contained in the *model*. A view updates itself by reading data from the model; and a *controller*, which is associated to a view, defines how the model changes in response to user input. The MVC pattern is illustrated in Figure 2.1.

2.2 Programmers' Work Practices

Programmers, the target group of Stacksplorer, have developed specific work practices to cope with the complexity of source code. Previous studies describing these practices are presented in this section.

Bottom-up models suggest, source code is understood by iteratively understanding increasingly large blocks.

Starting in the 80's, a lot of studies tried to find cognitive models, which describe how developers create mental models of source code. Bottom-up cognitive models suggest that developers understand source code by first reading the code and then iteratively chunking it into larger blocks. Pennington [1987] showed that during this iterative process a hierarchy of more and more abstract representations of the programs' control flow, decision hierarchy, data flow, and goal/subgoal structure evolves.

Different developers ask similar questions when trying to understand source code.

Sillito *et al.* [2008] published a more recent iterative bottom-up model. They found that questions different developers ask when working with an unknown software project are similar. Their model consists of 44 questions grouped in four phases, which represent the iterative steps required to

[1]http://msdn.microsoft.com/en-us/library/ms754130.aspx
[2]http://developer.apple.com/technologies/mac/cocoa.html
[3]http://qt.nokia.com/

build a mental model of the source code. In the first phase, developers want to find focus points to begin the investigation from. Afterwards, in the second phase, the context of these focus points is explored. Ultimately, programmers try to answer questions concerning structures of a complete subgraph containing a focus point in the third phase, and concerning the relationship between multiple subgraphs in the fourth phase.

Top-down models suggest developers understand source code by trying to confirm hypothesis.

In contrast to bottom-up models, top-down models suggest that programmers understand source code by mapping the problem domain (possibly through intermediate domains) to the source code [Brooks, 1983]. To build this mapping, programmers start with a number of hypotheses they refine and test iteratively.

Of course, hybrid strategies are also feasible. Letovsky [1987] found that experienced programmers are able to utilize top-down or bottom-up strategies as needed.

Structured source code browsing leads to better code understanding, which is required to perform successful modifications.

In a study of Robillard *et al.* [2004] developers were asked to perform a complex change task in a 65KSLOC software artifact and their success was analyzed. Robillard *et al.* found that, in order to complete the task successfully, it was essential to adequately analyze the source code to find the appropriate locations for changes. In contrast, unsuccessful participants implemented all changes in the same place. Additionally, the amount of structurally guided navigation through the source code was positively correlated with task success; opportunistic source code browsing was shown to be less effective.

A program slice isolates a computational thread influencing a single variable.

A full comprehension of source code is often unnecessary or impractical because the complete project is far too big to be fully understood [Chen and Rajlich, 2000]. Programmers then try to distill information relevant for the current task from the source code. One technique used is called *program slicing* [Weiser, 1982]. A slice of a program contains all source code lines that could influence a given variable and hence isolates a single computational thread. Program slicing can be performed algorithmically. Many variants of this technique have been developed, for example, an algorithm that allows slicing (C++) class hierarchies and hence allows utilizing the slicing technique on a more abstract level [Tip et al., 1996]. Tools to visually support program slicing using a graph representation of different program slices are also available [Gallagher, 1996].

Beacons are recurring patterns in source code.

Reoccurring patterns or features in the source code that programmers recognize to typically implement certain structures or operations are called beacons [Brooks, 1983]. They help programmers to get a rough understanding of code blocks. Similarly to slices, this prevents programmers from having to analyze the full source code in in detail.

2.3 IDE Utilization

Modern IDEs mostly support file-based source code access.

IDEs should support programmers with code understanding. Today, IDEs are mostly designed for object-oriented development. Object-oriented source code is typically structured into different files, which each contain one (or occasionally more) class(es) including their methods. This representation does neither fully represent the static object hierarchy, nor does it reveal the interactions of the objects at runtime. Nevertheless, most of today's IDEs primarily rely on file-based source code navigation. For file-independent access to source code or other information, IDEs often include a project wide search and a tool to reveal the definition of a variable, a method, or, more generally, a symbol. More specialized features often depend on the toolkit the IDE is designed for, so IDEs' capabilities may differ and not all features might be equally beneficial for working with any toolkit. Features of IDEs that are relevant for navigation will be discussed exemplarily for a particular IDE later (4.2.1 – "Navigation Tools"). In this section, we introduce some of the previous work analyzing how developers work with IDEs in order to perform software maintenance.

Structured documentation about software is often missing.

LaToza *et al.* [2006] investigated the work of software development engineers (SDEs) at Microsoft by conducting surveys and interviews. They found out that SDEs spend most of their time fixing bugs, followed by enhancement and refactoring tasks. Required knowledge about the program domain is often gathered as needed and not documented in a structured way. The tools used are for the most part limited to various source code editors and debuggers. If these are not sufficient, the *owner* of the code, mostly the developer who originally wrote it, is consulted personally. This, in turn, leads to interruptions, which are not supported very well by current development tools, because these tools do not capture the current task's context.

Current navigation tools in IDEs are time-consuming and cumbersome to use.

In addition, current IDEs for the most part lack tools supporting navigation that is meaningful for programmers. By observing programmers working with Eclipse on five change tasks in a 500SLOC Java application, Ko *et al.* [2006] found that about a quarter of developers' time was spent navigating, either by following dependencies or by searching for names. Navigating indirect dependencies using scroll bars and the package browser accounted for 14% of the test period. The find and replace tool was also frequently used for various navigation tasks, although more suitable and effective tools are available in Eclipse (see also [Murphy et al., 2006]). To perform comparisons, developers had to navigate back and forth between code segments, because Eclipse uses a single editing window.

Chapter 3

Related Work

"I find that a great part of the information I have was acquired by looking something up and finding something else on the way."

—Franklin P. Adams

Stacksplorer stands in the tradition of previous research that has put a tremendous effort in improving IDEs to make development of well-structured applications easier and hence also accessible to a broader audience. We present two kinds of approaches in this chapter. The first category contains projects that aim to simplify the process of finding relevant parts of the source code. The second category of projects contains novel approaches to source code visualization that try to either speed up navigation or to reduce the need for navigation by laying out related source code fragments side by side.

3.1 Making Information Accessible

In this section we present tools that, like Stacksplorer, try to reduce the effort spent on navigating through source code by making relevant information easier to find. This can be done either by providing clever recommendations for users where to navigate to or by allowing users to search relevant source code using techniques that are more efficient and versatile than the usual textual search. The systems mostly differ in the criteria they apply to determine importance of information.

3.1.1 Recommender Tools

Recommender tools automatically show files containing task related source code.

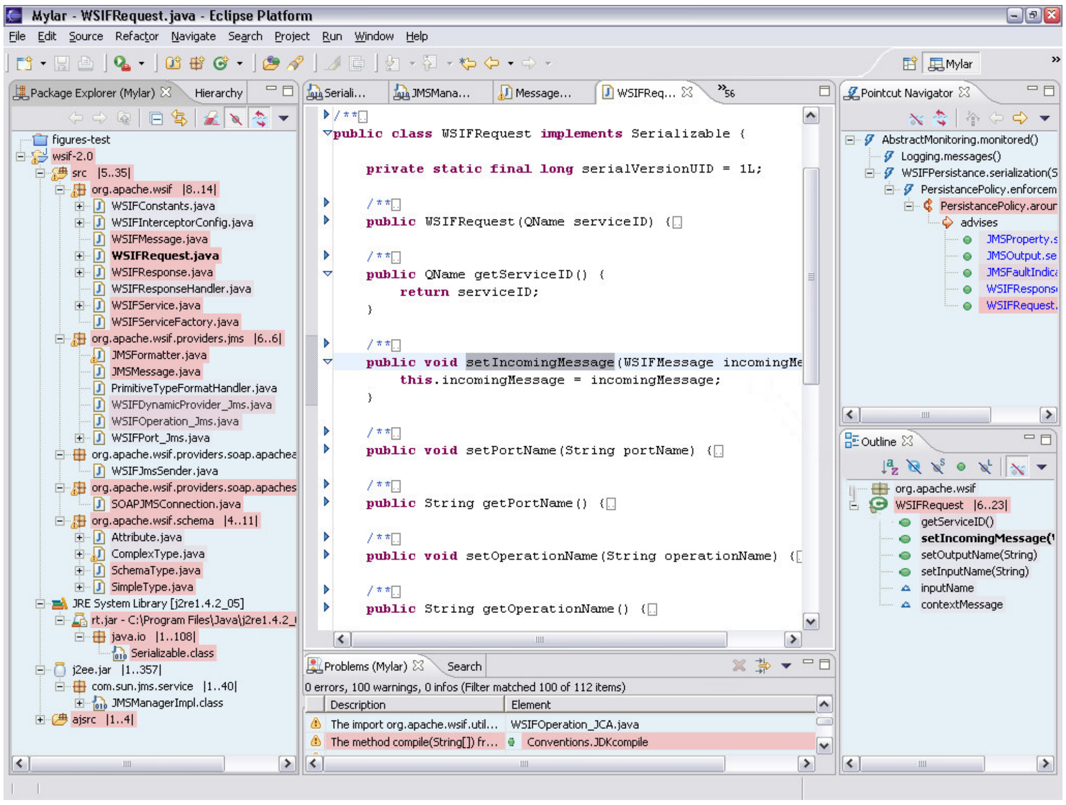
The purpose of a source code recommender system is to automatically find information related to the source code in the file a programmer currently works with. In this sense, Stacksplorer is also a recommender system. The systems presented here generally generate recommendations by calculating a *degree of interest*, which depends on the source code file that is currently edited, for all files in the project. They then recommend files with degree of interest exceeding some threshold.

Various techniques to determine degree of interest and to incorporate non-code information exist.

The degree of interest may be derived from a programmer's navigation history [Singer et al., 2005], maybe even including the navigation history of other authors of the same code [DeLine et al., 2005]. Kersten *et al.* [2005] propose to incorporate editing in addition to navigation activities when determining the degree of interest. Ĉubraniĉ *et al.* [2003] extend the analyzed data set with information other than source code, e.g., logs from version control systems. Robillard [2005] suggests to provide recommendations not only for a single file, which is currently edited, but for a user defined set of interesting elements in the source code. His fuzzy logic based algorithm extends the user defined set of elements with automatic suggestions.

Recommender systems were shown to be effective for small projects containing unknown source code.

User studies of the above mentioned tools consistently found that users considered automated guidance in large software projects helpful, especially if they were new to the projects. Evaluations with small tasks or case studies could also show that the total effort spent on navigating through the project could be significantly reduced.



```
/**
public class WSIFRequest implements Serializable {

    private static final long serialVersionUID = 1L;

    /**
    public WSIFRequest(QName serviceID) {

    /**
    public QName getServiceID() {
        return serviceID;
    }

    /**
    public void setIncomingMessage(WSIFMessage incomingMe
        this.incomingMessage = incomingMessage;
    }

    /**
    public void setPortName(String portName) {

    /**
    public String getPortName() {

    /**
    public void setOperationName(String operationName) {

    /**
    public String getOperationName() {
```

0 errors, 100 warnings, 0 infos (Filter matched 100 of 112 items)

	Description	Element
⚠	The import org.apache.wsif.util...	WSIFOperation_JCA.java
⚠	The method compile(String[]) fr...	Conventions.JDKcompile

Figure 3.1: Mylar [Kersten and Murphy, 2005] is a recommender system that filters files shown in Eclipse's standard tools based on their relevance for the developer's current task.

3.1.2 Query Languages

Query tools allow to exploit the source code's structure when searching.

Similar to recommender systems, the following systems using query languages try to reduce the amount of source code that is presented for exploration. Query tools do not automatically try to perform this reduction; instead, they offer efficient ways for users to find interesting source code. Query languages often allow searching by the means of structural relationships users know about, e.g., the call stack, which cannot be expressed easily using a textual search.

JQuery[1] [Janzen and Volder, 2003] is a logical programming language, similar to Prolog [Deransart et al., 1996],

[1]JQuery can be downloaded at http://jquery.cs.ubc.ca

```
class(?C, name, HelloWorld).
```

```
package(?P, class, ?C),
class(?C, super+, ?I),
interface(?I, method, ?IM),
method(?IM, signature, ?IS),
method(?IM, name, ?IN),
class(?C, method, ?CM),
method(?CM, signature, ?CS),
method(?CM, name, ?CN),
equal (?IS, ?CS),
equal(?IN, ?CN).
```

Figure 3.2: A JQuery defines a hierarchical browser. The simple query on the left defines a browser with all classes named "HelloWorld", the more complex query on the right produces a browser containing a class's methods grouped by the interface they belong to.

JQuery allows defining the content of a hierarchical file browser using a logical programming language.

to specify queries for source code. It allows querying for types and methods in Java source code. JQuery is expressive enough to exploit relationships that are meaningful for source code of object-oriented software. It is possible, for example, to query for the superclasses of an object. A hierarchical browser, similar to Eclipse's Package Browser, is used to display the results of JQueries. Each individual node in the browser can again be queried using a JQuery. A user study showed that users tended to formulate only rather simple queries.

In contrast to the evaluations of each individual tool presented so far, a larger study comparing JQuery and two recommender tools found that the tools had little effect in comparison to the generally dominating task and strategy specific effects [de Alwis et al., 2007].

UML defines several graph representations of source code.

Numerous tools use a graph-based representation of source code. For example, the call graph, which is visualized by Stacksplorer, is a graph representation of source code. More widely known examples are the diagrams defined in the unified modeling language (UML) [Object Management Group, 1997][2]. Particularly popular to provide an overview of source code is the class diagram (see Figure 3.3), which shows the static object hierarchy of an object-oriented soft-

[2]The UML is constantly developed further, the newest version is 2.3 [Object Management Group, 2010].

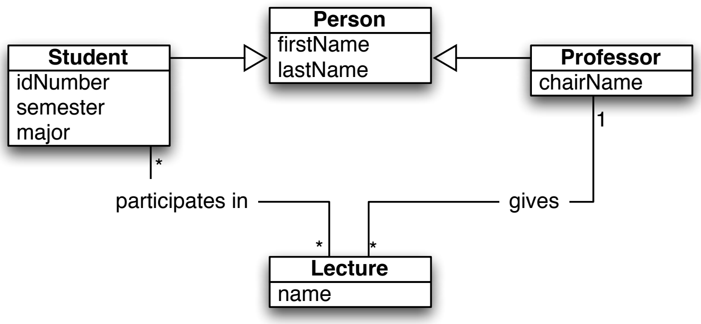


Figure 3.3: The exemplary UML diagram above shows inheritance (arrows) and associations between classes. The diagram does not depict how associations should be realized in source code. The labels on associations show how many instances of one class are associated with a single instance of the other class.

ware project and associations between instances of classes. To discern these different relationships between classes, multiple types of edges are used in the graph.

Robillard *et al.* [2002] create a graph that interlinks classes, fields, and methods in a single data structure to represent the structure of a program. A subset of this graph that contains all relevant structural information for a given task is called *concern graph*. The developer can interactively generate these concern graphs by querying vertices that are already part of the current concern graph. The graph itself is presented as a set of trees in an outline view, which allows opening a source code editor to display the source code associated with a node in the graph.

Concern graphs are user created graphs containing information about classes, fields and methods.

Query interfaces are not necessarily restricted to queries of source code. The Whyline [Ko and Myers, 2008] allows asking questions about the textual and graphical output of an application. Possible questions always start with "Why did" or "Why didn't"and refer to a particular output that the application produced during an execution. For example, a shape drawn in a painting application may, after the execution stopped, be queried "Why did this line's color = blue". Using a trace of the actual program execution, the

The Whyline searches a trace of an application's execution for the call stack, which is responsible for a particular output.

Whyline can then inform the user which call stacks influenced the relevant property. Using the Whyline, novice programmers could solve a bug fixing task significantly faster than expert developers without the Whyline.

Users need to pick the correct starting point to obtain useful recommendations or search results.

Recommender systems as well as query interfaces are only useful if users pick the correct query or start at a relevant file [Ko and Myers, 2008]. The Whyline was designed to solve this problem, because problems in the application output are easier to spot and name. However, even using the Whyline developers occasionally picked the wrong questions to start with in the conducted study.

3.2 Spatial Layouts

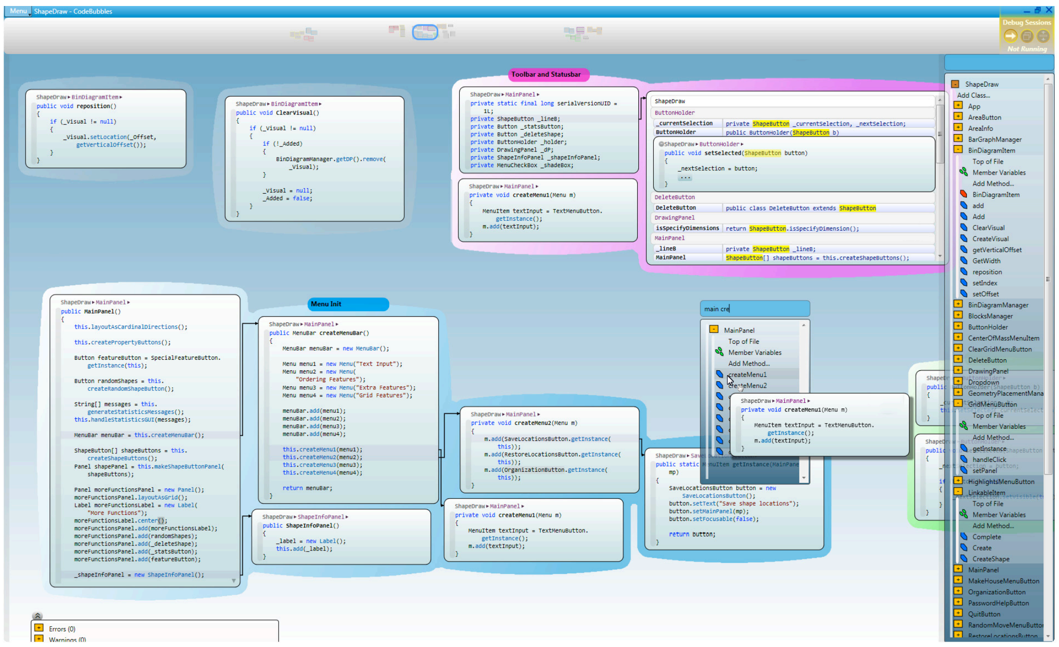
In spatial layouts information is arranged on a 2-D plane.

In contrast to recommender tools, spatial layouts do not help to retrieve interesting information in source code. They focus on laying out relevant source code elements visually in order to help developers finding and keeping track of source code. Many of theses visualization techniques exploit spatial memory, i.e., a human's ability to memorize where information is placed on a 2-D plane. Spatial layouts are similar to Stacksplorer, because they also provide visual representations of source code and (in some of the presented systems) its semantics.

3.2.1 Code Bubbles

In Code Bubbles, code editors are embedded in a graph visualization.

In Code Bubbles [Bragdon et al., 2010] code editors are embedded in a graph visualization. Starting with any bubble showing a part of the source code (e.g., a single method), users can use familiar tools like "Open Declaration" to find related elements of the source code that are opened in new bubbles. The bubbles are connected with arrows representing the relationship used to find them. Other relationships that exist between visible bubbles are inserted automatically. To facilitate space optimally, a lot of effort was put into automatically arranging bubbles and into reducing the width of overly wide bubbles by applying code reflow.



```
ShapeDraw▸BinDiagramItem▸
public void reposition()
{
    if (_Visual != null)
    {
        _Visual.setLocation(_Offset,
            getVerticalOffset());
    }
}
```

```
ShapeDraw▸BinDiagramItem▸
public void ClearVisual()
{
    if (_Visual != null)
    {
        if (!_Added)
        {
            BinDiagramManager.getDP().remove(
                _Visual);
        }

        _Visual = null;
        _Added = false;
    }
}
```

```
ShapeDraw▸MainPanel▸
private static final long serialVersionUID =
    1L;
private ShapeButton _lineB;
private Button _statsButton;
private Button _deleteShape;
private ButtonHolder _holder;
private DrawingPanel _dP;
private ShapeInfoPanel _shapeInfoPanel;
private MenuCheckBox _shadeBox;
```

```
ShapeDraw▸MainPanel▸
private void createMenu1(Menu m)
{
    MenuItem textInput = TextMenuButton.
        getInstance();
    m.add(textInput);
}
```

```
ShapeDraw
ButtonHolder
_currentSelection    private ShapeButton _currentSelection, _nextSelection;
ButtonHolder         public ButtonHolder(ShapeButton b)
  ShapeDraw▸ButtonHolder▸
  public void setSelected(ShapeButton button)
  {
      _nextSelection = button;
      ...
  }
DeleteButton
DeleteButton         public class DeleteButton extends ShapeButton
DrawingPanel
isSpecifyDimensions  return ShapeButton.isSpecifyDimension();
MainPanel
_lineB               private ShapeButton _lineB;
MainPanel            ShapeButton[] shapeButtons = this.createShapeButtons();
```

```
ShapeDraw▸MainPanel▸
public MainPanel()
{
    this.layoutAsCardinalDirections();

    this.createPropertyButtons();

    Button featureButton = SpecialFeatureButton.
        getInstance(this);

    Button randomShapes = this.
        createRandomShapeButton();

    String[] messages = this.
        generateStatisticsMessages();
    this.handleStatisticsGUI(messages);

    MenuBar menuBar = this.createMenuBar();

    ShapeButton[] shapeButtons = this.
        createShapeButtons();
    Panel shapePanel = this.makeShapeButtonPanel(
        shapeButtons);

    Panel moreFunctionsPanel = new Panel();
    moreFunctionsPanel.layoutAsGrid();
    Label moreFunctionsLabel = new Label(
        "More Functions");
    moreFunctionsLabel.center();
    moreFunctionsPanel.add(moreFunctionsLabel);
    moreFunctionsPanel.add(randomShapes);
    moreFunctionsPanel.add(_deleteShape);
    moreFunctionsPanel.add(_statsButton);
    moreFunctionsPanel.add(featureButton);

    _shapeInfoPanel = new ShapeInfoPanel();
```

```
ShapeDraw▸MainPanel▸
public MenuBar createMenuBar()
{
    MenuBar menuBar = new MenuBar();

    Menu menu1 = new Menu("Text Input");
    Menu menu2 = new Menu(
        "Ordering Features");
    Menu menu3 = new Menu("Extra Features");
    Menu menu4 = new Menu("Grid Features");

    menuBar.add(menu1);
    menuBar.add(menu2);
    menuBar.add(menu3);
    menuBar.add(menu4);

    this.createMenu1(menu1);
    this.createMenu2(menu2);
    this.createMenu3(menu3);
    this.createMenu4(menu4);

    return menuBar;
}
```

```
ShapeDraw▸ShapeInfoPanel▸
public ShapeInfoPanel()
{
    _label = new Label();
    this.add(_label);
}
```

```
ShapeDraw▸MainPanel▸
private void createMenu2(Menu m)
{
    m.add(SaveLocationsButton.getInstance(
        this));
    m.add(RestoreLocationsButton.getInstance(
        this));
    m.add(OrganizationButton.getInstance(
        this));
}
```

```
ShapeDraw▸MainPanel▸
private void createMenu1(Menu m)
{
    MenuItem textInput = TextMenuButton.
        getInstance();
    m.add(textInput);
}
```

```
ShapeDraw▸SaveLocationsButton▸
public static MenuItem getInstance(MainPanel
    mp)
{
    SaveLocationsButton button = new
        SaveLocationsButton();
    button.setText("Save shape locations");
    button.setMainPanel(mp);
    button.setFocusable(false);

    return button;
}
```

```
ShapeDraw▸MainPanel▸
private void createMenu1(Menu m)
{
    MenuItem textInput = TextMenuButton.
        getInstance();
    m.add(textInput);
}
```

Figure 3.4: Code Bubbles [Bragdon et al., 2010] is a development environment in which individual code fragments and related information, such as documentation, are shown and edited in bubbles, which can be arranged freely on a 2-D plane.

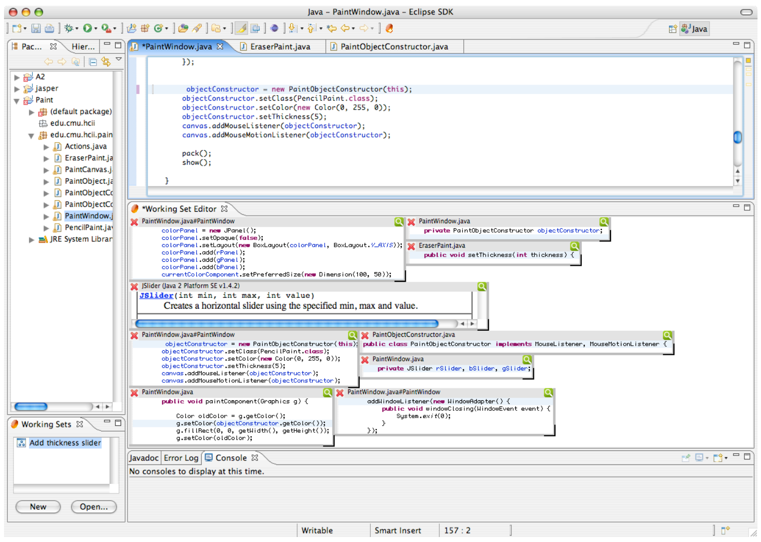
A qualitative study, where programmers were asked to complete a set of maintenance tasks with either Code Bubbles or Eclipse, showed that more tasks could be completed successfully with Code Bubbles and the total time required to complete a task could be reduced significantly for one of two tasks. Because related sections of the source code can be viewed side by side in Code Bubbles, the amount of navigation actions and especially the amount of back and forth navigation could be reduced significantly compared to the control group using Eclipse.

In a user study Code Bubbles allowed users to solve more tasks successfully.

3.2.2 JASPER

JASPER (Java Aid with Sets of Pertinent Elements for Recognition) [Coblenz et al., 2006] is an Eclipse plug-in to show a complete working set of related information. It allows users to add read-only representations of various types of information, such as text, URLs, and, of course, source code, to a 2-D plane where task relevant informa-

JASPER allows organizing task-relevant information on a 2-D plane.



```
        });

         objectConstructor = new PaintObjectConstructor(this);
        objectConstructor.setClass(PencilPaint.class);
        objectConstructor.setColor(new Color(0, 255, 0));
        objectConstructor.setThickness(5);
        canvas.addMouseListener(objectConstructor);
        canvas.addMouseMotionListener(objectConstructor);

        pack();
        show();

    }
```

PaintWindow.java#PaintWindow
```
        colorPanel = new JPanel();
        colorPanel.setOpaque(false);
        colorPanel.setLayout(new BoxLayout(colorPanel, BoxLayout.Y_AXIS));
        colorPanel.add(rPanel);
        colorPanel.add(gPanel);
        colorPanel.add(bPanel);
        currentColorComponent.setPreferredSize(new Dimension(100, 50));
```

PaintWindow.java
```
    private PaintObjectConstructor objectConstructor;
```

EraserPaint.java
```
    public void setThickness(int thickness) {
```

JSlider (Java 2 Platform SE v1.4.2)
```
JSlider(int min, int max, int value)
    Creates a horizontal slider using the specified min, max and value.
```

PaintWindow.java#PaintWindow
```
         objectConstructor = new PaintObjectConstructor(this);
        objectConstructor.setClass(PencilPaint.class);
        objectConstructor.setColor(new Color(0, 255, 0));
        objectConstructor.setThickness(5);
        canvas.addMouseListener(objectConstructor);
        canvas.addMouseMotionListener(objectConstructor);
```

PaintObjectConstructor.java
```
public class PaintObjectConstructor implements MouseListener, MouseMotionListener {
```

PaintWindow.java
```
    private JSlider rSlider, bSlider, gSlider;
```

PaintWindow.java
```
    public void paintComponent(Graphics g) {

        Color oldColor = g.getColor();
        g.setColor(objectConstructor.getColor());
        g.fillRect(0, 0, getWidth(), getHeight());
        g.setColor(oldColor);
```

PaintWindow.java#PaintWindow
```
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent event) {
                System.exit(0);
            }
        });
```

No consoles to display at this time.

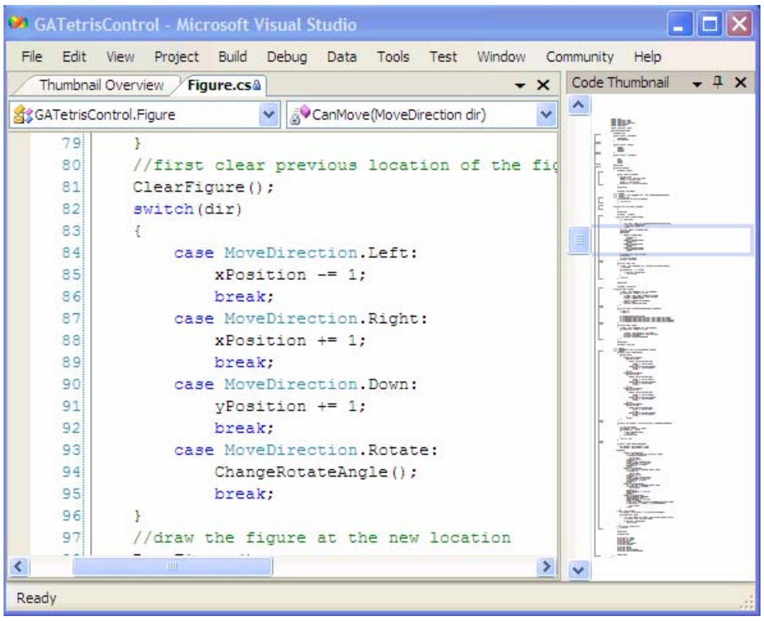
Figure 3.5: JASPER [Coblenz, 2006] allows collecting task relevant information in working sets on 2-D planes.

tion is collected. Multiple of these 2-D planes, which are also called working sets, can be managed, e.g., one for each task. Similarly to Code Bubbles, layout of the information on the plane is semi automatic. JASPER places new information without overlapping other items and resizes items to accommodate for the size of the complete plane. Users can change the layout determined by JASPER if they desire. Automatic adjustments to the layout never change the position of information on the plane. Otherwise, spatial memory could not be used to retrieve information on the plane.

3.2.3 Code Thumbnails

In Code Thumbnails, graphically scaled down versions of source code documents are used to support intra- and inter-file navigation.

Code Thumbnails [Deline et al., 2006] implements a different way to facilitate spatial memory to navigate to relevant aspects in source code. The system provides a graphically scaled down version of the file contents that is unreadable but allows discerning the structure of the source code. This thumbnail is used as a replacement for the traditional scroll bar for inter-file navigation. Multiple thumb-


GATetrisControl - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Test Window Community Help
Thumbnail Overview
Figure.cs
Code Thumbnail
GATetrisControl.Figure
CanMove(MoveDirection dir)
Ready

```
}
//first clear previous location of the fi
ClearFigure();
switch(dir)
{
    case MoveDirection.Left:
        xPosition -= 1;
        break;
    case MoveDirection.Right:
        xPosition += 1;
        break;
    case MoveDirection.Down:
        yPosition += 1;
        break;
    case MoveDirection.Rotate:
        ChangeRotateAngle();
        break;
}
//draw the figure at the new location
```

Figure 3.6: When used for inter-file navigation, Code Thumbnails [Deline et al., 2006] replaces the scroll bar with a scaled down version of the source code to allow exploiting spatial memory for navigation.

nails are shown on a flexibly organizable 2-D plane, the Code Thumbnails Desktop, to support intra-file navigation.

Search tasks could be completed significantly faster using Code Thumbnails.

An evaluation with 11 professional software developers showed that the new navigation technique was rapidly adopted. Code Thumbnails was preferred even for a time critical search task where participants had the option to use more familiar search techniques. The evaluation could also show that completion times for file searching tasks decreased significantly when using the Code Thumbnails Desktop.

Chapter 4

Prototyping Platform

"If there is ever a science of programming language design, it will probably consist largely of matching languages to the design methods they support."

—Robert Floyd

Before introducing our work on a new system to support developers when understanding source code, we discuss some aspects of the platform we have developed the prototype for. It makes sense to make a decision regarding the development platform early on, so that specific peculiarities of this platform can be considered throughout all design stages.

Stacksplorer is developed for programmers using Cocoa and Apple's Xcode IDE.

We chose to develop Stacksplorer for integration with Xcode[1], Apple's standard IDE. Xcode is built for development with the Cocoa framework, which is used to implement native applications for Mac OS X[2] and iOS[3]. Cocoa itself is written in Objective-C, which is also the programming language Cocoa developers have to use. (Bridges to Ruby and Python exist but are seldom used.) In theory, the restriction to a particular toolkit impacts the generalizability of our work. However, to develop and evaluate a work-

[1]http://developer.apple.com/technologies/tools/xcode.html
[2]http://developer.apple.com/technologies/mac/
[3]http://developer.apple.com/technologies/iphone/

ing prototype for and with multiple IDEs and languages is practically impossible. We hypothesized that developers working with Objective-C and Cocoa exhibit similar navigation behavior as found in previous studies for other languages and toolkits. To confirm this hypothesis we conducted a preliminary study, which we describe later (5 – "Navigation Behavior").

In this chapter we will introduce some of the fundamental properties of Objective-C and Xcode. They will be important to understand particular aspects of our study designs as well as the details of the implementation of our prototype.

4.1 Objective-C

Objective-C is an object-oriented programming language based on C.

Objective-C is, like C++, an object-oriented extension of C. In contrast to C++, which changes some of C's semantics, Objective-C is, syntactically, a strict superset of C. It adds a set of keywords to allow the definition and implementation of classes, as well as a syntax to send a message to an object. Conceptually, Objective-C is close to Smalltalk [Kay, 1993].

4.1.1 Messaging

Objective-C is dynamically typed.

Usually, a message is sent to an object in Objective-C by enclosing the object, the method name (which is called *selector* in Objective-C), and the parameters in brackets. Parameters for a method can be inserted in the method name after every colon. For example, the expression

```
[aString addString:anotherString
    withAttributes:attributes]
```

would call the method with the selector `addString:withAttributes:` on the object `anObject`, passing the parameters `aString` and `attributes`.

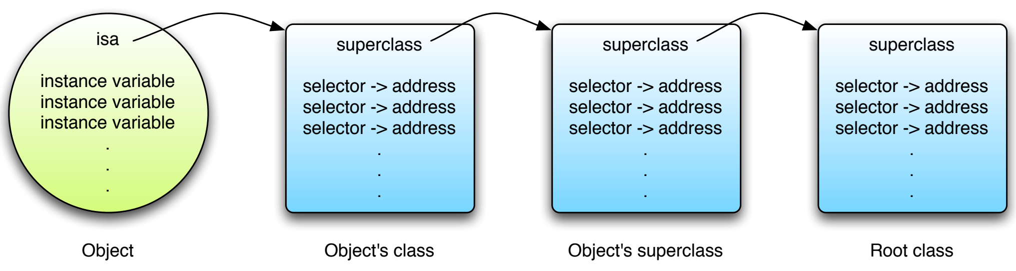


Figure 4.1: Messages in Objective-C are dispatched by traversing the static class hierarchy in the runtime system. Image adopted from [Apple, 2009].

Objective-C is a dynamically typed language. Message dispatch is performed at runtime, e.g., if and how `anObject` implements the `addString:withAttributes:` method is decided in the moment the method is called when the application runs.

Dynamic typing in Objective-C is implemented in the runtime system.

To implement dynamic typing, a runtime system is required that performs these decisions. The runtime system contains information about all classes and instantiated objects, and it offers an API, the Objective-C runtime library[4], to interact with this information directly. Each object is represented in the runtime system as a C struct, which stores the instance variables of the object and a pointer to the class of the object. The class is again stored as a C struct (each class in Objective-C is at runtime an instance of the class `Class`), which contains a pointer to the superclass and a dispatch table for messages. The dispatch table associates all selectors known to a class with a function pointer to their implementation.

Messages are mapped to their implementation at runtime when the message is sent.

If an object is sent a message, the runtime will follow the pointer from the object to the object's class and search in the class's dispatch table for the selector. If the selector is not found, the dispatch table of the superclass is queried. This way, queries move upwards the static object hierarchy until no more superclass exists because a root class of the class hierarchy is reached (see Figure 4.1). To increase performance, these lookups are cached, similarly to a previously

[4]http://developer.apple.com/mac/library/documentation/Cocoa/Reference/ObjCRuntimeRef/Reference/reference.html

suggested technique for Smalltalk [Deutsch and Schiffman, 1984].

The parameters `self` and `sel` are passed to method implementations implicitly.

A C function to which a selector is mapped in the dispatch table does not implicitly know that it is an instance method of an object. That is why these C functions take two parameters in addition to those the developer specifies when implementing the method: `self`, the object on which the method is called, and `sel`, the method's selector. Both parameters can be used in a method's implementation. They are passed to the C functions by the runtime system when dispatching a method and do not need to be passed explicitly.

4.1.2 Memory Management

Objective-C's garbage collector frees all non-reachable objects.

Objective-C offers two different memory management strategies: Reference-counting and garbage collection. The latter is only available on the Mac, not on iOS. When garbage collection is enabled, a garbage collector automatically frees unused memory from time to time [Apple, 2010b]. Therefore, the garbage collector creates a set of all reachable objects. This set is comprised of a fixed set of root objects, which are expected to always exist, and all objects that are connected to these root objects through a path of strong references. All references are strong references unless they are specifically marked as weak. Once the set of reachable objects is determined, the garbage collector frees all objects not in this set. In most cases, a developer does not need to care about memory management a lot when garbage collection is enabled.

Objects may assign, retain, or copy other objects stored in instance variables.

When using reference-counting, objects store a *retain count*. Initially, after allocation, an object's retain count is one. As soon as it reaches zero, the object will be deallocated. The developer is in charge to *retain* an object (increase the retain count by one) if a reference to the object is to be kept around and to *release* (decrease the retain count by one) the object if the reference is no longer needed. For an object `A` stored in an instance variable of another object `B`, typically, three options exists: Commonly, `B` *retains* `A`, so `A` cannot be deallocated without `B` releasing `A`. In this case `B` *owns* `A`. If

B does not own A, A should be just *assigned* to B, i.e., B does not retain A. In this case, B can not influence if and when A is released or freed. The last option is that B stores a *copy* of A, which is a new object and hence independent of whatever happens to A.

4.1.3 Declared Properties

Declared properties support automatic generation of accessor methods.

In well-designed object-oriented software, a pair of accessor methods (getter/setter) is typically used to access an instance variable. This realizes the principle of information encapsulation [Kim and Lochovsky, 1989]. Declared properties add twofold support for using this pattern in Objective-C. Firstly, syntactic features are added to Objective-C that allow to declare accessor methods for an instance variable. Developers can also define if the instance variable should be assigned, retained, or copied. Optionally, the according implementation can be generated automatically. Per convention, getters are named like the property; setter's names are constructed by prefixing the property name with `set` and capitalizing the first character of the property name.

The dot operator offers an alternative syntax to call accessor methods.

Secondly, the declared properties feature introduces a dot (.) operator, which can be used as an alternative to square brackets when calling an accessor method. The new operator does not add any new features, but it allows for very compact and readable source code. For example, the expression `object.propertyName`, which uses the dot operator, is equivalent to `[object propertyName]`. Using the dot operator more than once in a single expression is also possible, e.g., `object.aProperty.anotherProperty` is equivalent to `[[object aProperty] anotherProperty]`. The dot syntax can be used to call setters if an equal sign follows the property name. For example, the expression `object.aValue = 10` could be replaced with `[object setAValue:10]`.

```
static char addedPropertyKey;

- (id)addedProperty;
{
    return objc_getAssociatedObject(self, &addedPropertyKey);
}

-(void)setAddedProperty:(id)newValue;
{
    objc_setAssociatedObject(self,
                             &addedPropertyKey,
                             newValue,
                             OBJC_ASSOCIATION_RETAIN);
}
```

Listing 4.1: The minimal implementation for getters and setters of an instance variable, which is added to a class in a category.

4.1.4 Plug-Ins

Loading a plug-in for an Objective-C-based application adds information about a number of classes to the runtime system.

Because the complete information about the static object hierarchy can be manipulated at runtime through the runtime system, it is easy to load and link additional classes at runtime. This is how plug-ins are realized in Objective-C. A plug-in is a compiled version of information about a number of classes, their methods and instance variables. If a plug-in is loaded, these classes are made available in the runtime and then behave as if they belonged to the application from the start. Using the runtime API, plug-ins can not only add classes to an application but also change existing ones.

Associated references provide a workaround to add instance variables to an object from a category.

The simplest way to change an existing class is to write a category. A category for a class contains methods that are added to the class's dispatch table in the runtime system. However, categories have two practical limitations: Firstly, because the category might be loaded when objects are already instantiated, categories can only add methods, no instance variables. Adding instance variables to a class would require to allocate more memory for every instance of the class. Therefore, a reference to all instances of the class would be required. Instead, *associated references*, which were introduced in Mac OS X 10.6, can be used. Associated references allow to add storage for an associated object to an object. The associated object is referenced through a key, so an arbitrary number of objects can be associated with an object. Additionally, associations ensure proper memory management, as properties would do (4.1.3 – "Declared Properties"). The minimal source code required to add an (retained) associated object with a getter and setter to a class from a category is shown in listing 4.1.

Method swizzling allows exchanging the implementations of two methods.

A second limitation of categories is that, although methods can be overwritten in a category, there is no way to call the old implementation that has been overwritten like it would be possible when implementing a subclass. A workaround for this problem involves using *method swizzling*, a technique that allows exchanging the implementations of two methods. That means, in the runtime system the function pointers for two entries in the dispatch table are exchanged. Using method swizzling, overriding a method in a class and calling the old implementation from the new one is possible in three simple steps:

1. The new implementation of the method is added to a category using a new method name. For example, when overriding a method called `fullName`, a new method called `swizzleFullName` is added to a category.

2. At the point in code where the old implementation should be called, a call to the new method on `self` is inserted.

3. The implementations of the old and the new method are swizzled. This will cause the new method to be executed when the old one is called. Additionally, the call to the new method from step 2 will actually execute the old implementation.

A class's `load` method is called when the class is added to the runtime system.

The implementation of a simple `NSObject` category that allows to swizzle the implementations of two methods from `aClass` by calling

```
[aClass exchangeInstanceMethod:@selector(old)
                    withMethod:selector(new)]
```

```
@implementation NSObject (MethodSwizzling)

+ (void)exchangeInstanceMethod:(SEL)sel1 withMethod:(SEL)sel2;
{
    Method method1 = class_getInstanceMethod([self class], sel1);
    Method method2 = class_getInstanceMethod([self class], sel2);
    method_exchangeImplementations(method1, method2);
}

@end
```

Listing 4.2: Implementation of an `NSObject` category containing a convenience method to swizzle the implementations of two instance methods.

is shown in listing 4.2. A good place to perform method swizzling is the class's `load` method, which is called once the class is added to the runtime. If the `load` method is overwritten in a category, it will be called after the class's original `load` method was called, so both will be executed.

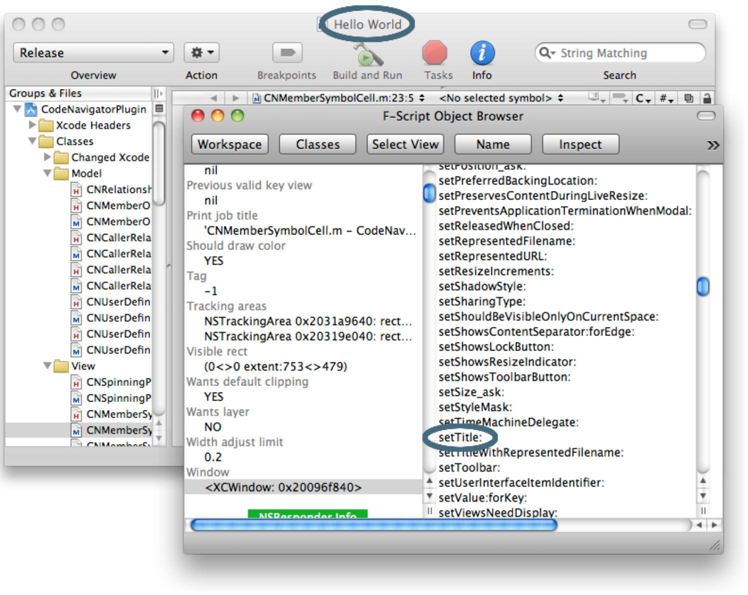
4.1.5 Reverse Engineering

Before the techniques explained above to change classes from within a plug-in can be applied, reverse engineering is required to find the appropriate locations for changes. In particular, four tools have proven to be useful to introspect an Objective-C application.

class-dump[5] The class information, which is available through the runtime system, is stored in the compiled binary files. This information can be extracted using `class-dump`. The tool generates class header files, i.e., the class interfaces, from a given binary. Developers of plug-ins can than refer to these generated headers if the original source code of the application is not available.

F-Script Anywhere[6] F-Script is an object-oriented scripting language, which uses Objective-C and Cocoa

[5]http://www.codethecode.com/projects/class-dump/



```
    nil
Previous valid key view
    nil
Print job title
    'CNMemberSymbolCell.m – CodeNav...
Should draw color
    YES
Tag
    -1
Tracking areas
    NSTrackingArea 0x2031a9640: rect...
    NSTrackingArea 0x20319e040: rect...
Visible rect
    (0<>0 extent:753<>479)
Wants default clipping
    YES
Wants layer
    NO
Width adjust limit
    0.2
Window
    <XCWindow: 0x20096f840>
```

```
setPosition_ask:
setPreferredBackingLocation:
setPreservesContentDuringLiveResize:
setPreventsApplicationTerminationWhenModal:
setReleasedWhenClosed:
setRepresentedFilename:
setRepresentedURL:
setResizeIncrements:
setShadowStyle:
setSharingType:
setShouldBeVisibleOnlyOnCurrentSpace:
setShowsContentSeparator:forEdge:
setShowsLockButton:
setShowsResizeIndicator:
setShowsToolbarButton:
setSize_ask:
setStyleMask:
setTimeMachineDelegate:
setTitle:
setTitleWithRepresentedFilename:
setToolbar:
setUserInterfaceItemIdentifier:
setValue:forKey:
setViewsNeedDisplay:
```

Figure 4.2: F-Script can be used for runtime introspection of Objective-C applications. Here, F-Script is used to browse through the methods of Xcode's window class and the window's title was changed using the `setTitle:` method (see marked spots).

classes internally. F-Script Anywhere can be loaded as a plug-in into any Objective-C application. It allows browsing graphically through the dynamic object hierarchy of the application and through all classes available in the application's runtime system. Additionally, F-Script Anywhere allows interacting with objects while the original application is running. Methods can be called ad-hoc through a graphical interface, with the effects immediately visible in the application (see Figure 4.2).

GDB[7] Of course, GDB, one of the most widely used debuggers, also supports Objective-C. GDB can be attached to every running application to perform reverse engineering. Unfortunately, proprietary applications typically contain no debugging information, so it is not possible to step through the im-

[6]http://www.fscript.org/
[7]http://www.gnu.org/software/gdb/

plementation of a method. It is, though, possible to break on method calls, because method names in clear text are required at runtime for method resolution in Objective-C. Then, the call stack can be explored by either showing a backtrace or by executing instructions stepwise, to advance to methods called from the current one. This technique helps to understand how methods use each other. Further, arguments passed to methods can be revealed in GDB, to understand how a method is used. This is possible, because arguments for a method call are always stored in the same CPU registers. For example, on an x86_64 architecture, `$rdi` holds the `self` parameter of the method call, `$rsi` holds the `sel` parameter, `$rdx, $rcx, $r8, $r9` hold the first four parameters for the method. For other architectures, the appropriate registers are different[8]. However, regardless of the architecture, the registers can be inspected easily with GDB.

Instruments "Instruments" is a performance measurement utility, which comes with Apple's developer tools. Although it is not designed for reverse engineering, Instruments can, among other things, sample which methods are executed by an application. Call stacks executed while the program was running can be browsed graphically afterwards.

4.1.6 Cocoa

Cocoa is the framework provided by Apple for Mac and iOS development.

Cocoa and Cocoa Touch are the frameworks provided by Apple to develop software for Mac OS X and iOS respectively. The versions for both systems differ mostly in their respective UI toolkit. Apart from the UITK only minor differences between the two versions exist.

The UI toolkits of both Cocoa versions adapt the MVC paradigm, which was introduced before (2.1 – "Object-oriented Software Development") as a pattern to split responsibilities in applications with a graphical user interface

[8]http://www.clarkcox.com/blog/2009/02/04/inspecting-obj-c-parameters-in-gdb/

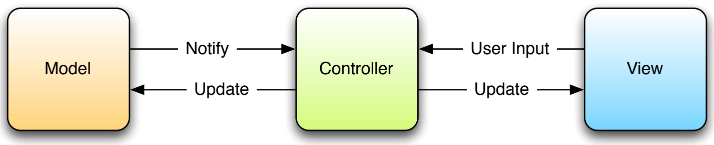


Figure 4.3: In Cocoa, all communication between model and view is mediated by a controller. Image adopted from [Apple, 2010a].

into classes. Apple's implementation [Apple, 2010a], however, is different from the original version of the MVC pattern. Whereas in the original version of the MVC pattern, a view updates itself by reading data from a model, in Cocoa controllers mediate all communication between models and views. Consequently, in Cocoa, the controller is in charge of reformatting model data for display in a view. Apple's implementation of MVC is depicted in Figure 4.3.

Im Apple's implementation of the MVC pattern, all communication between model and view is mediated by a controller.

In the following sections, we give an outline about some of Cocoa's most important design patterns, because these structures are important to keep in mind when analyzing where and why programmers navigate through source code.

Delegation

Typically, in an object-oriented user-interface toolkit, widgets or, more generally, objects are specialized for the use in a particular application by subclassing them. In Cocoa, subclassing is often avoided by using delegation. Delegation allows an host object to hand off control to a delegate object, which can supply application specific behavior. To implement delegation, the host object defines a protocol, i.e., a set of methods to implement, containing the methods it will call on the delegate while executing a task. All methods in this protocol are optional; hence, the delegate object only needs to implement those that are required to achieve the desired effect. Before the host object calls one of the delegate methods, the runtime is queried to determine if the delegate implements that method. Because the

Delegation allows refinement of objects without subclassing.

runtime is able to provide this information, the type of the delegate object is irrelevant and generally unknown to the host object.

Notifications

Objects can send notifications to inform interested objects about changes.

Notifications are a mechanism for an object to inform any interested object about changes. This inter-object communication is mediated by a *notification center*. An object, the *poster*, can send a message with a unique name, an object, and a dictionary containing additional information to the notification center. Other objects, the *receivers*, can register at the notification center to receive notifications with a specific object or a specific name or both. The notification center is responsible for delivering the correct notifications from the posters to the receivers.

Receivers of a notification can change at runtime and are unknown to the poster.

From a software architectural standpoint, notifications are similar to a method call from a poster to all receivers. However, the receivers can change at runtime and are not known to the poster, what makes notifications extremely flexible. This flexibility also makes it harder for a developer to understand the control flow by reading the source code.

Concurrency Programming

Operation queues support implementing concurrency.

Operations and operation queues provide a powerful way to implement concurrency, i.e., to have multiple things happen in parallel. For example, complex calculations should happen while the user can still interact with the user interface. An `NSOperation` encapsulates a single task that is supposed to run concurrently to something else. Instead of running the operation on a background thread manually, as it is usually required for multithreaded applications, it can be scheduled on an `NSOperationQueue`. Operation queues launch each operation on a separate thread automatically, and they can decide how many operations should run in parallel to facilitate a computer's capabilities best. The operation queue executing a part of the source code can be determined programmatically.

4.2 Xcode

Xcode is Apple's IDE for the development of Cocoa-based applications. It ships with a variety of tools to help programmers developing, testing, and deploying their applications. Most importantly, besides Xcode, the developer tools include Interface Builder, a tool to graphically lay out user interfaces, and Instruments, a performance analysis tool.

Xcode is an IDE for development with Cocoa.

Xcode is not localized and only available in English. For our work, this implies we will also develop our software localized to English only. Additionally, we can assume that users are used to reading left-to-right. This is important, because it allows us to assume that UI elements are also read or used from left to right, due to a cultural constraint [Norman, 1988].

Xcode's user interface is only available in English.

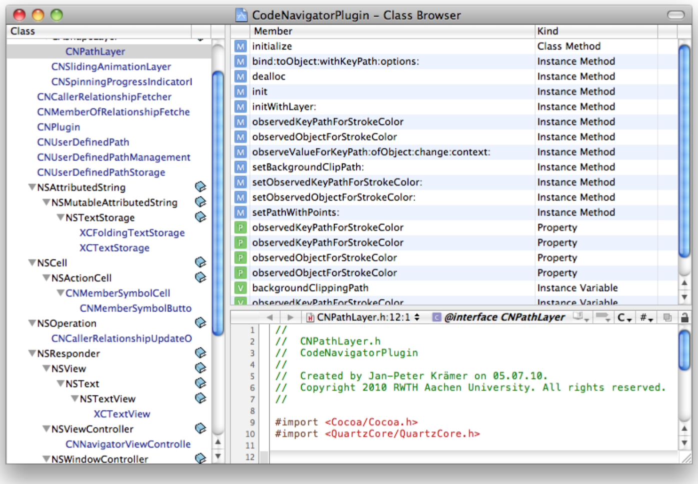
While Xcode offers a wide range of editing, refactoring, and debugging tools, for the purpose of this work we are only interested in the tools to navigate and understand source code as well as in Xcode's plug-in API. Throughout this thesis, we always refer to Xcode in version 3.2.4.

4.2.1 Navigation Tools

Xcode already includes tools that can aid navigation through and understanding of a project's source code. In particular, we identified the following 13 tools to be relevant for navigation.

File Browser The file browser is shown at the left side of Xcode's main window by default and allows browsing and accessing files in the project.

Jump to Definition This feature allows to open the definition of a selected symbol in the current editor. It can be accessed from the context menu of a symbol or by double-clicking the item while holding down the command key. If the symbol is ambiguous, i.e., multiple symbols with the same name exist, a menu is



```
CNPathLayer
  CNSlidingAnimationLayer
  CNSpinningProgressIndicatorl
CNCallerRelationshipFetcher
CNMemberOfRelationshipFetche
CNPlugin
CNUserDefinedPath
CNUserDefinedPathManagement
CNUserDefinedPathStorage
▼NSAttributedString
  ▼NSMutableAttributedString
    ▼NSTextStorage
      XCFoldingTextStorage
      XCTextStorage
▼NSCell
  ▼NSActionCell
    ▼CNMemberSymbolCell
      CNMemberSymbolButto
▼NSOperation
  CNCallerRelationshipUpdateO
▼NSResponder
  ▼NSView
    ▼NSText
      ▼NSTextView
        XCTextView
  ▼NSViewController
    CNNavigatorViewControlle
  ▼NSWindowController
```

Member	Kind
initialize	Class Method
bind:toObject:withKeyPath:options:	Instance Method
dealloc	Instance Method
init	Instance Method
initWithLayer:	Instance Method
observedKeyPathForStrokeColor	Instance Method
observedObjectForStrokeColor	Instance Method
observeValueForKeyPath:ofObject:change:context:	Instance Method
setBackgroundClipPath:	Instance Method
setObservedKeyPathForStrokeColor:	Instance Method
setObservedObjectForStrokeColor:	Instance Method
setPathWithPoints:	Instance Method
observedKeyPathForStrokeColor	Property
observedKeyPathForStrokeColor	Property
observedObjectForStrokeColor	Property
observedObjectForStrokeColor	Property
backgroundClippingPath	Instance Variable
observedKeyPathForStrokeColor	Instance Variable

```
//
//  CNPathLayer.h
//  CodeNavigatorPlugin
//
//  Created by Jan-Peter Krämer on 05.07.10.
//  Copyright 2010 RWTH Aachen University. All rights reserved.
//

#import <Cocoa/Cocoa.h>
#import <QuartzCore/QuartzCore.h>
```

Figure 4.4: Xcode's Class Browser allows browsing the inheritance hierarchy of a project.

shown from which the user can select which symbol's definition Xcode should open. If the implementation for the symbol is located inside the project, it will be opened, otherwise the declaration will be shown.

Project-wide Search The project-wide search searches in all files of the project for either text, symbols, or by matching a regular expression.

Find (Selected Text) in Project A feature that allows starting a project-wide search from the context menu for selected text or a symbol.

Search Documentation An incremental textual search can be used to quickly find information in the documentation.

Find (Selected Text) in Documentation This feature allows starting a search in the documentation from the context menu of selected text, similar to the "Find (selected text) in project" feature.

Switch to Header/Source File The public interface of a class is typically declared in a separate header file. This feature allows switching between this header file and the implementation of the class by clicking a button or using a keyboard shortcut.

Class Browser The class browser (see Figure 4.4) shows the static class hierarchy, methods and member variables of classes, and the corresponding implementation.

File History A history of visited locations in the source code can be browsed using forward and backward navigation like in a web browser.

Bookmarks Lines in source code files can be bookmarked and accessed later from the file browser.

Open Quickly A quick incremental search through symbol and file names, that can be accessed via a hotkey.

Single Step Advance (Debugger) A feature of the debugger that allows advancing program execution line by line.

Call Stack (Debugger) If a program is paused or crashed, the current call stack for each thread can be explored from the debugger.

In comparison to other IDEs, such as Eclipse or Microsoft's Visual Studio, we think Xcode's selection of tools is quite exemplary for a modern IDE. The most important differences to both Eclipse and Visual Studio are that Xcode, firstly, does not support tabs, which can help users to maintain a set of source code they are currently interested in; secondly, Xcode has no feature dedicated to revealing callers of a method.

Xcode provides similar tools as other IDEs.

4.2.2 Plug-in API

Xcode's plug-in API utilizes Objective-C's dynamic binding capabilities. Xcode automatically loads all bundle files whose file extension is `pbplugin` and that are located in either the user's or the system's `Library/Application Support/Developer/Shared/Xcode/Plug-ins` path. Interesting for plug-in development are the possibilities

Xcode automatically loads plug-ins located in a specific folder.

Xcode offers internally. We will now give an overview about the subsystems that were particularly interesting for Stacksplorer.

Project Index

The project index stores information about classes, categories, and protocols.

The *project index* contains information about all classes, categories, and protocols that are defined or used (through a framework) in a project. Xcode creates a project index for each project in the background.

Information in the project index is represented by instances of `PBXSymbol` or a subclass.

Internally, each project is represented in Xcode by an instance of `PBXProject`. It can be queried for the project's index, an instance of `PBXProjectIndex`. This class allows performing various queries on the project index, e.g., it can determine to which class a given line in a given file of the source code belongs. Each indexed item, such as a class or a method, is represented by an instance of `PBXSymbol` or one of its subclasses. These objects also structure the available information. For example, a `PBXClassSymbol` instance, representing information about a particular class in the project, can return an array of its methods, each represented by a `PBXMethodSymbol`. Multiple `PBXSymbol`s may be instantiated for the same represented entity. Generally, they can most reliably be tested for equality by comparing their name and their container symbol's name.

Source Scanner

The source scanner implements a lexer for languages supported by Xcode.

The project index stores information about everything defined in any source file in the project, but not about how it is implemented. This is done by the *source scanner*. A `XCSourceScanner` is able to parse a single source file according to a grammar specifying the language's syntax. Each symbol in the grammar is represented by an instance of `XCSpecification`. The scanner builds up a tree of `XCSourceScannerItem` instances, which represent the source code. In this tree, a source scanner item can have children if it represents a pair of expressions that enclose other source code (e.g., parenthesis). The source scanner can be queried either for symbols of a specific type or for symbols at a specific location in source code.

Code Completion Engine

The code completion engine provides access to another parsing engine in Xcode, the *CParser*. This parser is written in C++ and hence cannot be used by a plug-in like Objective-C classes can be. The code completion engine, however, uses the CParser and is accessible through an Objective-C class. Similarly to the source scanner, the CParser scans a single source file. It is superior to the source scanner in its ability to determine the type of expressions from the source code. This allows the code completion engine, if invoked for an expression, to return the type of the expression as well as a list of methods that this type responds to.

The CParser can determine the type of an expression in source code.

When using the code completion engine programmatically to complete a given expression in a file, an instance of the `PBXCodeCompletion` class has to be set up with the project index for the project containing the file. Furthermore, the class containing the expression has to be specified, and the code completion needs to know if the expression is used in an instance method or in a class method. Afterwards, parsing requires two steps: Firstly, the code completion parses the source code of the method containing the expression in order to find the locally defined variables; then, the expression can be passed to the code completion instance to obtain its type and a list of completion suggestions. The returned list is exactly equivalent to the one a user could obtain by placing the cursor at the end of the expression and invoking the code completion there.

The CParser's features can be utilized through the code completion engine.

Project Search

Project-wide search was already explained as a navigation tool previously (4.2.1 – "Navigation Tools"). It can, however, also be used programmatically. The different kinds of project-wide searches, e.g., textual or regu-

The project-wide search can also be invoked from source code.

lar expression based search, are implemented as different `PBXBatchFinder` subclasses. To perform a search, an instance of one of these subclasses is set up with a list of projects to search in and the query string. Additional options can be configured by passing an instance of the `PBXFindOptions` class to the `PBXBatchFinder` instance. Commonly used find options, e.g., for a project wide search, are available through a class method of `PBXFindOptions`. The actual search is automatically performed in the background. To get informed about new results, an object has to register itself as notification observer for the finder. Retrieving the results is a two-step process: Firstly, the finder is queried for all files that contain results; then, for each file the list of results can be obtained.

A hack is required to hide the project-wide search from the file browser.

A problem when using the project-wide search is that searches automatically show up in the file browser's category "Find Results". To change this behavior, we added a Boolean variable to the finder class that determines if the search shows up in the file browser. We also had to accordingly change the class that is responsible for displaying the "Find Results" category. The full source code for this change is available in appendix A – "Hide a Search from the Project Browser".

Code Editor

Xcode's code editor is based on Cocoa's text system.

Xcode's code editor is implemented in the `XCSourceCodeTextView` class, a subclass of Cocoa's `NSTextView` class. It works in conjunction with a custom `NSTextStorage` subclass, which is particularly interesting because it holds a reference to a `XCSourceScanner` for the currently edited file.

Different file editors are used to accommodate for different file types.

The delegate of a `XCSourceCodeTextView` is a `XCEditFileEditor`, which allows accessing the cursor position and which provides some convenience methods to query the source scanner and the project index for information about the edited file. The file editor is managed by a `XCFileNavigator` instance. The file navigator chooses an appropriate editor for different files types that are opened, so the file navigator might replace the source file editor with another editor to show, e.g., an image file.

Chapter 5

Navigation Behavior

"Science is a way of trying not to fool yourself."

—Richard Feynman

Before we could start implementing a new tool to improve navigation in source code, we needed to understand the requirements and current work practices of programmers. To acquire this information we conducted a preliminary study.

5.1 Study Design

The study consisted of two parts; it incorporated a contextual inquiry and a questionnaire. In this section, we will explain which participants we chose to participate in the study, how we executed and analyzed the contextual inquiry, and why we complemented the inquiry with a questionnaire.

5.1.1 Participants

Since the Stacksplorer project focuses on the development of a prototype for Xcode and Objective-C (4 – "Prototyping Platform"), eligible participants had to use these tech-

Only Cocoa developers could participate in the study.

nologies. We only accepted participants who had at least 6 months experience with Cocoa. We wanted to avoid observing programmers that spend most of their time looking up documentation that more experienced developers would know by heart.

5.1.2 Contextual Inquiry

Contextual design aims to create systems supporting existing work practices.

In the first part of the study, we observed participants during their work on real programming tasks using a technique called contextual inquiry. This technique originates from the contextual design method [Wixon et al., 1990]. The purpose of this methodology is to create systems that fit into users' existing work practices to improve users' experience and efficiency. At the same time, the methods used to design these systems should be time and cost effective, so they can be used in tight development schedules. Many contextual design techniques, such as contextual inquiries, can be applied independent of each other [Beyer and Holtzblatt, 1999].

A contextual inquiry combines an observation of and a discussion with the user.

During a contextual inquiry, a user is observed in a real work setting. The designer discusses and reflects with the user the forces, requirements, and problems that influence the work process. This helps the designer to understand how tools influence the way users approach their tasks and where the tools support or hinder effective practices. A small sample of representative participants is sufficient to obtain profound results, as for most qualitative evaluations [Nielsen and Landauer, 1993].

Execution

The term "maintenance task" describes coding activities related to existing source code.

In the contextual inquiries we conducted, we observed participants that worked on maintenance tasks. "Maintenance tasks" is an umbrella term for all coding activities, which relate to an existing code base, such as refactoring, bug fixing, adding a feature to, or changing one in an existing product. For the purpose of this work, we adopt the very general definition of refactoring by Fowler *et al.* [1999]:

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."

In contrast to Ludewig and Lichter [2007], we do not imply that a specific process is used to perform these changes.

The inquiries focused on navigation behavior.

All interviews took roughly one hour, which allowed participants to finish one or two tasks. During the study, we focused on participants' navigation behavior. We wanted to understand which relationships in source code are explored and which tools are used. In the interviews, we asked for the motivation for a navigation action (unless it was obvious), so we could understand which information was considered relevant for a task and why. Users were generally asked to think aloud, so the number of interruptions due to clarifying questions could be reduced. Too many interruptions might prevent users from successfully completing tasks that require a high cognitive effort.

All sessions were videotaped for more detailed analysis afterwards.

During the observation we took notes. Additionally, the screen contents were video captured and the discussion with the user was audio recorded to allow a more detailed analysis afterwards. These recordings allow to transcribe for each navigation action from a code segment A to a code segment B the following information: 1) How A and B are related (e.g. "A calls B" or "A writes to a variable that B reads"); 2) which tool was used; and 3) how long it took to find B. The feasibility of this transcription was tested with the recordings of one participant beforehand. Unfortunately, the detailed transcription of a user's navigation is very time consuming. We decided to only produce these transcriptions if the other results obtained from the inquiry were insufficient or unclear.

Hypothesis

Navigation types represent reasons to navigate.

Before running the study, we defined the following navigation types. One navigation type should roughly represent one motivation to perform a navigation action. Most of these types (N2-N6) are actually based on a structural relationship between two segments of source code.

N1: Navigating to a known part of the source code refers to navigation to source code that is saved in a location (in the file system or in the project) that the user knows beforehand.

N2: Navigating the call stack refers to navigation from a part A of the source code to either a part B, so that A is called from B, or so that B is called from A.

N3: Navigating variable access refers to navigation from an occurrence of a variable to other locations in the source code where the same variable is read or written.

N4: Navigating between a poster and recipient of a notification is related to the notification system in Cocoa (4.1.6 – "Notifications").

N5: Navigating between interface and implementation means to navigate from the header file, which contains a class's interface, to the file containing the implementation of the class.

N6: Navigating between objects and delegates refers to the delegation pattern (4.1.6 – "Delegation").

N7: Other navigation is used in the analysis to describe navigation that we did not expect beforehand.

We assumed that users often navigate to source code that is structurally related to the code they investigated before.

We assumed that users could confidently navigate the parts of the source code they were actively working on, hence primarily navigations of type N1 occur. If users had to look up the context of some part of the source code, we assumed that they would navigate to other parts of the code that were structurally related to their starting point. This navigation is represented by N2-N6. If changes are implemented that span across multiple classes, we expected that these classes are also related and hence the navigation between them is similar. Because information about a class is always split across interface and implementation (in Objective-C and many other languages), we assumed to observe a lot of navigation of type N5 throughout the observation, even if the user was not currently exploring structurally related source code. The following hypotheses can be formulated:

- H1: Programmers working on a software maintenance task in Objective-C mostly perform navigation of type N1.
- H2: If they access source code containing contextually relevant information, programmers perform navigation of types N2-N6.
- H3: If they implement changes that span across multiple classes, programmers perform navigation of types N2-N6.

5.1.3 Questionnaire

Available tools may prevent users from doing what they ideally would like to do.

The contextual inquiry has two potential weaknesses: Firstly, only one or two tasks per participant can be observed. Because each task has to be approached differently, chances are that the user is observed working on a task that is not typical for his regular work. Although this is counterbalanced over all observations, we still wanted to get an idea of the differences to each user's usual work. Secondly, as the tool that is used might shape the user's workflow, the actually observed navigation actions may not be equivalent with what the user would ideally like to do.

Participants were asked to fill a questionnaire after the inquiry.

To compensate for these weaknesses, we asked each participant to fill out a questionnaire after the inquiry session. The questionnaire was answered in a web browser and was implemented using Google docs[1]. The full questionnaire can be found in appendix B – "Preliminary Study: Questionnaire".

Information about participants' programming experience was collected.

In Q1-Q11 demographic information about participants was collected and it was tested if the sample of participants is broad enough in terms of programming experience to obtain valid results. To take part in the study, participants had to had at least 0.5 years experience with Objective-C.

[1]http://www.google.com/google-d-s/forms/

Importance of different navigation types was evaluated for different tasks.

In the following questions, we wanted to investigate how important users consider the navigation types listed above. The importance of a navigation type for a task is measured by the frequency with which a navigation of this type is performed (Q12-Q14). In addition to the per task type analysis, in Q15 answers should not be specific to any task, so non-maintenance tasks were also included.

We expected users to rate N1 and N5 most important.

Consistent with the anticipated observation results (H1-H3), we expected users to rate N1 and N5 most important, independent of the task. We assumed that the other navigation types (N2-N4, N6) are rated less important with little differences between them. Because implementing new features is a task that can be approached more isolated than the other tasks, we expected the importance of N2-N4 and N6 to be rated lower for this task.

- H4: For three tasks, "bug fixing", "refactoring", and "adding new features", Objective-C developers rate N1 and N5 the most important navigation types, with no significant difference among the tasks.
- H5: Developers rate the importance of N2-N4 and N6 roughly equal for all tasks.
- H6: The importance of N2-N4 and N6 is lower for "adding new features" tasks, than for "bug fixing" or "refactoring" tasks.

Xcode's support for different navigation types and the usefulness of its tools was assessed.

The last section of the questionnaire was concerned with the tool support provided by Xcode. Firstly, participants could rate how well Xcode supports the different navigation types (Q17) and where they miss support from Xcode (Q18). Next, the importance of tools that are provided by Xcode (4.2.1 – "Navigation Tools") was rated (Q19).

We assumed that Xcode does not support structurally guided navigation satisfactorily.

We expected that answers to this last set of questions showed that the tools provided in Xcode do not fully satisfy users' needs for navigation guidance. This means, in Q17 we anticipated users to rate Xcode's support for navigation types N2-N4 and N6 mediocre or low. Moreover, we assumed users would rate the importance of tools low if these tools are not directly applicable for one of the navigation actions N2-N6 (e.g., "Class Browser", "File History", and "Bookmarks").

- H7: Objective-C developers rate Xcode's support for N2-N4 and N6 3 or worse on average.
- H8: Importance of features not directly applicable for a navigation type N2-N6, is rated 3 or worse on average.

5.2 Results

The study could, generally speaking, confirm that developers using Cocoa and Xcode exhibit similar navigation behavior as Java developers. This indicates that we can, when designing Stacksplorer, build on previous results from studies with Java developers. Furthermore, we could show that developers are not fully satisfied with the tools provided by Xcode. These conclusions are based on our observations regarding developer's work practices and on the results from the questionnaire. Both will be extensively described in this section.

5.2.1 Demographics and Experience

Most participants currently exclusively develop with Cocoa.

Six developers (P1-P6) participated in our study. Five participants were males; the average age was 26.2 ($SD = 1.83$). All participants were computer-science students; two had a finished Bachelor or Master degree (equivalent to a German Diploma). On average, participants had 6.92 years ($SD = 3.26$) experience with programming in general and 1.22 years ($SD = 0.90$) experience with Objective-C. The median of their experience ratings for Objective-C was 2. In Q9, one participant stated he was most experienced with Pascal; another participant rated his experience with PHP highest. Everyone else selected to be most experienced in Objective-C. All participants but one currently exclusively develop for the Mac and/or the iPhone/iPad.

5.2.2 High-level Strategies

Cocoa developers exhibit similar high-level strategies as developers on other platforms.

During the observation, users worked on a variety of maintenance tasks. Three participants added new features to their existing software, two participants performed bug fixing, and one participant worked on a refactoring task. Two participants managed to complete two tasks during the interview session, the other participants worked on the same task all the time. The high level approaches to these tasks were similar to those Ko *et al.* [2006] observed in their work. Two users started with a written plan describing what changes should be implemented and how. One of these two users currently tested this strategy and was not sure if he would adopt it ultimately. Users not using a written plan approached their tasks less structured and started with a longer exploration of the existing source code.

5.2.3 Documentation

Quick access to the documentation was crucial for all developers.

The resource users accessed by far most frequently, besides the parts of the source code they actively worked on, was documentation. Methods to access the documentation were manifold. The most prominent method was using the "Quick Help" tool, which shows the documentation for a single symbol inline. Often the full documentation was used after the "Quick Help" tool, because users felt the documentation shown in the "Quick Help" tool was insufficient, or they were unsure if the method, whose documentation was shown, was appropriate for their task at all. One participant complained that the "Quick Help" tool did not offer a way to open the full documentation (that includes a search feature), if it did not find documentation for a symbol ("If it did not find documentation, I want to search for it manually.", P5).

Some users preferred a web browser to access documentation because of its feature set.

Two participants did not or not always use the built-in features of Xcode to access documentation, but used a browser instead. They explained that they prefer the features their favorite browser offers (for example, tabbed browsing, P2). Users also stated that Google found some documentation more reliably, for example, documentation for APIs not developed by Apple, such as OpenGL.

5.2.4 Source Code Access

Structured source code browsing was preferred over opportunistic browsing for work in unknown projects or by more experienced developers.

If source code was accessed, we observed three basic strategies participants used. The first strategy was to search old source code that was similar to the code users currently worked with. When users encountered a problem, they could often remember if they solved a similar problem before (“I did this some time ago in another project...”, P3). If they implemented the solution recently, users were quite successful finding the relevant parts of their old source code, even if the old source code was located in another project. Users could then copy this old source code to reuse it for their current task. If the source code was too old, users could still remember that they solved the problem a while ago, but they had to search for hints that reminded them of the correct solution in the documentation. The other two strategies could be interpreted as opportunistic and structured browsing, as described by Robillard *et al.* [2004]. Users doing opportunistic navigation frequently scrolled through a file to scan it for probably relevant code fragments. Structured browsing was more frequently performed if the user had less experience with the project he was working on (so the chance to be successful with opportunistic browsing was too low), or if the user was more experienced with Objective-C.

The file browser is the most obvious and most used way to navigate to a known location in the source code.

Analyzing navigation at a lower level, we found that the most used method to access a particular part of the source code was to select the containing file in the file browser. Mostly users knew by heart where they had to navigate to, which supports H1. However, we assume that one important reason we observed that much navigation happening using the file browser is that it is the most obvious way to navigate. Consequently, users did a lot of navigation actions using the file browser, although more effective tools were available. For example, one user always switched between header and implementation using the file browser instead of using the “Switch between header/source file” command, although this was time consuming (especially when the file browser showed a lot of files on a small screen) and error prone (because the user frequently accidentally clicked the wrong file).

The "Jump to definition" tool and the project-wide search were used to reveal contextually relevant code.

If users started to look for contextual information in the source code, they increasingly used more advanced tools, in particular the "Jump to definition" feature and the project-wide search. These tools can help to answer questions like "Where is that accessed?" (P5) or "Is it save to delete this [validation] from this method?" (P4), because they can be used to reveal how a method is used in the context of the complete application. Hence, a frequent use of these tools to access contextual information supports H2.

The project-wide search is more versatile but slower to use than the "Jump to definition" tool.

A good example of how these tools can be used effectively was given by P5, who worked on a project in which she should replace the current user interface with a new one. She analyzed potential call stacks in the current implementation to figure out which responsibilities the UI code had. To jump to the implementation of a method, which was called from the current part of the source code, she used the "Jump to definition" tool. Unfortunately, no similarly convenient way exists to jump to methods that are calling the currently viewed method. Hence, she always did a project-wide search for the selector of the currently viewed method for this purpose. Compared to the "Jump to definition" tool, the project-wide search needs much longer to show results. Additionally, the results have to be further analyzed manually, because selector names can be ambiguous if they are used in different classes. For example, a search for the selector `init`, the default initialize method for an object (similar to Java's constructors), will reveal all initializations of any object in the project, which is mostly not useful. If a method takes more than one argument, regular expressions are required in the search to find all calls to methods with exactly that name. For example, to match calls to a method `initWithFirstName:lastName:` the search query "initWithFirstName:.* lastName:.*" is required. No user used such a query, though. Instead, users only searched for the first part of the method name if a method took more than one argument. In the example above, users would have searched for "initWithFirstName:". This amplifies the ambiguity problem, especially if a class contains a whole cluster of methods taking various amounts of arguments, e.g., `initWithFirstName:lastName:` and `initWithFirstName:lastName:emailAddress:`.

In contrast to the "Jump to definition" tool, the project-wide search shows results in a separate window.

Another important difference between the project-wide search and the "Jump to definition" tool is that the first feature uses a separate editor window to show the results, whereas the second one changes the contents of the editor from where it was used. Some users preferred the project-wide search for its behavior, because it allows to view the method they are working on and contextual information that was revealed by the search side by side. Hence, some users even used the project-wide search to explore the call stack in both directions (i.e., from a method to the called methods, and to methods calling the method). However, not all users doing so really preferred the project-wide search, some were not aware of the "Jump to definition" tool, or they had trouble predicting if this tool would take them to the implementation or to the definition of a symbol. The project-wide search is also the only tool users were aware of that can help exploring relationships of type N4. Some users tried to circumvent both tools by keeping the knowledge about the important parts of the call stack entirely in their heads.

If variable access does not happen through an accessor, it was mostly searched for using opportunistic strategies.

Navigation to locations in the source code where a given variable is accessed (N3) happened mostly by scrolling through the source code in an opportunistic fashion. Some participants used a Xcode feature that allows jumping to methods in the current file directly if they knew where the information they were looking for was located. If the variable is accessed from a different class, the access usually happens through an accessor. As accessors are regular methods, the techniques used to find the relevant information were, in this case, equivalent to those used to explore the call stack.

5.2.5 Importance of Navigation Types

Figure 5.1 shows an overview of users' estimates about the frequency with which they use the different navigation types when working on different tasks. The diagram la-

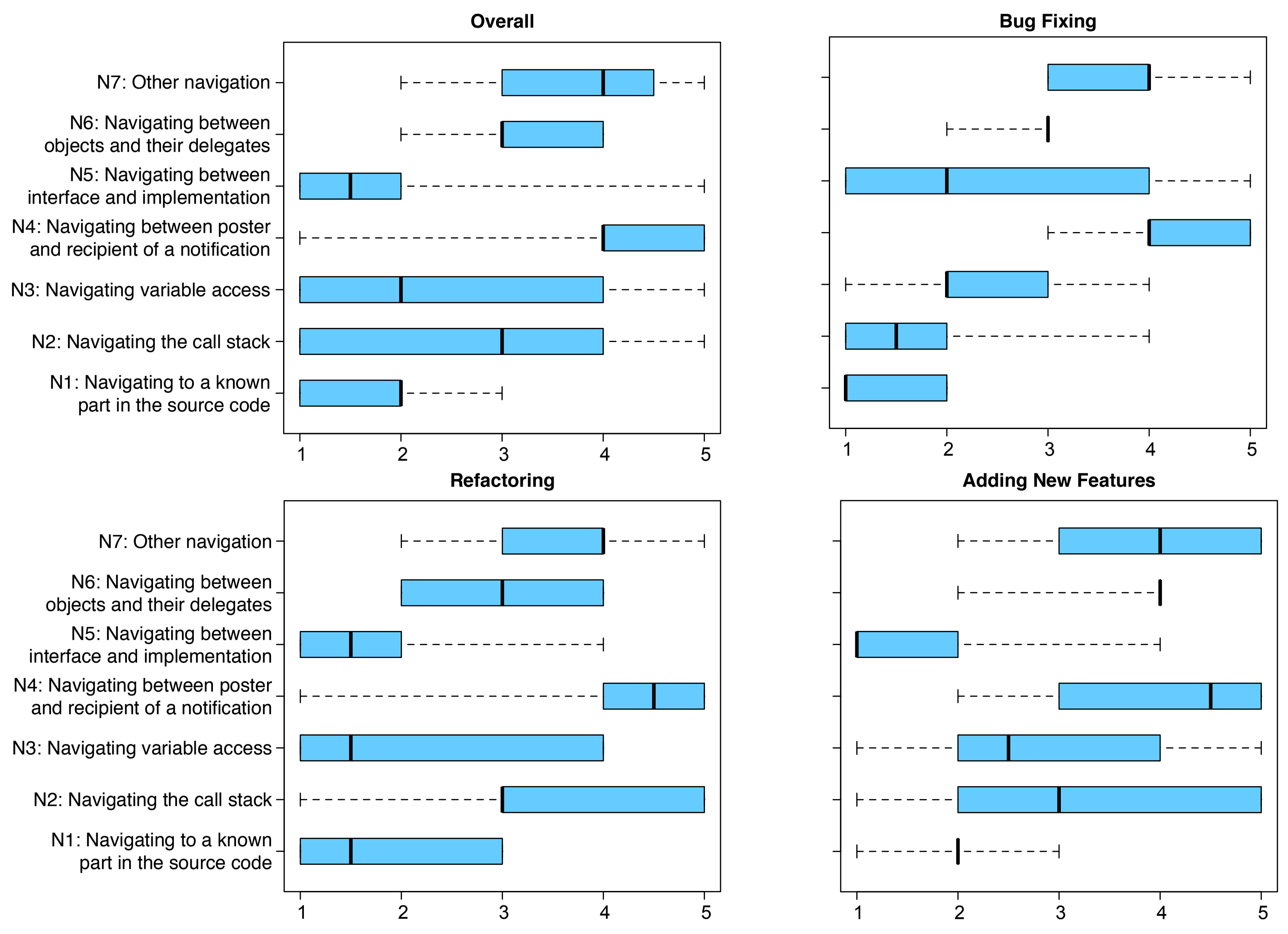


Figure 5.1: Boxplots showing users' decision how frequently they use each navigation type for each kind of task. The diagram labeled "Overall" shows all of users' responses, independent of the task. The rating scale was always labeled with 1=frequently, 4=seldom, 5=never.

beled "Overall" plots all responses users gave, independent of the task.

Switching between header and implementation was considered the most frequently used navigation type.

Overall, navigation of type N5 was rated the most frequently used navigation type. This is consistent with the actually observed behavior, where users primarily used the "Switch to header/source file" feature for this type of navigation. However, only two users were aware of the shortcut for this feature; another one complained, that the button for this feature was inconvenient because it was too small.

Users try to memorize the currently relevant slice of the code.

The second most important navigation type is N1 (N3 has the same median but responses are spread more). For bug fixing, two thirds of the users answered to do this type of navigation "frequently". Considering also the observed behavior, we think users build a good model of the slice of the source code they are currently actively working on and can hence navigate in these parts of the source code mostly by using knowledge they have in their head.

Structurally guided navigation is required if contextual information is accessed.

Navigation types other than N5 and N1 are only necessary if contextual information is required. Hence, their usage frequency is only rated between 2 and 3.5 (one exception being N4, what will be discussed later). H4 and H5 are hence confirmed, which can additionally be backed by the results of Q15: N1 and N5 both had top positions in user's Top 5 rating (N1: $M = 1.67$, $SE = 0.33$ and N5: $M = 3.00$, $SE = 0.55$).

Call stack exploration was considered especially important for debugging.

Besides N1 and N5, only N2 appeared in every user's Top 5 list (rating: $M = 2.50$, $SE = 0.72$). There is a consistently high agreement that N2 is essential for debugging (see Figure 5.1). For other tasks it is rated much less important; for refactoring there is only one rating indicating more than average usage frequency. Because refactoring can be performed using relatively fixed procedures for restructuring [Ludewig and Lichter, 2007], we assume that in depth comprehension of the call stack is not required. Additionally, for tasks other than debugging the ratings for N2 are much wider spread. This indicates that different strategies when working on these kinds of task have a big influence on the frequency with which this navigation type is performed.

Variable access is explored particularly frequently while refactoring.

N3 is used especially frequently while refactoring and less often for other tasks and hence seems similar to N2, which was especially important for bug fixing. Although N3 got worse ratings on users' Top 5 lists than N2, it was considered to be used slightly more often overall. Probably this is because N2 relates to an inter-method only relationship while N3 can also apply to intra-method analysis.

Navigating to delegates was considered less important to add new features.

Navigation of type N6 is performed with an average frequency overall, but less frequently when adding new features. We assume implementing new delegates is a relatively straightforward process, which is very well documented in most cases. So the relationship between an object and its delegate is only explored in more depth if a bug occurs or the structure of the source code is being questioned when working on refactoring tasks.

Notifications are used seldom in Cocoa, so exploration of notification posters and receivers was considered unimportant.

The least used navigation type for all tasks is N4. The reason for this is probably that notifications are not used very frequently in Cocoa and hence participants lacked experience in using them. Two participants did not even know what notifications are, because they never had to use them. Most other participants selected to use this navigation type "seldom" or "never", but commented that it would in fact be more important if notifications were used more frequently. One participant, who was currently working on a system relying a lot on notifications, rated the importance of N4 with 1.

H6 could only be confirmed for N6. Both N2 and N3 were especially important for one task, but similar for the remaining tasks. The results obtained for N4 are too biased by the low experience of participants with notifications to draw valid conclusions.

Access to documentation is seamless enough to not be noticed.

Other navigation is performed with a slightly less then average frequency. With "other navigation", users referred to navigation to the documentation or to a xib file (a file that specifies the user interface of an application and that can be edited in a graphical interface builder). Interestingly, the user relating "other navigation" to navigating the documentation rated the frequency of this navigation task with 3 (for all tasks), although we found that all users looked up documentation very frequently during our observation.

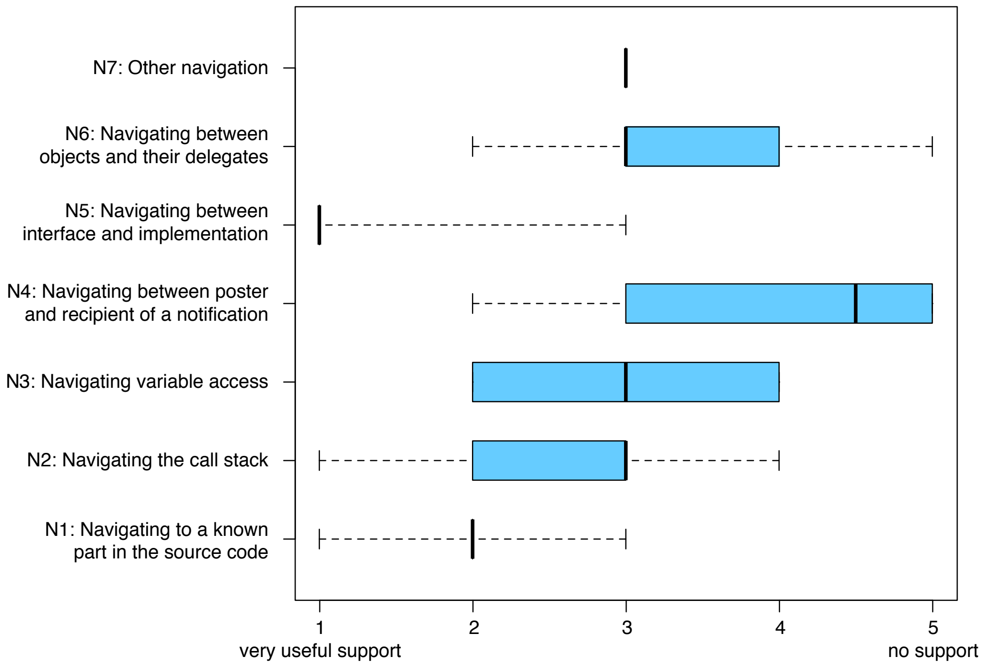


Figure 5.2: The boxplot diagram shows users' rating of Xcode's support for different types of navigation.

This indicates that accessing the documentation is seamless enough for users to not really notice how frequently they do it.

5.2.6 Xcode Tools

Users' rating of Xcode's support for the different types of navigation is shown in Figure 5.2; their rating of Xcode's tools' importance is shown in Figure 5.3.

Xcode supports the most used navigation types best.

Xcode's support was rated best for the most frequently performed navigation tasks N1 and N5. As expected, the tools supporting these navigations are also rated quite essential: For N1 the most important tool is the "File Browser"; to perform N5 efficiently the "Switch to header/source file" tool is crucial.

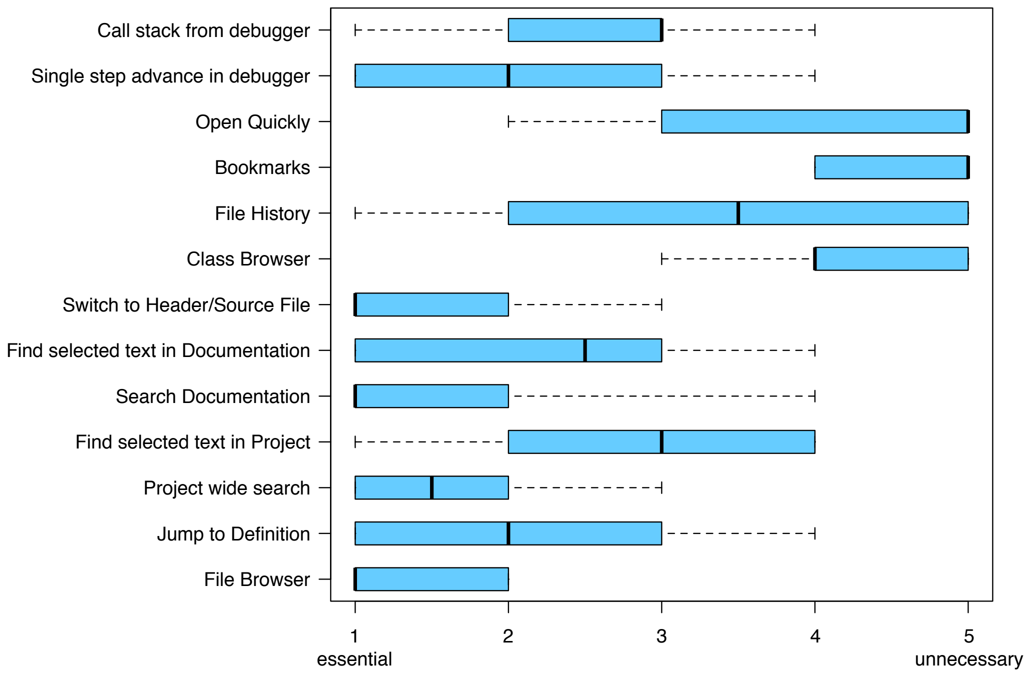


Figure 5.3: The boxplot diagram shows users' rating of importance of different tools in Xcode.

The documentation search was considered crucial.

Not surprisingly, because documentation was the most accessed non-code resource, documentation search was rated similarly important as tools supporting N1 and N5. Access to the same feature from the context menu of a symbol or text was considered less important. The context menu command is harder to find, requires using the mouse, and hence is slower to use.

Xcode's support for structurally guided navigation was rated mediocre or worse.

Support for navigation of type N2, N3, N4, and N6, i.e., for all types reflecting a structural aspect of the source code, was rated with a median of 3 or worse (see Figure 5.2). From these four navigation types, N2 is still supported best, since the "Jump to Definition" tool at least provides a convenient way to navigate the call stack in one direction (from a method to the methods that are called). Unfortunately, no comparably convenient way exists to navigate the call stack in the other direction. This might also explain why the importance of the "Jump to Definition" tool was rated slightly worse than the features supporting N1 and N5.

The project-wide search features were rated similarly to the "Jump to Definition" feature. While the standard search feature was considered nearly essential, the importance of the context menu item was much worse. The project-wide search was most likely considered essential because of its versatility. It can be used to access a multitude of relationships in the source code (5.2.4 – "Source Code Access"); for example, the project-wide search allows, in addition to the "Jump to Definition" feature, exploring the call stack, or exploring where notifications are sent and received.

The project-wide search was considered essential because of its versatility.

Regarding the debugger features, single step advance was considered more important than the ability to explore the call stack of a halted application. The two features differ in the direction of the call stack they offer for exploration: When the application is halted in a method A, the call stack presented in the debugger is a back trace, i.e., a list of methods that were called to finally call A, while single step advancements allow inspecting which methods are called from A. We also assume that users prefer the more interactive exploration the single step advance mechanism offers, because it helps slowly tracking what the application does to generate a particular output.

Single step advance in the debugger was considered more important than the debugger's ability to show a back trace.

The class browser, file history based navigation, the "Open Quickly" tool, and bookmarks were rated much less important than the other tools. Only few participants actually knew these features. This supports hypothesis H8.

Tools not supporting structured code browsing were considered unimportant.

5.2.7 Suggestions for Improvement

In two open questions (Q18, Q21) users had the opportunity to describe their ideas to improve Xcode's support for navigation tasks. Although Q21 did not specifically ask for navigation in source code, most comments still related to this problem domain. Many suggestions were directly related to one of the navigation types explained above and suggested improvements helping to explore structural relationships in source code. This supports our hypothesis that this kind of navigation is in fact important and not satisfactorily supported.

Users wished for more support of structurally guided navigation.

Users' primary focus should not get lost when exploring contextual information.

One user was concerned with the loss of his current focus when exploring related source code, for example, using the "Jump to Definition" tool. Although he was aware of workarounds, such as the possibility to split the editor view, he suggested the addition of tabs for open files. He would like to be able to lock the tabs for files he was actually working on in a fixed position, and use other tabs to explore related code. Tabbed code editors are not uncommon for modern IDEs, e.g., they are available in Microsoft's Visual Studio.

Variable access should be represented graphically.

Two users suggested a visualization of variable access. While one user thought of a graph-like representation, showing the access to a variable chronologically, the other user would already be satisfied if other locations where a given variable is accessed could be highlighted in the source code. This feature actually exists in Xcode. If the cursor is placed within a variable name, other occurrences of this name are underlined with a dashed blue line. This highlighting is too subtle to be noticed for most users, though.

Graph representations should be used to represent the structure of source code.

Improvements in support for navigation of type N2 were also suggested by two users. One user requested a feature that always takes him to the implementation of a method. Using the existing "Jump to Definition" tool the user was unsure if it would take him to the definition of a symbol or to its implementation, although for him the implementation was the more interesting information. The other user suggested a more sophisticated tool that shows the *dynamic object graph* side by side with the source code. He had no strict definition of a dynamic object graph, however it should contain "owns" and "uses" relationships between classes. Hence, it is similar to the relationship we referred to with N2. Some sketches the user showed also incorporated other communication between objects besides method calls, for example, notifications. Occurrences of the delegation pattern were also specially marked. So this kind of dynamic object graph incorporates information related to N2, N3, N4 and N6 in a single graphical representation.

Chapter 6

Software Prototype

"Vision without implementation is hallucination."

—Benjamin Franklin

From previous work and our own preliminary study, we could conclude several implications for the design of a new visualization tool for source code navigation. This chapter will introduce the design and the idea of Stacksplorer, as well as explain how a first prototype was implemented, which could be used to evaluate the design idea in a user test.

6.1 Design

In the preliminary study (5 – "Navigation Behavior"), we learned that users considered the call stack as one of the most important structural relationships in source code. Hence, supporting visualization of and navigation along the call stack is crucial in order to make the system beneficial to developers. The call stack also includes information about access to instance variables, which usually takes place through accessor methods.

The call stack also includes information about variable access, if it happens through accessor methods.

Structured and opportunistic browsing has to be supported.

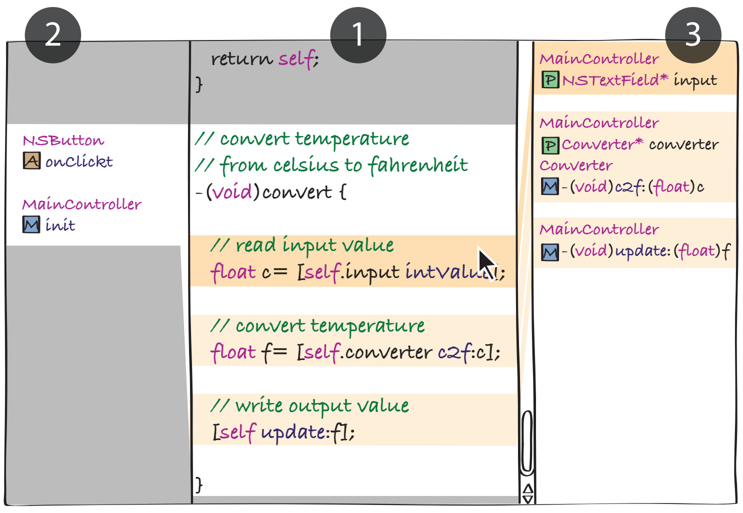
Stacksplorer has to support two different kinds of source code browsing strategies [Robillard et al., 2004]. For users with structured browsing behavior, the system should make the call stack accessible more easily and the navigation through the call stack faster. If opportunistic strategies are applied, providing targeted support is harder. However, we still have to make sure that the displayed information is relevant for the task at hand. This might lead to a “sightseeing” behavior, where users gather knowledge about nearby methods when they pass by while navigating [Storey et al., 2000].

Possible call stacks of an application can be represented as a graph.

Possible call stacks in an application can be represented as a finite, directed graph. Each node in this graph corresponds to one method in the source code. An edge from method A to method B exists in the graph, if method B is called from the implementation of method A.

Stacksplorer shows a focus method’s neighborhood in the call graph.

The idea of Stacksplorer’s design is to show a section of the call stack as contextual information for a *focus method*, i.e., the method the user is currently working on or trying to understand. The context of a focus method is, in Stacksplorer, the neighborhood of the focus method in the call stack graph, i.e., callers of and methods called from the focus method.



```
  return self;
}

// convert temperature
// from celsius to fahrenheit
-(void)convert {

  // read input value
  float c= [self.input intValue];

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

  // write output value
  [self update:f];

}
```

Figure 6.1: Stacksplorer utilizes horizontal navigation to explore potential call stacks of an application.

A paper prototype of our tool is shown in Figure 6.1. The central editor (1) is equivalent to Xcode's standard editor, retaining all its features and functionality. The cursor in this window marks the focus method. The left hand column (2) shows methods *calling* the focus method; the right column (3) shows methods that *are called* by it. The information in both side columns is gathered and updated automatically with no user interaction required at any point.

The side columns show who calls and who is called by the focus method.

This visualization technique could be interpreted as fisheye view [Furnas, 1986] for the call stack graph. The focus method is shown completely, including the implementation. Neighbor methods in the call stack graph are visible at the same time, but in a reduced form, without their implementation. In a more general sense, Herman *et al.* [2000] collated this kind of incremental exploration of a graph with placing a window on top of the graph, so that one *logical frame* is shown at a time. Huang *et al.* [1998] phrased the term *focus node* for the logical frame's central node, which defines which other nodes will belong to the logical frame.

Stacksplorer applies previously known ideas for graph exploration to source code browsing.

Stacksplorer also adds a new degree of freedom for navigation through a project's source code. In addition to navigating through a single class by scrolling vertically in the editor, our design allows navigating horizontally through the call stack graph by clicking a method in one of the side columns. For example, navigating to a method that calls the focus method will cause all 3 columns to shift to the right. The method that was selected moves to the center and opens in the central editor (1), the previous focus method appears in the list of called methods to the right (3), and the left column is updated with new information (2). This can be interpreted as sliding the logical frame over the call stack graph.

Stacksplorer uses the horizontal navigation axis for call stack navigation.

To help understanding why methods appear in one of the side columns, optional graphical overlays are provided, which relate the method call in the source code with the corresponding item in the right column. In the paper prototype (see Figure 6.1), the overlays are shaded yellow and are drawn behind the source code. In the final implementation (see Figure 6.2), overlays are drawn as grey paths, which reassemble visual elements already used in Xcode. A path surrounding all methods in the left column and the

Graphical overlays connect the source code and the entries in the side columns.

focus method's source code indicates that all methods in the left column call the focus method. Because the remaining source code in the central editor is slightly grayed out, the current focus method stands out visually. This affords a method-based navigation in contrast to today's class-based navigation. Of course, the overlays can be hidden if call stack exploration is not the developer's primary task or if the overlays are undesirable for other reasons.

Important paths through the source code can be stored.

Important paths through the code may also be stored for later reference. Firstly, this allows tagging methods while exploring source code to capture knowledge about the source code. Secondly, it can be used by the original developer of the application as a new form of documentation. This documentation can communicate to other developers what purpose a method serves or which features of the application use a method. Storing a method in a path should be possible with minimal effort, because developers are often not willing to put much effort in creating documentation [LaToza et al., 2006].

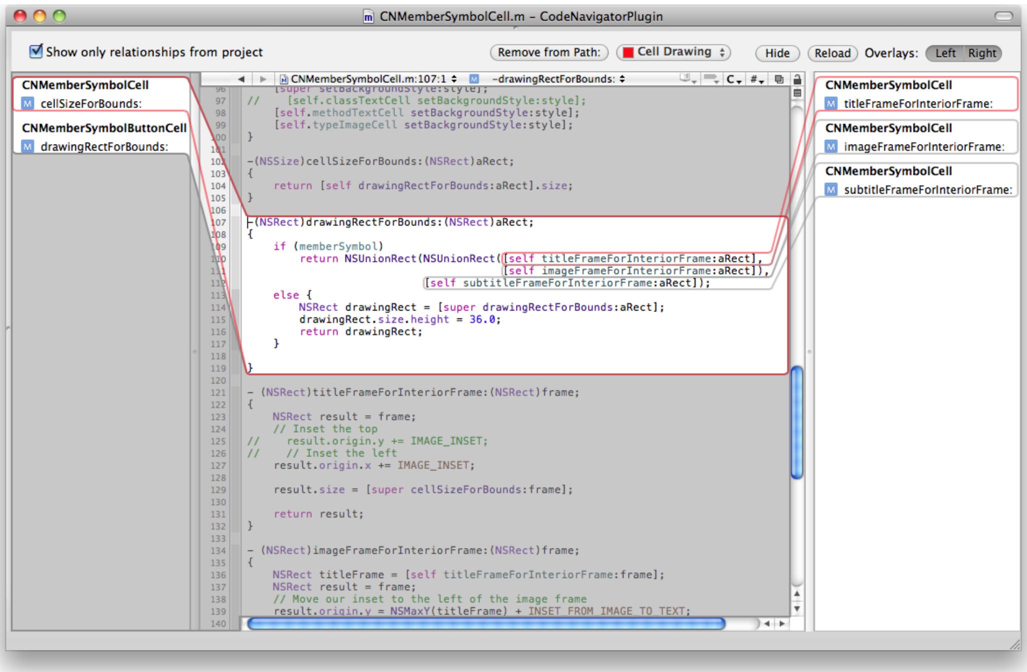
The side columns can be collapsed.

A downside of Stacksplorer's design is that it occupies additional screen space horizontally. Hence, it works best on high-resolution, wide-screen displays. Collapsing the side columns in case they are not needed is possible, to accommodate for smaller screens.

6.2 Stacksplorer Xcode Plug-in

Stacksplorer should be evaluated in a real world setting.

To evaluate the effectiveness of a novel visualization for runtime interactions of objects in an application, it is necessary to build a working prototype that can be tested in a real world scenario. Users should be able to navigate in the source code of an application freely, as they would do using tools that are currently available. Additionally, we wanted to investigate if Stacksplorer works well with users' familiar programming workflows. If the system was easy to use and helped programmers to understand source code more easily but was impractical for them to integrate into their everyday workflows, it would still be a bad design.



```
    [super setBackgroundStyle:style];
//    [self.classTextCell setBackgroundStyle:style];
    [self.methodTextCell setBackgroundStyle:style];
    [self.typeImageCell setBackgroundStyle:style];
}

-(NSSize)cellSizeForBounds:(NSRect)aRect;
{
    return [self drawingRectForBounds:aRect].size;
}

-(NSRect)drawingRectForBounds:(NSRect)aRect;
{
    if (memberSymbol)
        return NSUnionRect(NSUnionRect([self titleFrameForInteriorFrame:aRect],
                                       [self imageFrameForInteriorFrame:aRect]),
                               [self subtitleFrameForInteriorFrame:aRect]);
    else {
        NSRect drawingRect = [super drawingRectForBounds:aRect];
        drawingRect.size.height = 36.0;
        return drawingRect;
    }

}

- (NSRect)titleFrameForInteriorFrame:(NSRect)frame;
{
    NSRect result = frame;
    // Inset the top
//    result.origin.y += IMAGE_INSET;
//    // Inset the left
    result.origin.x += IMAGE_INSET;

    result.size = [super cellSizeForBounds:frame];

    return result;
}

- (NSRect)imageFrameForInteriorFrame:(NSRect)frame;
{
    NSRect titleFrame = [self titleFrameForInteriorFrame:frame];
    NSRect result = frame;
    // Move our inset to the left of the image frame
    result.origin.y = NSMaxY(titleFrame) + INSET_FROM_IMAGE_TO_TEXT;
```

Figure 6.2: Stacksplorer is implemented as a Xcode plug-in. In the screenshot, Xcode's file browser and toolbar are hidden.

Regarding the concluding evaluation, we had to implement the prototype on the same level of detail and using the same technologies as existing implementations to be able to compare the approach with existing ones. In addition, a software prototype is beneficial, because it can be easily distributed to practitioners all over the world to gather broad qualitative feedback. The typical drawback of a software prototype is that it affords too detailed feedback, although in early design stages high-level feedback about concepts is required. This problem may not impede our work as much, since we will evaluate Stacksplorer with programmers. They should have a better idea of the distinction between the concept of a visualization and implementation details. We actually found this hypothesis confirmed later during the user test, since we got a lot of helpful conceptual feedback during the evaluation (7.2.6 – "Users' Comments").

To test Stacksplorer in a real world setting, a software prototype was required.

The prototype was implemented as a plug-in for Xcode (see Figure 6.2). It integrates the navigation technique explained before into Xcode for Objective-C source code. When a

The software prototype was implemented as a Xcode plug-in.

project is loaded, the two additional columns are shown. By default, these columns only show callers and called methods that are implemented inside the project, so calls to methods in other frameworks, e.g., Cocoa, are hidden. For methods in a framework, the source code of the implementation is usually not available anyways. However, the user may decide to include calls to methods for which the source code is not available in the visualization.

Methods can be stored in user defined paths during the exploration.

Users can easily store a path for later reference while they explore source code. They can add a method to a *user defined path*, as these stored paths are called in Stacksplorer, by just pressing a button while the method is the focus method. If the focus method and one of the methods in the side columns belong to the same path, the overlay connecting both is colored in a color specific to the path. In the situation shown in Figure 6.2, the focus method, the caller `cellSizeForBounds:` and the called method `titleFrameForInteriorFrame:` belong to the same path, which is colored red. A separate user defined path editor window (see Figure 6.3) is used to create new paths, name paths, and assign them a color. Additionally, all methods on a path can be reviewed and navigated to immediately. The paths are stored inside Xcode's project file, so they can be shared with others and are compatible to version control systems, such as SVN[1] or GIT[2].

6.3 Implementation

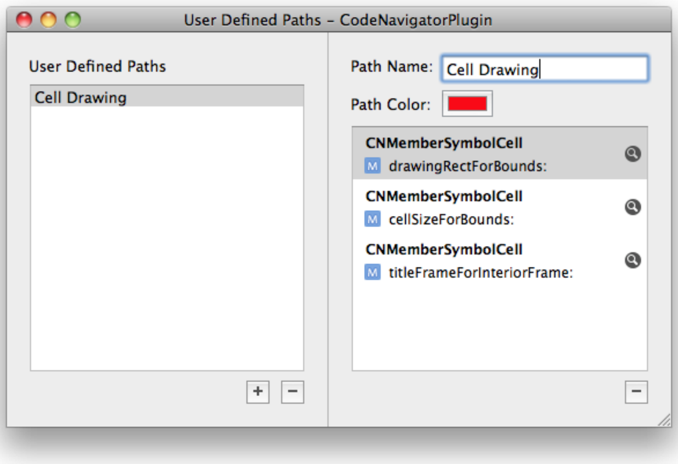
Graphical overlays are rendered in a separate transparent window.

We developed the plug-in using Xcode's plug-in API (4.2.2 – "Plug-in API"). All graphical overlays that Stacksplorer draws on top of the source code are rendered in a separate transparent window, to make sure our changes do not interfere with Xcode's original views. Apple's Core Animation[3] library is used extensively for fluent animations.

[1]http://subversion.apache.org/

[2]http://git-scm.com/

[3]http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/CoreAnimation_guide/Introduction/Introduction.html



Cell Drawing

Path Name: Cell Drawing

Path Color: 🟥

CNMemberSymbolCell
M drawingRectForBounds:

CNMemberSymbolCell
M cellSizeForBounds:

CNMemberSymbolCell
M titleFrameForInteriorFrame:

Figure 6.3: The user defined path editor allows creating and reviewing paths the user stored for later reference.

The source code to realize the visualization is relatively straightforward to implement for an experienced Cocoa developer, but the algorithms to extract the incoming and outgoing edges for a given focus method are more interesting. Hence, we will now give an overview about both algorithms and how they utilize Xcode's integrated code parsing features.

6.3.1 Callers

Xcode does not provide a method to immediately reveal callers of a method (neither for users nor for plug-in developers). In our preliminary study, we learned that users commonly search for the method name using a project wide search to navigate to callers of a method. We mimic this technique with our algorithm.

To determine callers of a method, Stacksplorer mimics what users would do.

Firstly, we construct an appropriate search query from the focus method's selector by adding a wildcard regular expression "`.*`" after every colon. The result is a regular expression that matches every occurrence of the method name regardless of the arguments. We then set

To search for a call to a method, wildcard regular expressions are inserted into the method name after every colon.

up a `PBXRegexBatchFinder` (4.2.2 – "Project Search") to search for the regular expression in the background. This search finds all occurrences of a method call to a method with the given name, regardless of the arguments.

The code completion is used to determine the type of an expression on which a method is called.

Unfortunately, method names are ambiguous and a method with the same name as the focus method may be implemented in another class. Hence, we have to check for each result of the search if the method is really called on an instance of the class containing the focus method. For that purpose we use Xcode's code completion engine (4.2.2 – "Code Completion Engine"). We query it for the list of suggestions that the code completion would present if the user invoked it at the cursor position immediately before the result we found. If the list of suggestions contains a `PBXMethodSymbol` equal to the focus method's symbol, we consider the search result as a call to our focus method.

6.3.2 Called Methods

To find method calls, the source code is pared using Xcode's source scanner.

To find methods called from the focus method's implementation, we have to parse the method's source code in order to find all method calls in it. To parse the source code, Xcode's `XCSourceScanner` (4.2.2 – "Source Scanner") is used. Unfortunately, the source scanner does not parse selectors from a method call into a single `XCSourceScannerItem`. Hence, problems arise because method calls can be nested, arguments can be as complicated as a block, which is again an arbitrary piece of source code, and methods are called implicitly when Objective-C's dot operator (4.1.3 – "Declared Properties") is used. In the latter case, we have to distinguish wether the getter or the setter method is called.

The code completion is used to find out to which class a called method belongs.

Once we found the syntactic features that indicate a method call, we have to find out in which class the called method is implemented. This can again be figured out by using the code completion engine. We invoke the code completion before the method call and filter the results for the `PBXMethodSymbol` with the selector that is actually used in the source code. The `PBXMethodSymbol`'s reference to the method's implementation can be used to navigate to the method.

6.4 Limitations

It is undecidable if a given call stack may be executed during a real program execution.

To accurately determine all call stacks that may be executed during real program runs is nearly impossible. In general, the question if a given path is reachable at runtime is undecidable [Lewis and Papadimitriou, 1981]. Techniques to obtain approximations of the paths used during program executions can be split in two categories.

Running an application and logging all method calls is called *tracing*.

The first category describes techniques that involve running the application. During a single execution of the application all method calls can be logged. This process is also called *tracing* the application. If the application is run multiple times with different input data, more and eventually different paths are used. To generate test cases for software testing, software engineering research has developed techniques to generate input data for applications to test as many different call stacks as possible. As these tests require a lot of effort to be set up and run, they are not suitable for our plug-in. Furthermore, it is impossible to decide if all call stacks that might occur during program execution are actually covered by a given set of tests, since it is not decidable if a given path is executed with any input data.

Static code analysis tools are able to detect some defects in source code without running the application.

Secondly, techniques exist that analyze the application's source code statically, without running the application. The term *static code analysis* is, today, typically associated with analysis performed by automated tools. These tools work similar to a compiler in that they apply dataflow analysis to detect problems such as access to a previously deallocated object. The algorithms used are often, e.g., in FindBugs (for Java) [Ayewah et al., 2008] or clang[4] (for Objective-C), restricted to interprocedural analysis, and hence are not suitable for call stack, i.e., intraprocedural, analysis. To impede that too many false positives (because testing is usually done to find defects in software, a positive result in this context is a defect) are found, static code analysis tools

[4]http://clang-analyzer.llvm.org/

often focus on defects that are easy to detect. Static code analyzers do not claim to find or investigate all possible paths through the source code. For our purpose, however, accidentally found method calls are not dramatically bad; to the contrary, calls we do not find may cause the programmer to miss an important piece of information.

Stacksplorer caches call stack information for the last visited focus method.

Another limitation of our prototype is sub-optimal performance. Only call stack information for the last visited method is cached to improve performance when the user is navigating back and forth between two methods, which was shown to be common [Ko et al., 2006]. Also, when performing navigation along the call stack, the old focus method is shown in the respective side column immediately to make the horizontal navigation easy to grasp. More advanced caching is not performed, so in most cases the call stack information has to be gathered on the fly after the focus method changed. However, we found that this process works sufficiently fast on recent hardware (2.8GHz Core 2 Duo, 4GB RAM). Hence, we decided not to apply further performance tweaks, because the current implementation should work good enough to obtain profound results from a user test. This was also confirmed by three beta testers, who opted in to test the plug-in during their daily work before the user test. Thanks to the beta test, we found and fixed a lot of annoying bugs before the evaluation.

Chapter 7

Evaluation

"In theory, there is no difference between theory and practice. But, in practice, there is."

—Jan van de Snepscheut

To test how Stacksplorer works for practitioners, we ran a user test in which we compared Xcode with the plug-in installed to a default Xcode installation. In this user test we wanted to explore five hypotheses.

H1 Given a time-constrained task that requires browsing and understanding previously unknown source code, more programmers can solve this task correctly using Stacksplorer than using a default Xcode installation.

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

H3 Using Stacksplorer, programmers can analyze side effects of changes more quickly than using a default Xcode installation.

H4 Programmers (subjectively) find that Stacksplorer helps them understanding previously unknown source code.

H5 Programmers (subjectively) find that Stacksplorer helps them knowing where they are in the source code.

In this chapter, we will introduce the methodology and the results of the study we conducted to test these hypotheses.

7.1 Experimental Setup

Hypothesis H1-H3 can be tested by performing quantitative measurements. Supporting H4 and H5 requires qualitative methods, such as a questionnaire and observation of users working on tasks. In this section we present in detail the setup of the experiment we conducted.

7.1.1 Participants

Participants were students experienced in Cocoa development.

For the study, we recruited graduate and undergraduate students, who at least had basic experience with Objective-C. By hiring students, we could reduce the impact of different levels of programming expertise on the study. In contrast to professional developers, experience levels among students are less varying [Bragdon et al., 2010]. However, to be sure that students do not behave dramatically different than professional software developers, we also recruited two professionals.

7.1.2 Conditions and Tasks

Participants had to understand source code to answer a question.

The goal in developing the evaluation of the Stacksplorer plug-in was to test H1-H3 with quantitative methods. We did not want users to spend too much time on actually writing code, because Stacksplorer is not designed to support this activity specifically. Instead, the tasks should focus as much as possible on inter-method navigation and require a thorough understanding of the order in which methods are called at runtime. In the end, we chose tasks that required users to read and understand the source code in order to answer a specific question. Questions were either asking for a location in which a simple change could be implemented or for side effects a particular change in the source code would have. Because Bragdon *et al.* [2010] pointed out that users are often curious about how precise such questions should be answered, we made sure that each question could be answered with a single method name, class name, or by pointing out a specific UI element. Pilot tests confirmed that users were very confident how thorough their answer should be for the questions we developed.

A within-groups study design should compensate for huge individual differences between programmers.

Curtis [1981] pointed out that huge individual differences in performance between programmers exist. Hence, we decided to do a within-groups study design to make sure we obtain comparable pairs of performance measurements. Each participant had to be tested in two conditions: Once working with Stacksplorer and once working without it. So, two tasks, one per condition, were required.

Tasks concerned the source code of the open source application BibDesk.

The tasks users had to solve are listed in appendix C – "User Test: Task Descriptions". All tasks concerned the source code of BibDesk[1], an open-source BibTeX bibliography manager for Mac OS X. BibDesk comprises 88000 SLOC in roughly 400 classes. We used the source code from the BibDesk SVN repository in revision 17029. To make task 2.2 more interesting, we changed the source code slightly: The categories implemented on `BibItem` and `BibDocument`, in which the finding algorithms used by the "Search For" command are implemented, were moved to separate files. Previously, they were all implemented in the implementation file for the "Search For" command.

We asked for appropriate locations for changes or side effects of changes.

To equal out differences in task difficulties, the two tasks we used each consist of two subtasks. For the first subtask of both tasks, users had to search for the appropriate location of a change. The second subtask always included an analysis of side effects of a change. Tasks were given to participants one subtask at a time in printed form. Users were allowed to work up to 25 minutes on the first subtask and 15 minutes on the second one.

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

Users were provided with a hint where to start.

For each task, we provided users with a hint, to simulate knowledge about the very high level structure of the source code. This should reduce the time users spend searching for a starting point, which helped us keeping the total time for the user test below two hours. Providing a starting point of some kind has also proven to be feasible in other studies [Bragdon et al., 2010; de Alwis et al., 2007]. Task order and condition-to-task assignment were counterbalanced to compensate for learning effects, which inevitably occur when working on a previously unknown code base for two hours. To further accommodate for these learning effects, users were given 10 minutes at the beginning of the study to familiarize with the project and its organizational structure.

7.1.3 Methodology

All of Xcode's tools could be used except for the debugger.

During the study, participants were not allowed to use any additional tools that analyze a running instance of BibDesk. For example, debugging or compiling the application (with inserted trace statements) was not allowed. These additional tools might have confounded the results, since we wanted to measure the efficiency of Stacksplorer for code understanding from reading it, and not the effectiveness of other tools. This choice is also consistent with previous studies [Bragdon et al., 2010; Robillard et al., 2004]. Of course, this choice limits the validity of the study, since in real world scenarios these tools would be available. All other tools in Xcode were available to participants, although we did not put any effort in explaining them, since we assumed that participants were sufficiently familiar with Xcode and we did not want to influence their current work practices.

Stacksplorer was explained before the study.

We introduced users to Stacksplorer before the study using a small sample project. In particular, we explained which information is shown in the side columns, how Stacksplorer allows navigating through the source code, how the overlays can be turned on and off, and how the "User defined paths" feature works.

During the study, we asked participants to think aloud, so we could get insights about their mental model while working. However, they were not allowed to ask questions to the experimenter regarding the BibDesk source code. To accommodate for different levels of knowledge about Cocoa, participants were allowed to ask questions about Cocoa. Participants should not be hindered by missing expert knowledge regarding Cocoa at any point.

Participants were asked to think aloud.

The dependent variables we measured were correctness of the given answers (to check H1) and time required to complete the tasks (to check H2 and H3). The screen contents and audio were recorded using Screenflow[2] to allow further analysis afterwards if necessary.

Correctness of solutions and task completion time were measured.

The Stacksplorer prototype was not optimized with regard to performance due to development time constraints. To minimize the impact of this limitation on the results of the study, the tests were performed on a fairly powerful computer (Mac Pro, 2.8GHz Intel Quad-Core processor, 2GB RAM). Participants used a 23″ screen with a 1920x1680 resolution, which is common for a modern work place for programming.

All participants worked with the same hardware.

7.1.4 Postsession Questionnaire

After participants worked in both conditions, we also wanted to find out their subjective opinion about Stacksplorer. To measure their satisfaction with the prototype, we used the System Usability Scale (SUS) [Brooke, 1996]. The SUS consists of 10 statements, for which participants express their level of agreement using a 5 point Likert scale. It yields a single value between 0 and 100 where higher values correspond to users being more satisfied with the tested system. Each rating contributes a value between 0 and 4 to the result. Individual contributions are then summed and scaled to be in the range 0-100 (using a factor of 2.5). The statements are formulated alternately positive and negative.

To measure users' satisfaction with Stacksplorer, the System Usability Scale was used.

[2]http://www.telestream.net/screen-flow/overview.htm

The SUS is a widely known and thoroughly tested metric.

The SUS was initially intended to be a “quick and dirty” measurement. Nevertheless, analysis of finished experiments using the SUS indicated that it yields very reliable results (Cronbach’s alpha = 0.91) [Bangor et al., 2008]. Although the SUS was initially designed to be a unidimensional scale, a factor analysis done by Lewis and Sauro [2009] revealed two independent factors, which are measured by the SUS: Learnability (aligned with statements 4 and 10) and usability (aligned with all remaining statements). The same analysis could also show that the scales for both factors meet common reliability requirements. However, this multi-dimensional analysis of SUS should be considered carefully, because other studies, e.g., [Bangor et al., 2008], could not confirm the existence of two individual factors.

Six additional questions were added to the SUS to address specifically Stacksplorer’s impact on programming.

For the post-session questionnaire, we extended the set of SUS statements with 6 more statements that address aspects which are specifically interesting for Stacksplorer. We wanted to find out how well users feel supported with code understanding (H4) and navigation (subjective measure of H2), and if Stacksplorer helped them not to feel lost in the project (H5). Each of these aspects was addressed with two statements, one positive, and one negative one (to be consistent in style with the statements from the SUS). Additionally, one of both statements for each aspect was formulated as a comparison to Xcode without Stacksplorer. These additional statements had to be evaluated isolated from the regular SUS test, of course.

The full post-study questionnaire can be found in appendix D – “User Test: Post Session Questionnaire”.

7.2 Results

We generally found that Stacksplorer was well adopted by participants. They were not only objectively able to complete tasks faster when using Stacksplorer, buy they were also subjectively highly satisfied with Stacksplorer. In this chapter, we explain in detail the results of the study and additional observations we made.

7.2.1 Participants

In the study, we tested 16 participants, which were all male (although this was not intended by design). Apart from two professional software developers, all participants were students; six of them were graduates. All students were majoring in computer science. On average, participants had worked with Objective-C and Xcode for 2.25 years ($SD = 1.97$) and spent an average of 13.1 hours per week ($SD = 13.0$) on programming. Ten participants were familiar with BibDesk, but none of them had seen the source code before.

All participants had thorough experience with Cocoa.

7.2.2 Task Success

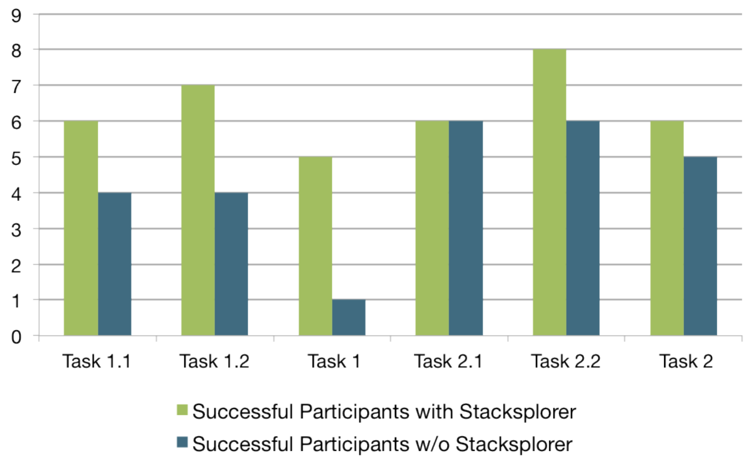


Figure 7.1: The figure shows how many participants were able to complete the tasks in each condition.

Figure 7.1 shows how many participants were able to complete each individual task and subtask successfully depending on the condition. A task was considered to be solved successfully if both subtasks were solved correctly. Only four participants were able to solve both tasks.

Only four participants could solve both tasks successfully.

The diagram shows that all tasks but task 2.1 were solved correctly more often by participants using Stacksplorer. However, a Fisher's test comparing the number of correct

Only for task 1, the difference in number of correct solutions is significant.

solutions for both conditions could only show significance for Task 1 ($p = 0.041$). Consequently, we cannot generally confirm H1.

Task 2.2 was likely the easiest task.

We can only hypothesize why task 2 produced less significant results. We assume two factors influenced the results: Firstly, in task 2.1 participants could more easily than in task 1.1 utilize knowledge about Cocoa, because it was concerned with document saving, which uses a standardized Cocoa API. Secondly, task 2.2 was probably the easiest of all tasks, because the code related to this task was better isolated than in the other tasks. The three methods that participants had to inspect to find the appropriate location for the change were distributed among two class categories and one class, which contained only four methods in total. Another indication that task 2.2 was the easiest task is the fact that, independent of the condition, it was completed successfully most often.

7.2.3 Task Completion Times

Overall, tasks could be solved significantly faster using Stacksplorer.

Task completion times for task 1 and 2 were normally distributed (tested using a Shapiro-Wilk test, for task 1: $W = 0.93$, $p = 0.21$, for task 2: $W = 0.94$, $p = 0.41$). Hence, they can be compared using a paired t-test to show that there is no significant difference in completion time between the two tasks ($t(15) = 1.13$, $p = 0.27$, $d = 0.28$). This indicates a fair comparison between the task solved with and without Stacksplorer is possible. In this comparison we found that participants could solve the task in which they were allowed to use Stacksplorer significantly faster ($t(15) = -1.91$, $p = 0.038$, $d = 0.48$, one-tailed) than the other task. This result supports H2.

Among successful participants, task completion times only differ significantly for task 2.2.

When analyzing each task separately and considering only the measurements from participants who were able to solve the task correctly (see Figure 7.2), no significant difference in task completion times between conditions can be found in any task except 2.2 (Welch's t-test, $t(10.6) = -3.35$, $p = 0.003$, $d = 1.68$). Hence, we can confirm H3 for task 2.2 but not for task 1.2. However, many participants not using Stacksplorer could not solve task 1.2 at all within the given

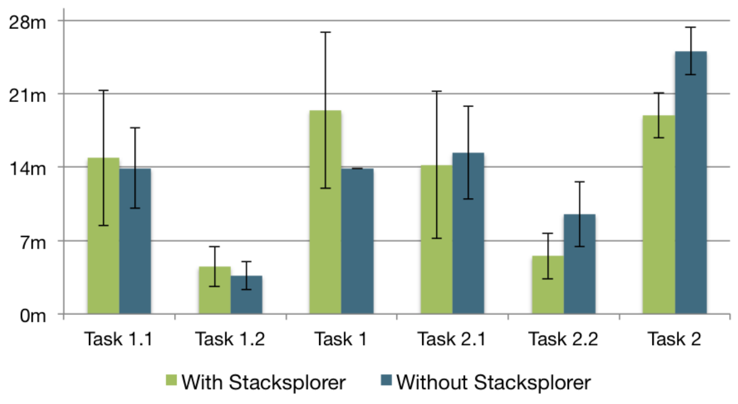


Figure 7.2: The figure shows the average time required to solve the different tasks, comparing measurements with and without Stacksplorer and considering only measurements of successful participants.

time limit. If they were given additional time to complete the task, results may have been significant for task 1.2 as well.

More differences in task completion times may be significant, if all participants had had enough time to complete the tasks.

We assume that we observed the non-significant difference in task completion times for most tasks, because most tasks have not been solved by all participants. Those who did solve them were generally more trained developers anyways and hence knew how to cope with the existing tools in Xcode better. For example, in Figure 7.2, it may appear surprising that, on average, participants not using Stacksplorer could solve task 1 faster. However, the average task completion time for participants not using Stacksplorer in task 1 is calculated from one single measurement (see Figure 7.1). Hence, a valid comparison to the average task completion time of users using Stacksplorer is not possible at all.

Stacksplorer does not provide guidance to find a place to start.

Additionally, we could observe during the tests that participants typically solved the tasks in two steps: Firstly, they searched using different techniques for a point to start, then they followed the call stack to find the place to implement the required change (this will be explained in more detail in section 7.2.4 – "Qualitative Observations"). Since Stacksplorer does not provide specific guidance for the first phase, experience and luck — especially if opportunistic strategies (2.2 – "Programmers' Work Practices") are applied — had a strong influence here. Some participants even went through these two phases multiple times until they found the correct location for an edit. We think this caused the wide spread in measurements, which in the end lead to non-significant results.

7.2.4 Qualitative Observations

Initial Exploration

In the initial exploration, users started either at a model, a controller, or a view class.

Although this was not the primary scope of the study, it was interesting to see how different participants approached the unknown project during the 10 minutes they were given before the first task to familiarize with the source code and its structure. After skimming the folder names in the file browser, their exploration started at either a model, a view, or a controller class (2.1 – "Object-oriented Software Development").

Model classes were too extensive to provide an overview of the application's capabilities.

Half of all participants started out exploring the model classes. Either they were simply overwhelmed by the length of the important classes, or they read at least all method names to get an idea of the capabilities of the classes. Only very few participants took more time to understand how the classes work and how they are connected. The idea of exploring a model class first is to understand what the program is working with and hence what the application can possibly do with the data.

The application delegate is a good place to start, since it is a part of nearly every Cocoa application.

Five participants started at a controller class, mostly the `AppDelegate`. The `AppDelegate` is the delegate of the `NSApplication` class, which, in Cocoa, always represents the running application. Controllers implement all features of the application, so participants learned how the features are split among different controllers by exploring them.

Only three participants started out from the user interface to understand which processes are triggered by certain important UI elements. These participants usually started at a xib file, which represents how various widgets are arranged in the user interface (i.e., in a window), and not at a view class. View classes are only implemented if a particular UI need cannot be satisfied using existing widgets from the toolkit. Also, these custom view classes mostly do not give away much information about the feature or functionality they enable.

Starting from the user interface, users explored how specific features are implemented.

Without a clear goal, it was impossible to determine which parts of the applications are relevant. Hence, after this starting point most participants started to navigate through the source code randomly. Some of them looked for aspects in the source code they found personally interesting. For example, one participant was currently writing an own parser for BibTeX, so he spent some time inspecting BibDesk's BibTeX parser.

Without a clear goal, exploration was mostly opportunistic.

Surprisingly, what most participants did not look at was the included documentation and test cases. The included documentation was a short text file named "Hacking BibDesk", containing an explanation where to start when trying to modify certain aspects in BibDesk. Only two participants had a look at this file. Additionally, all but one participant ignored the included unit tests, although previous studies [Kiel, 2009] found out that they can serve as valuable sample code and intuitive documentation about a class's designated behavior and use.

Included documentation was mostly ignored.

Two-phase Navigation

As explained before, for task 1.1 and 2.1 participants usually started with an exploration phase, in which they searched for an *anchor point*. From this anchor point, they traversed the call stack until they either found the correct location for a change or they noticed that they got lost and had to start again with a new exploration phase. During the second phase, participants often tested an outgoing path and came back to the previously viewed method or to the anchor point if they decided to discard the path. This simple model is depicted in Figure 7.3.

Users explored the source code in two distinct phases.

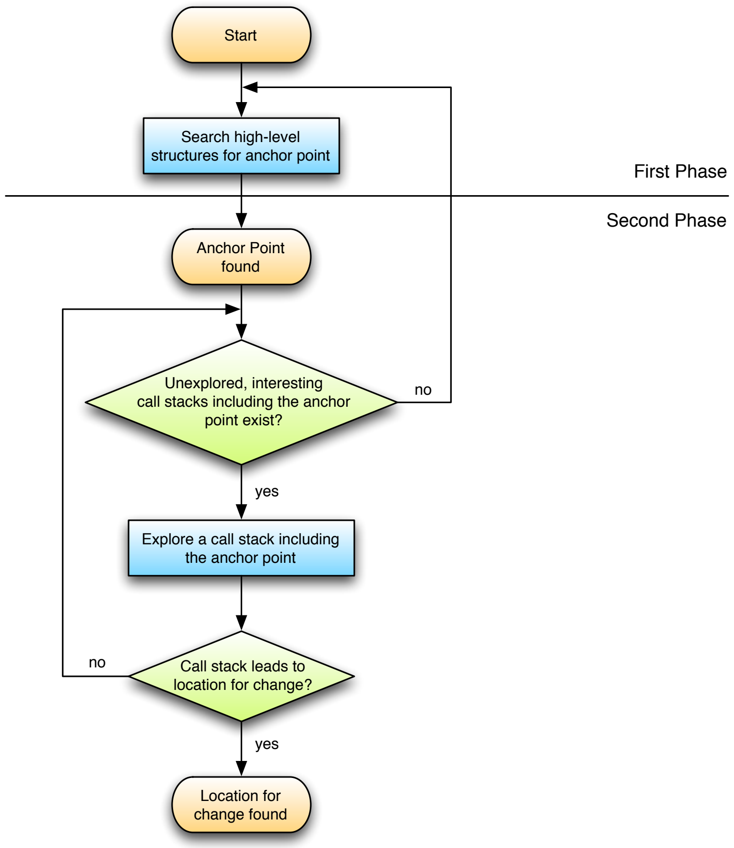

Start
Search high-level structures for anchor point
First Phase
Second Phase
Anchor Point found
Unexplored, interesting call stacks including the anchor point exist?
no
yes
Explore a call stack including the anchor point
Call stack leads to location for change?
no
yes
Location for change found

Figure 7.3: A simplistic two-phase model for a typical developer's strategy when searching for the appropriate location for a change in unfamiliar source code.

In the first phase users primarily utilized the project-wide search.

Although we tried to shorten the explorative phase by providing hints for each task, all users started with some exploration phase, whose length varied a lot, especially for task 1.1. For this task, a multitude of possible starting points exists, and it depends on individual preference which one a participant used. During the exploration phase, different techniques were used. The most prominent technique was performing a project-wide search for "autofile" in task 1.1 or "writeToURL:" in task 2.1. Some users managed to find a correct starting point in the file browser using the provided hint. Another popular technique to start was to find a user interface related to the task and to look up which methods were called by the controls in the interface. Stacksplorer was not very helpful during the initial exploration phase, as it does not provide a high-level overview of the project and does not facilitate searching or similar opportunistic approaches.

In task 1.2 and 2.2, the starting point for participants was clearly given, so the explorative phase was much shorter and had much less influence on success and completion time for these tasks.

Stacksplorer Adoption

Stacksplorer's right column can be interpreted as a summary of the focus method.

Stacksplorer was a welcome addition for the explorative browsing phase in tasks 1.1 and 2.1 as well as in the first part of task 2.2, in which participants navigated along possible call stacks until they eventually reached the location they decided to change. We could clearly differentiate two techniques participants employed to utilize the plug-in. Most participants read the source code and tried to understand (in varying level of detail) what it does. When they were at least somehow sure which part of the source code was relevant to the task at hand, they enabled Stacksplorer's overlays to see which methods were called from this part and then navigated there. Another group of participants used the methods presented by Stacksplorer in the right column as a summary of the method. In the extreme case, they did not read the source code at all; instead, they only browsed through the called methods and navigated to whatever they found interesting. Once they could no longer find such a method, they started reading the method's source code, to decide if they had arrived at the correct location for the requested change. The latter technique is of course much more prone to error, but it can be very fast. The two different techniques to make use from Stacksplorer could represent the two code browsing strategies (structured and opportunistic browsing) introduced before (2.2 – "Programmers' Work Practices"). If participants applied opportunistic strategies and used Stacksplorer, they had an increased chance to stumble upon relevant information by accident. For example, during task 1.2 participant 5 was to discard a relevant method before he accidentally saw a method name in the left column that he thought could be interesting. This accidental discovery led to his success in solving the task.

Stacksplorer usage leads to more forward and backward navigation in the history of visited methods.

When using Stacksplorer, users started to also use Xcode's forward and backward navigation buttons much more than before. In the preliminary study, these were only used in very rare occasions. Since Stacksplorer made it easy to peek into a call stack involving the current focus method, participants were more tempted to explore a path to see where it brought them and to discard it if they found themselves getting stuck. At this point, they would use the backward button to navigate back to a previous anchor point to start exploring another path from there.

Analyzing side effects of a change was easy using Stacksplorer.

For the tasks that required analyzing side effects (1.2 and the second part of 2.2), Stacksplorer was highly appreciated by all participants. It was faster and more robust against errors in comparison to the otherwise required project-wide search for occurrences of method names, because no problems regarding ambiguous selector names (5.2.4 – "Source Code Access") occurred. Participants using Stacksplorer in the first task frequently joked if they could get the plug-in back when working on the second task ("Can I start the plug-in for that [task 2.2] again?", participant 8).

Navigating back in the call stack more than once was difficult for some participants when using the project-wide search.

Another interesting observation from tasks 1.2 and 2.2 is that users tended to go back only one step. For example, in task 2.2 the method containing the change is only called once. Side effects exist only because this single caller of the changed method is used in a different context than the one described in the task. When using Stacksplorer, participants navigated back once from the starting point and then immediately saw the relevant information in the left side column. However, when users had to perform a search, they often hesitated to do this multiple times to explore different levels of the call stack. Some even started guessing when their first search did not reveal any side effects, because they had no idea how to continue. We assume that users are frightened to get lost when they use the search from a fixed starting point and not in an explorative manner.

User Defined Paths Usage

Only one fourth of participants used user defined paths.

The "user defined path" feature was only used by roughly 25% of all users. Those who used it primarily used it to maintain a list of methods they already visited and understood. When it comes to adding methods to a path, two different types of use can be differentiated: Some users added every method they saw, so they would not spend time reading it again later. Others only added methods they thoroughly understood and considered important, so their paths contained isolated program slices (2.2 – "Programmers' Work Practices"). Mostly, participants created one path per task. Some users also used the path editor as a navigation tool to jump back to anchor points. Participant 4 even rearranged the windows so he could see the path editor and Xcode's main window side by side.

7.2.5 Postsession Questionnaire

SUS

Stacksplorer's usability was rated "excellent".

The post-session questionnaire was comprised of 16 statements for which participants had to rate their level of agreement on a 5-point Likert scale. Ten of these statements belonged to the SUS and should not be evaluated individually [Brooke, 1996]. The combined SUS score for Stacksplorer was 85.4 on average ($SD = 7.4$). Bangor *et al.* [2008] presented an interpretation of SUS scores, where starting from 85 products can be considered "excellent". Considering that the tested version of Stacksplorer was clearly a research prototype, with issues in performance and some minor bugs, this result is very positive.

The factor learnability was rated slightly better than usability.

Using the factor analysis by Lewis and Sauro [2009], the results can be analyzed further to obtain separate scores for learnability and usability. The learnability rating ($M = 94.1$, $SD = 10.0$) was much higher than the usability rating ($M = 83.3$, $SD = 7.8$). Partially, this seems to be an effect inherent to the SUS measurement [Lewis and Sauro, 2009]. However, it also indicates that participants understood the concept of our visualization intuitively. The fact that the usability rating is lower (although it is still a very good result on Brooke's interpreted scale) is most likely at least partially due to the performance issues of our prototype.

Non-SUS questions

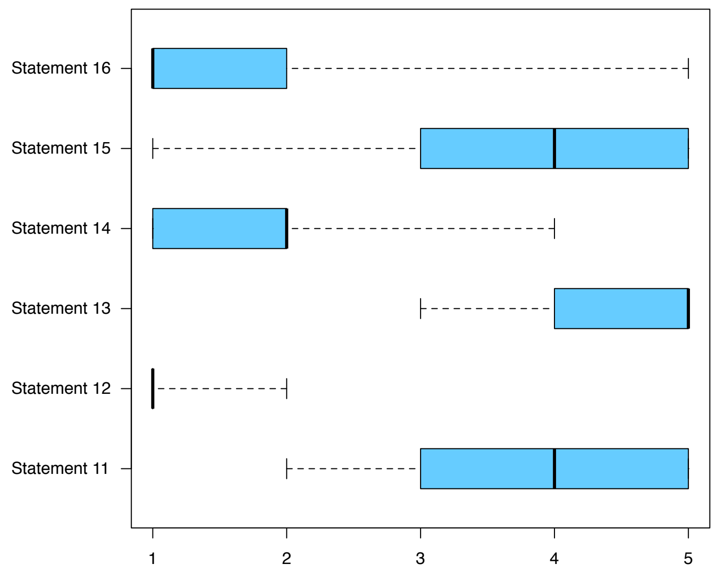


Figure 7.4: The boxplot diagram shows participants agreement to statements 11-16 from the post-session questionnaire on a five point Likert scale.

Agreement to the statements not part of the original SUS has to be evaluated separately.

Participants' agreement to the six statements we added to the SUS specifically for Stacksplorer has to be evaluated independently of the original SUS. Figure 7.4 shows a boxplot diagram for the rating of participants' agreement to these statements. Although the statements were grouped in three pairs, which were each concerned with a particular aspect of how Stacksplorer changed the participant's experience, the 16 samples are not sufficient to perform a factor analysis that could prove that responses to the two statements of each group actually align with the same factor.

Participants found Stacksplorer to be beneficial for source code understanding.

Regarding source code understanding, nearly all participants strongly agreed that Stacksplorer has benefits compared to Xcode without the plug-in (statement 12). Hence, we could confirm H4. However, agreement to statement 11, stating that code understanding was easy using Stacksplorer, is not similarly overwhelming. Quite a few participants still found the tasks challenging when using Stacksplorer. This comes as no surprise, since a large, feature-rich software project is always a complex artifact and hard to understand without prior knowledge.

Navigation was considered to be faster with Stacksplorer than without it.

For the next pair of statements, which were concerned with how fast source code could be navigated using Stacksplorer, answers were also very positive. More than half of the participants strongly agreed, that navigation with Stacksplorer is faster than without it. We think that this is primarily because of Stacksplorer's clear advantages for navigation to callers of a method, when compared to the project-wide search (7.2.4 – "Qualitative Observations"). What is additionally notable about the positive ratings for statements 13 and 14, is that both statements did not imply that "navigation" referred to "navigation along the call stack". We conclude that this type of navigation is so important for programmers that Stacksplorer manages to improve their overall impression of how quickly they can navigate through source code by solely improving this particular type of navigation.

Participants miss support for orientation in the project on a higher level.

The last two statements were concerned with the support Stacksplorer provides to help users knowing where they are in the source code. More than half of the participants agreed that they did not feel lost in the source code when using Stacksplorer (statement 16). However, users least agreed with statement 15, which states that Stacksplorer is an improvement compared to Xcode in this regard. The reason users did not see as much improvement compared to Xcode as for the other factors we asked for is probably that Stacksplorer only provides context that is relevant locally. There is no way to get a "bigger picture" of the software's structure directly from Stacksplorer. Although users did not agree with statement 15 as clearly as with the other statements, the median rating is still an agreement, so H5 could be confirmed.

7.2.6 Users' Comments

Nearly all participants asked for a public release of Stacksplorer.

After the tests, we spent some time chatting with the participants to get some additional feedback that could not be sufficiently expressed through the questionnaire. The most noticeable observation was that a vast majority of participants asked where and if they could download Stacksplorer. Some participants even asked that again when they met us later in the university. The overly positive reactions show that Stacksplorer definitely appeals to developers a lot.

Overlays from Stacksplorer could become messy with many methods in the side columns.

However, users had several minor concerns about Stacksplorer. Firstly, the overlays, which connect a method call in the source code with the corresponding entry in the right side column, got messy quickly. To ease following a particular overlay from the method call in the code to the entry in the side column, it was highlighted slightly when users hovered over it. Many users had problems noticing this indication, because the effect was too subtle to really stand out.

Xcode's support to navigate back to previously visited methods was problematic.

The forward/backward buttons in Xcode, which were increasingly used by participants when using Stacksplorer, navigate through a history of visited *points of interest*. Although Xcode considers every navigation performed with Stacksplorer as a point of interest, other locations in the source code may be considered points of interest, too. As it is not made obvious what is a point of interest for Xcode, this concept was often confusing for participants. Many users would have preferred a visualization of the navigation history within Stacksplorer.

Users had to wait for Stacksplorer too frequently.

Another concern users had was about speed. Because of the rather simple parsing algorithm we used, updating the side columns could take a while, depending on the current focus method. Although users understood that a research prototype may suffer from problems like this, they still found themselves hindered in navigating more quickly sometimes.

User defined paths turned out to be complicated to grasp for many users. The name "path" is misleading, since it suggests methods on a path had a defined order. One user even assumed that user defined paths could be used to define call stacks manually, which Stacksplorer would not detect automatically. In fact, adding a method to a path only means tagging it. To make things worse, visual feedback for a path is only visible in rare occasions, concretely, if the focus method and a method in one of the side columns are on the same path and overlays are enabled.

The "user defined path" feature was hard to understand.

We will address these concerns in the improved prototype (8 – "Improved Prototype").

Some users noted that, although Stacksplorer is a valuable addition for their workflow, they needed additional support to get an idea of the higher-level software structure. Mostly, users suggested some sort of graph visualization that shows a larger portion of the call stack at a time. However, these visualizations are problematic, since they would quickly contain lots of nodes and thereby become impractical to use. For example, Stacksplorer's source code contains 330 non-accessor methods; the number of outgoing edges of particularly interesting methods can easily exceed 10 or 20. Hence, users suggested an iterative technique to generate a graph containing only the information they were interested in. While some users thought of this interface to be detached from the source code, others indicated that they liked Stacksplorer especially because it shows contextual information next to the source code ("I like having information next to the code, the stuff I actually work with.", participant 3). Participant 5 suggested to allow collapsing the code editor to quickly switch between a bigger graph view and the current Stacksplorer view. Other participants imagined that advanced functionality could be incorporated into the "user defined paths" feature. For example, paths could auto-arrange themselves to reflect the actual call graph.

A visualization of a larger portion of the call graph would be desirable.

Problematic for this kind of advanced features is that the first prototype we used for the user test would not technically be able to gather the information required to show a larger portion of possible call graphs in an acceptable time. The technical foundation for these advanced visualization techniques will also be introduced in the improved version of the prototype.

The first prototype of Stacksplorer lacked the technical prerequisites to visualize larger parts of the call graph.

Chapter 8

Improved Prototype

"In the practical world of computing, it is rather uncommon that a program, once it performs correctly and satisfactorily, remains unchanged forever."

—Niklaus E. Wirth

Following the principle of iterative design, we incorporated user's suggestions and comments into an improved version of the prototype. The changes we made regarding Stacksplorer's design and implementation are explained in this chapter.

8.1 User Interface Refinements

Most suggestions for improvement related to smaller UI problems and not to Stacksplorer's concept.

Many comments we got concerned smaller UI problems. This is not surprising, because the prototype was aimed to explore the high-level concept of visualizing information about potential call stacks in two columns at the sides of the code editor. The detailed layout of associated controls was not the scope of the prototype. Furthermore, the concept was very appealing to the participants of the user study, so they were tempted to suggest rather iterative improvements than radical changes.

In particular, three aspects of the user interface were commonly criticized: The way overlays are highlighted, the "user defined path" feature, and the lack of a navigation history. We introduce our improvements for these aspects in the following sections.

8.1.1 Overlay Highlighting

Overlays were highlighted too subtly to be noticed.

To provide help with mapping a method call in the source code to an entry in the side column, Stacksplorer shows overlays connecting both. About half of the participants had problems following these overlays from the source code to the right side column or vice versa. When users hovered over an overlay, it was emphasized by doubling the opacity (effectively changing the color from a light gray to a very dark gray). This was considered to be too subtle by most users.

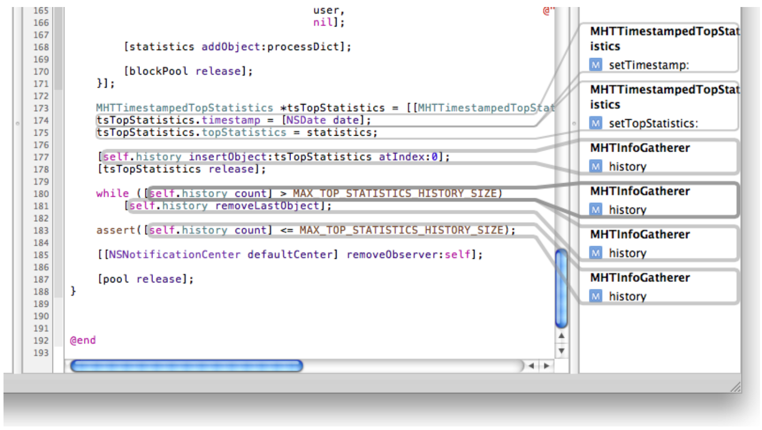
Users would like to see multiple calls to the same method easily.

Additionally, users brought up that they would like to be able to see *all* calls to one particular method from the focus method more easily. Showing each method only once in the side column was not an option, because all users were very satisfied with the current design, which shows all method calls in the same order as they appear in the source code, possibly including multiple calls to the same method, because this list can serve as a summary of the focus method.

Overlay highlighting was changed to be bolder.

To satisfy both requirements, we changed the way overlays are highlighted when hovering above them (see Figure 8.1). When hovering with the cursor over an overlay, the overlay's line width is doubled, in addition to the changing opacity. All overlays that belong to a method call to the same method are also highlighted, but without changing their opacity, i.e., in a more subtle way. The contents of the side columns could then remain unchanged.

Considering what actually makes emphasizing overlays necessary, we became aware that the connections between the source code and the table view cells became really hard to follow if the source code was far away from the entries in the side column (see upper screenshot in Figure 8.2). Matching a method call in the source code with an entry



```
                                    user,
                                    nil];

            [statistics addObject:processDict];

            [blockPool release];
        }];

        MHTTimestampedTopStatistics *tsTopStatistics = [[MHTTimestampedTopStat
        tsTopStatistics.timestamp = [NSDate date];
        tsTopStatistics.topStatistics = statistics;

        [self.history insertObject:tsTopStatistics atIndex:0];
        [tsTopStatistics release];

        while ([self.history count] > MAX_TOP_STATISTICS_HISTORY_SIZE)
            [self.history removeLastObject];

        assert([self.history count] <= MAX_TOP_STATISTICS_HISTORY_SIZE);

        [[NSNotificationCenter defaultCenter] removeObserver:self];

        [pool release];
    }


@end
```

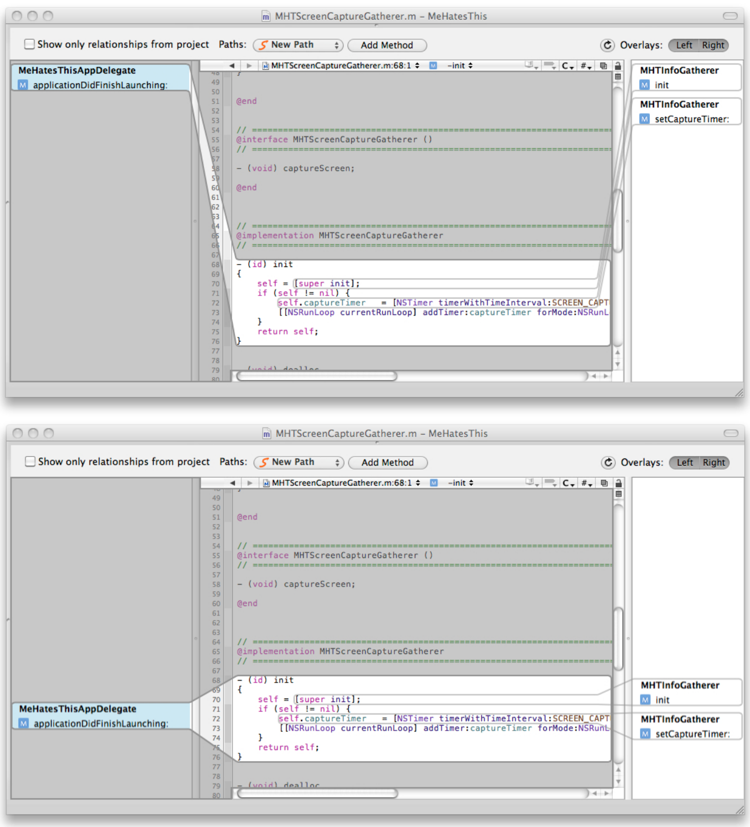
Figure 8.1: When hovering over an overlay with the mouse cursor, it is more prominently highlighted than before and other calls to the same method are highlighted as well.

in the table view is much easier if the table view cells are shown right next to the call in the source code (see lower screenshot in Figure 8.2). The improved prototype scrolls the table view automatically to achieve this layout. More exactly, the right side column is not always scrolled to be centered next to the method. Instead, it is scrolled so that the summed length of the connections between method calls in the source code and entries in the side column is minimal. This is achieved by calculating the summed difference in y-position between the cells in the side column and the position of the associated method calls in the source code. Then an offset is calculated, so that the summed difference becomes 0. If the currently visible part of the focus method contains more method calls than the right side column can show, a similar algorithm is used to show those methods in the side column for which the summed difference is minimal. However, the algorithms will always make sure that a maximal amount of information is shown in the side columns, i.e., the number of methods shown in the side column is maximal.

Side columns are automatically scrolled so that its entries are as close to the respective source code as possible.

To additionally tidy up the right side column, the improved version of Stacksplorer offers an option to hide calls to accessor methods from the side columns. We found that by analyzing calls to accessors mostly variable access was explored. However, from the preliminary study we con-

Calls to accessor methods can be hidden from the side columns.


MHTScreenCaptureGatherer.m – MeHatesThis
Show only relationships from project
Paths:
New Path
Add Method
Overlays:
Left
Right
MHTScreenCaptureGatherer.m:68:1
-init
MeHatesThisAppDelegate
applicationDidFinishLaunching:
MHTInfoGatherer
init
MHTInfoGatherer
setCaptureTimer:

```
@end


// ============================================================
@interface MHTScreenCaptureGatherer ()
// ============================================================

- (void) captureScreen;

@end



// ============================================================
@implementation MHTScreenCaptureGatherer
// ============================================================

- (id) init
{
    self = [super init];
    if (self != nil) {
        self.captureTimer   = [NSTimer timerWithTimeInterval:SCREEN_CAPT
        [[NSRunLoop currentRunLoop] addTimer:captureTimer forMode:NSRunL
    }
    return self;
}


- (void) dealloc
```

Figure 8.2: The improved prototype automatically scrolls the side columns so entries appear next to the source code. In comparison to the old layout (above), the new layout allows more easily mapping a location in the source code to the according entry in the side column.

ducted we can conclude that variable access and the call stacks are two different relationships. By hiding calls to accessor methods, we allow users to focus more on call stack exploration.

8.1.2 User Defined Paths

In our prototype, user defined paths were implemented like a tagging mechanism for methods. Contrary to user's expectation when reading the name, user defined paths do neither order methods according to a call stack, nor do they require that methods on a path call each other at all. Though some users used and liked the feature (7.2.4 – "Qualitative Observations"), the misleading name mostly caused user confusion, so the feature was not used. Hence, in the improved version of Stacksplorer "User Defined Paths" are called "Method Tags".

User defined paths provided merely a tagging mechanism for methods.

Those users who used the paths feature to capture a particular path through the source code also wanted to give an explicit order to items on the path to have the path editor reflect the actual order in which the items are called. In the refined prototype, this is possible by rearranging items on the path via drag and drop in the path editor (which is now called "Method Tag Editor").

Users can reorder the list of tagged methods.

Another problem with the initial design was that information about a method's path membership was only visible in very rare occasions. Path information was only shown if overlays were turned on and the focus method as well as one of the methods in the side column were on the same path. This also contributed to the fact that users had problems understanding the "user defined paths" feature. To solve this problem, we included an icon in the method's cell in the side columns if the method is tagged. Tag icons match the icons used in the tag selection drop-down at the top of the screen in color and shape.

Method tags are shown in the side columns.

The order of controls used to add a method to a path, was also problematic. Although only one user mentioned the problem explicitly, many users accidentally added methods to the wrong path and afterwards wondered why the path would not be visualized correctly. This is a typical mode error, with the selected path being the mode that influences the effect of the "Add to path" button. When reading from left to right, which is the usual reading direction for English, the button to add or remove a method to or from a path comes before the path selection drop-down. So, users were tempted to use it before being aware of the current mode. We switched the order of these controls, so users see the path selection first.

Controls to tag methods were rearranged.

8.1.3 Navigation History

Xcode's navigation through the history of visited locations in source code is problematic.

After exploring a particular call stack, users often navigated back using Xcode's forward and backward buttons at the top of the source editor. These, however, cause problems: Although many users think so, these buttons do not navigate through the history of open files. They merely navigate through a history of points of interest that have been visited. This is, in fact, every opened file, but also different locations in a file may be considered a point of interest. Hence, the forward and backward buttons will not reliably take users back to the method they previously inspected with Stacksplorer. After all, using the forward and backward buttons often leads to confusing results.

Users could not remember the methods they visited.

Some users evaded using the forward/backward buttons and instead always used the side columns to navigate. Unfortunately, method names are often similar, especially if a class offers a collection of methods performing the same functionality but with a varying number of user specified parameters (e.g., `parseFormat:forField:ofItem:` and `parseFormat:forField:ofItem:suggestion:`). Hence, backwards navigation by memorizing the path and traversing it backwards was often error prone.

Stacksplorer marks recently visited methods in the side columns.

As a result, we decided to visualize the history of visited methods in the side columns. Therefore, the last five visited methods are stored. If one of these methods appears in the side column, the cell's background is rendered in blue. The further the method is at the back of the history stack, the lighter (less saturated) the color is. We chose blue color, because people can discriminate saturation levels for blue best[1].

8.2 Performance Enhancements

One of the major problems in the user test was the speed of our plug-in. In particular, users who did not bother reading source code a lot sometimes waited for a couple of seconds

[1]http://www.visualexpert.com/FAQ/Part2/cfaqPart2.html

for information to appear. For incoming edges, the time to obtain a result depends on the size of the project (because a project wide search is performed). If the project is very large (like BibDesk), obtaining incoming edges may take a while (about six seconds in BibDesk). The time required to obtain information about outgoing edges only depends on the length of the focus method. Although the performance of the first prototype was reasonable for a research prototype, we decided to put some effort into making Stacksplorer much faster.

The performance of the first prototype was not sufficient for users that navigated quickly.

8.2.1 Cached Call Graph

Implementation

To reduce time required to update the side columns once the focus method changed, we implemented a caching mechanism for information about potential call stacks. The cache is a doubly linked directed graph, in which each node represents one method. An edge from a node A to a node B in the graph exists iff B is called from the implementation of A. We refer to this graph as *call graph*. In this call graph, we can obtain the information about callers and called methods for a given method in $\mathcal{O}(n)$, where n is the number of nodes, since a search for the node representing the focus method in the graph is required. (Note that the search is not actually performed on the graph, since it might not be connected. Instead, all nodes of the graph are additionally stored in an array for lookups.) For all practically feasible projects, lookup of information happens with no noticeable delay.

Stacksplorer caches information about all potential call stacks in an application.

To obtain an initial call graph, the algorithm iterates through all methods in the project and determines the methods they call using the same algorithm used in the first prototype. While this initialization is running, Stacksplorer works exactly like the first prototype. Once the initialization has finished, the object used to gather the information displayed in the side columns is substituted by an object complying with the same protocol, but using the call graph to obtain the information. The benefit of this tech-

As long as the call graph is created initially, Stacksplorer works like the first prototype.

nique is that users can start working right after they opened a project, although generating the call stack might take a while. Users can cancel the generation of a call graph if they prefer so. To keep the call graph up to date, it supports partial reloads. These are triggered whenever Xcode's project index updates.

Implications

The call graph allows revealing information, that could not be retrieved using the first prototype of Stacksplorer.

In contrast to our first prototype, where at each point in time only information about callers and called methods for the current focus method was available, Stacksplorer now knows the full information about potential call stacks everywhere in the project all the time. Besides improving the speed of lookups, this can also enable new kinds of queries for information. Stacksplorer's call graph, for example, implements Dijkstra's algorithm to obtain the shortest path from the focus method to each other method in the project. This could be used for a "smart path" feature that allows defining a user defined path that contains all paths that call a given method at some point. In addition, many suggestions users brought up for visualizations on a higher abstraction level (7.2.6 – "Users' Comments") could be realized using the functionality provided by Stacksplorer's call graph. For example, many users imagined a feature to generate a graph that contains all relevant method calls for a particular task. The user interface for this feature could be very responsive if Stacksplorer's call graph is used for the implementation.

The call graph provides a foundation to build visualization techniques for higher-level overviews.

A detailed exploration of the various interaction techniques that are possible by using the cached call graph goes beyond the scope of this work. In its current version, Stacksplorer contributes a framework other researchers or developers of Xcode plug-ins can build upon to implement and evaluate these techniques easily.

8.2.2 Algorithmic Improvements

Caching a list of all methods in the project decreases time to update the call graph by over 30%.

While the call stack is initialized, the algorithm used in the first prototype to extract method calls from the source code of a method is performed on all methods in a project. Hence, its weak performance becomes much more obvious, because even for small projects the generation of the call graph takes a lot of time. We sampled the CPU usage of Stacksplorer during a call graph update using Apple's Instruments (4.1.5 – "Reverse Engineering"). This analysis allowed us to identify which parts of the algorithm to extract outgoing edges from a piece of source code require the most processing time. It turned out that querying the project index for a list of all methods accounts for 32% of the runtime of an update process. Simply caching this list during a single call graph update could save this time.

Invocations of the code completion engine accounted for the biggest portion of the time required to update the call graph.

An even bigger portion of the runtime for a call stack update was consumed by invocations of the code completion engine (49%). A simple caching mechanism was not suitable to solve this problem, since the code completion is required to determine the type of each expression on which a method is called. Additionally, it was impossible to use only parts of the code completion's algorithm, since the code completion engine uses a C++ based parser internally. However, once the parser has determined the type of the expression that should be completed, it internally creates an Objective-C object again (an instance of `PBXCCType`) to represent this type. This Objective-C object is then queried for the list of suggestions returned by the code completion engine. Fortunately, this is also what consumes most processing time required for code completion. Hence, we could change the `PBXCCType` class, to not return a list of suggestions if the code completion is used from Stacksplorer's update algorithm, but to return only its type instead. This change led to another significant increase in update speed for the call graph. To determine from the changed implementation of `PBXCCType` if it is called from our update algorithm or not, we check if it runs on the same operation queue (4.1.6 – "Concurrency Programming") as our update algorithm. For that, we save the operation queue the update algorithm runs on in a singleton object that can then be accessed from the implementation of `PBXCCType`. The type name determined by the code completion engine, which is what we are interested in, is passed back to our algorithm by writing it to an instance variable of the singleton. The regular methods to obtain results from the code completion engine break due to our changes (if the code completion is used from our update algorithm).

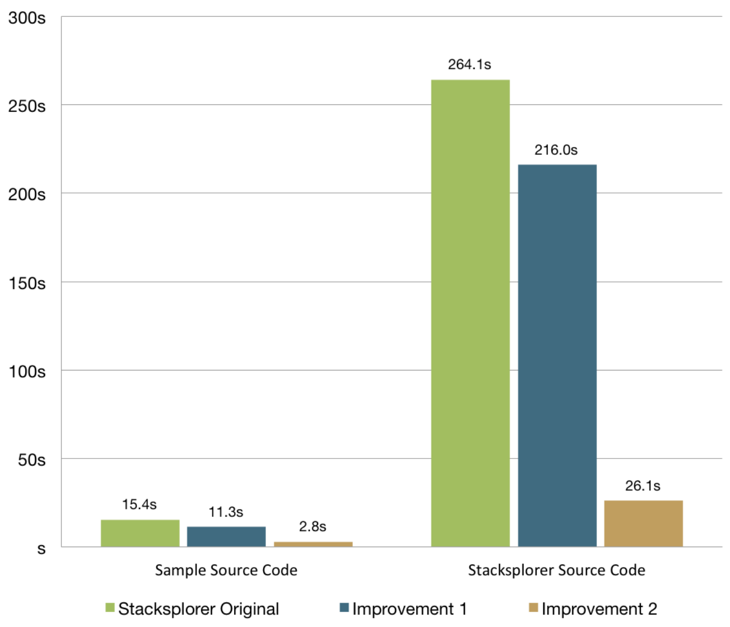


Figure 8.3: The chart shows for two different code bases how much time is required to create a complete call graph with the algorithm from the first version of Stacksplorer, after caching the list of all methods returned from the project index (Improvement 1), and after changing the code completion engine (Improvement 2).

The time required for call graph updates could be decreased by 90%.

To test the effectiveness of these changes to the update algorithm, we measured the time required to create a complete call graph for two different code bases after each iteration of the algorithm. Firstly, we used a small project that evolved from a different research project at our chair, which we will refer to as *sample code base*. It comprises roughly 1700 SLOC. As a second code base to test the performance with, we used the source code of Stacksplorer itself, which contained roughly 7000 SLOC at the time of testing. Figure 8.3 shows how much time is required to create a complete call graph for these two projects using the algorithm of the version of Stacksplorer that was used in user tests, after caching the list of all methods in the project (Improvement 1), and after hacking into the code completion algorithm (Improvement 2). Overall, we could reduce the time required to initialize the complete call graph by up to 90%. This improvement is also clearly noticeable when using the plug-in without generating the call graph; outgoing edges appear with no noticeable delay.

Chapter 9

Summary and Future Work

"The future is not what is coming at us, but what we are headed for."

—Jean-Marie Guyau

This work complements existing research about programmer's work practices and about tools to support software developers. In this last chapter, we give a summary of our work and point out interesting questions for future research.

9.1 Summary and Contributions

Stacksplorer allows navigating through potential call stacks of an application.

In this thesis, we presented Stacksplorer, a novel visualization technique for code browsing, which displays information about potential call stacks in two columns next to the source code. The method currently edited in the central source code editor is called the *focus method*. The side columns display methods calling the focus method (on the left) and methods called from the focus method (on the right). Stacksplorer allows navigating through potential call stacks in an application horizontally, leaving intact the well-known vertical navigation through a single implementation file (typically representing one class). Graphical overlays that extend over the source code and the side columns make intelligible which information is displayed in the side columns. Additionally, Stacksplorer allows tagging methods for further reference.

A software prototype is available as Xcode plug-in.

We presented a working prototype of this visualization technique, which integrates into Apple's Xcode IDE. The prototype is able to gather and visualize data from arbitrary real world applications developed in Objective-C. Its performance and its visual appearance have been iteratively refined. The latest iteration of Stacksplorer includes a cached call graph, in which Stacksplorer stores information about all potential call stacks in an application. Hence, the prototype also contributes a framework for the development of more advanced visualization tools, which show a larger extent of call stack information at a time.

A user test could show the effectiveness of Stacksplorer's visualization technique.

To evaluate Stacksplorer, we conducted a user test, in which participants had to work on maintenance tasks in a large software project that participants did not know beforehand. The participants were overly satisfied with Stacksplorer. In the study, they had to work on one task with and on one without Stacksplorer. One of two tasks could be completed significantly more often when participants used Stacksplorer. Additionally, the task in which participants used Stacksplorer could be solved significantly faster than the other task. Although other comparisons in our study were not significant, we can conclude that Stacksplorer is a highly valuable addition to a developer's toolkit.

Developers need to navigate along structural relationships in the source code to understand a project.

The work on Stacksplorer is rooted in a preliminary study we conducted, in which we could confirm that developers using Xcode to develop Objective-C applications exhibit similar navigation behavior as Java developers. The similarity in navigation behavior between Java developers and developers of other languages had not yet been tested. Additionally, we could show that structural dependencies in source code can serve as important guidance for programmers working on maintenance tasks, i.e., on an existing code base. However, not all structural dependencies are equally important for each kind of task: While bug fixing requires a thorough understanding of a program's call stack, refactoring tasks require more insight in the use of variables in a single method or class. Study participants found that Xcode, as one example of a modern IDE, does not support this structurally guided navigation satisfactorily. Some of Xcode's tools that do not support structurally guided navigation were considered nearly superfluous and, hence, increase Xcode's complexity unnecessarily. These results should apply to other modern IDEs providing similar tools as Xcode as well.

The results from this thesis up to and including chapter 6 – "Software Prototype" were published as work in progress at UIST 2010 [Krämer et al., 2010].

9.2 Future Work

Besides the lessons learned from the work on Stacksplorer, we also noticed several open research questions, which we consider worth investigating in more detail. In the following sections, we present these open questions.

9.2.1 Structural relationships

Other structural relationships besides call stacks might be visualized.

Because developers gave Stacksplorer an enthusiastic reception, it is enticing to increase its applicability by supporting a wider range of structural relationships. The preliminary study we conducted showed that, for example, access to variables would be important for developers. In addition, exploring notifications in Stacksplorer would be desirable, since Xcode's native support for this task was considered particularly underwhelming in the preliminary study.

Stacksplorer's design is optimized to visualize potential call stacks.

Other kinds of relationships might also require slight variations of the visualization to be useful. At least the mapping which information is shown in the side columns has to change slightly. Regarding notifications, three different kinds of locations in the source code are relevant: Firstly, all methods from which a given notification is posted; secondly, all methods which receive the posted notification; and thirdly, all methods that register a receiver. The existing concept of Stacksplorer provides no clear reasoning which of these kinds of information should appear in which side column for which focus method.

When visualizing variable access, information relates to a single symbol instead of a method.

Considering variable access, the more important relationship compared to notifications according to the preliminary study, problems are even tougher. Interesting about variable access is to see all methods from where a given variable, not a complete method, is accessed. To incorporate variable access into Stacksplorer would require to introduce the concept of a *focus variable* to which the information in the side columns relates instead of a focus method. Another problem arises, because variable access and call stack information are hard to separate completely since variable access in object-oriented software is often wrapped in accessor methods.

9.2.2 Runtime Traces

Information about actual program traces could be visualized as well.

To better isolate relevant paths through the source code, actual application runs could be traced to find out which methods are actually called in which order and how frequently. This would, for example, allow to give edges in the call graph a specific weight that might be mapped to the thickness of the overlays in Stacksplorer, which connect the entries in the side columns with method calls in the source code. In combination with unit tests, specific (erroneous) behavior of an application might be traced and the actual call stack could be visualized. Traces also allow developers to spot methods which are never called, or only very rarely, and hence to find opportunities to clean up the code by removing parts that are no longer used.

Information obtained through static analysis is available faster.

We think runtime traces should always be an addition to static code analysis, not a replacement. Static analysis as used for Stacksplorer is applicable at any time during development, even if the source code contains a compile time error. Information obtained through static analysis is quickly available, as it does not require running the application. Static analysis also reveals call stacks that existing test cases do not cover.

9.2.3 Storing Interesting Paths

Users wanted to store a particular path instead of just tagging methods.

Our prototype did include a method tagging feature, which was originally designed to allow storing certain paths through the source code. Although some users liked and used this feature, there also was a demand for a tool that works more automatically and is more tied to the concept of call stacks.

Other means to specify a path to store should be explored.

Consequently, other methods to define a path should be explored. For example, one might pick a single method and visualize all paths eventually calling this method at some point, or all paths that eventually are called from the single method. The latter would be especially useful if users picked an anchor point during their exploration, since it would always show them how to navigate back to the anchor point. Another possibility to define a path in the source code would be to pick two methods to select all paths connecting these two methods in the call stack. This would, for example, help to understand why a given method in the data model is called after a certain UI action was triggered.

More graphical representations of paths in the path editor should be explored.

Additionally, it might be explored if a more advanced representation of methods in the path editor is useful. Instead of showing a list of methods on a path, the path editor could also present a graph that shows how methods are connected in the call graph.

9.2.4 Advanced Visualization Techniques

Visualization techniques providing a high level overview of source code are still missing.

More advanced visualization techniques are not only applicable in the path editor but would also be useful in general to provide a higher level overview of the software's structure, which is not provided by the current version of Stacksplorer. Some ideas from users how such an interface could integrate with Stacksplorer were presented earlier (7.2.6 – "Users' Comments"). High-level visualizations of source code have also been the topic of related work (3 – "Related Work"). However, there is still much room for improvement and innovation. In the user test we conducted, we could observe that, in an unknown code base, finding the correct place to start the investigation from is still a big problem.

A design space for code visualization would help to map existing visualizations.

To compare the different existing approaches as well as the techniques proposed in the discussion of our user study, it might be useful to define a design space in which these visualization techniques can be categorized. Possible dimensions of this space include the amount of source code shown, the structural relationships that are visualized (e.g., static object hierarchy or call stack), and the kind of representation used (e.g., query language or graph visualization).

9.2.5 Mental Models of Software

Mental models of developers are widely unknown.

Although some analysis has been done regarding the cognitive models developers use to understand unknown source code, less is known about the mental models developers build of source code they understood. Learning more about these models would help to design more appropriate representations of source code. We suggest a study in which developers are asked to build their own representations or are asked to explain structural aspects of their source code to novices using any kind of visualization they like. This study may provide interesting insights into the different mental models of developers, and how well these models work to convey the information to others.

Appendix A

Hide a Search from the Project Browser

```
//
//  PBXBatchFinder.h
//

#import <Cocoa/Cocoa.h>

@interface PBXBatchFinder (CNPlugin)

-(BOOL)visibleInSmartGroup;
-(void)setVisbleInSmartGroup:(BOOL)flag;

@end

//
//  PBXBatchFinder.m
//

#import "PBXBatchFinder+CNPlugin.h"
#import <objc/runtime.h>

@implementation PBXBatchFinder (CNPlugin)

static char visibleInSmartGroupKey;

- (BOOL)visibleInSmartGroup;
{
    return [objc_getAssociatedObject(self, &visibleInSmartGroupKey)
            boolValue];
}
```

```
-(void)setVisbleInSmartGroup:(BOOL)flag;
{
    objc_setAssociatedObject(self,
                             &visibleInSmartGroupKey,
                             [NSNumber numberWithBool:flag],
                             OBJC_ASSOCIATION_RETAIN);
}

@end

//
//  PBXFindSmartGroup+CNPlugin.h
//

@class PBXBatchFinder;

@interface PBXFindSmartGroup (CNPlugin)

- (void)swizzleObserveBatchFinder:(PBXBatchFinder *)finder;

@end


//
//  PBXFindSmartGroup+CNPlugin.m
//

#import "PBXFindSmartGroup+CNPlugin.h"
#import "PBXBatchFinder+CNPlugin.h"

@implementation PBXFindSmartGroup (CNPlugin)

+ (void)load;
{
    [self exchangeInstanceMethod:@selector(observeBatchFinder:)
                      withMethod:@selector(swizzleObserveBatchFinder:)];
}

- (void)swizzleObserveBatchFinder:(PBXBatchFinder *)finder;
{
    if (![finder visibleInSmartGroup])
        return;

    [self swizzleObserveBatchFinder:finder];
}

@end
```

Appendix B

Preliminary Study: Questionnaire

Question numbers were not part of the original test and are included for reference only.

Working with source code

This test is a part of my, Jan-Peter Krämer's, diploma thesis. I research tools to support structurally guided navigation in source code. With this test, I want to investigate how programmers use available tools for navigation.

Background

Q1 **Age**

Q2 **Gender**

- ◯ Male
- ◯ Female

Q3 **Occupation**

Q4 **What is your highest academic degree?**

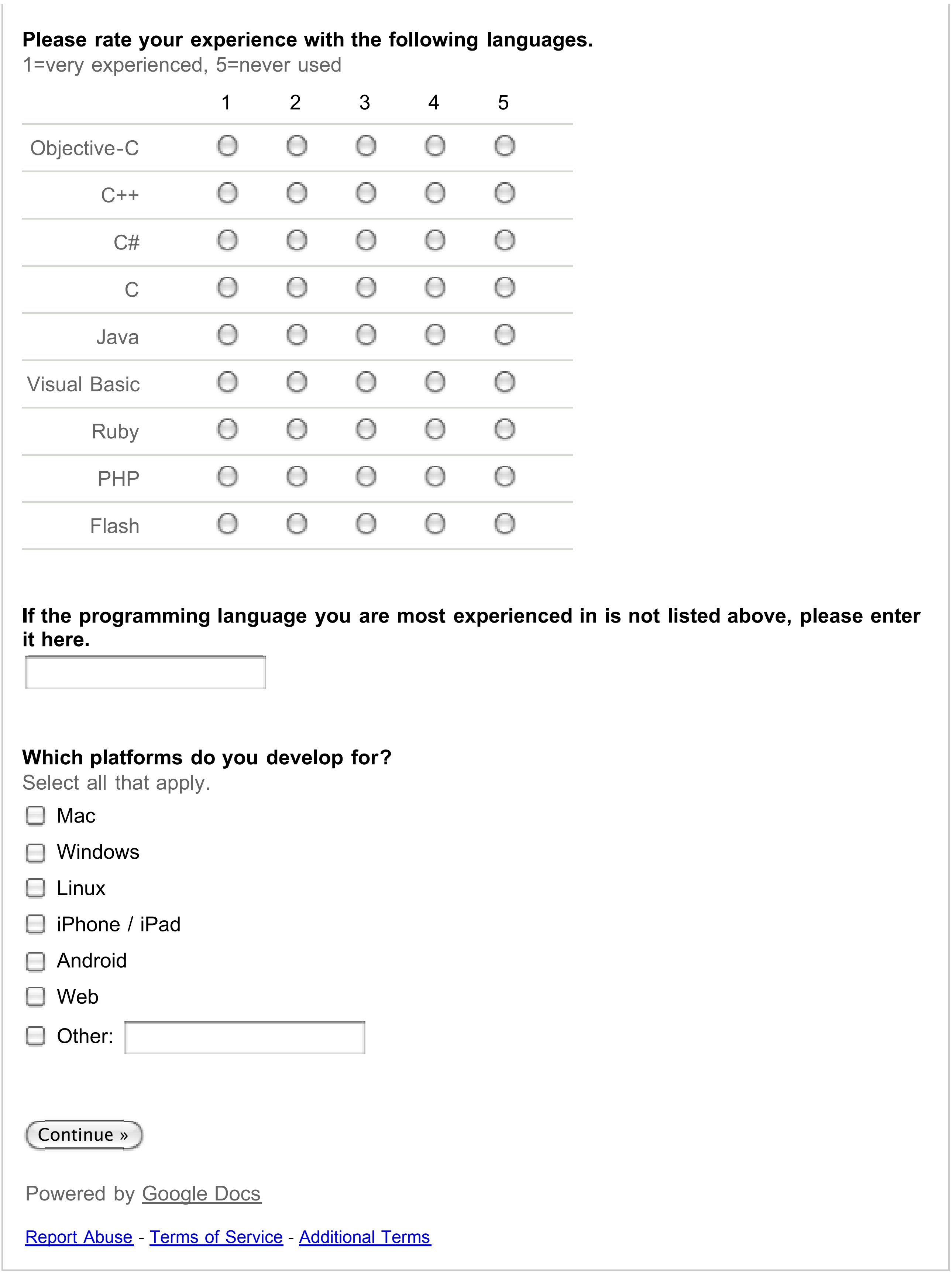
- ◯ Abitur
- ◯ Bachelor (Diplom-FH)
- ◯ Master (Diplom)
- ◯ PhD
- ◯ Other:

Q5 **If your highest degree is university level or if you are a student: What field did/will you major in?**

Q6 **How many hours per week do you spend programming?**

Q7 **How many years of programming experience do you have?**
Please include experience with any language.

Q8 **How many years of experience with Objective-C and Cocoa/Cocoa Touch do you have?**



Q9 **Please rate your experience with the following languages.**
1=very experienced, 5=never used

	1	2	3	4	5
Objective-C	○	○	○	○	○
C++	○	○	○	○	○
C#	○	○	○	○	○
C	○	○	○	○	○
Java	○	○	○	○	○
Visual Basic	○	○	○	○	○
Ruby	○	○	○	○	○
PHP	○	○	○	○	○
Flash	○	○	○	○	○

Q10 **If the programming language you are most experienced in is not listed above, please enter it here.**

Q11 **Which platforms do you develop for?**
Select all that apply.

- ☐ Mac
- ☐ Windows
- ☐ Linux
- ☐ iPhone / iPad
- ☐ Android
- ☐ Web
- ☐ Other:

Working with source code

Navigation

The following questions are concerned with different types of navigation in Xcode. Some of them might need clarification: "Navigating to a known part in the source code" refers to navigation actions where you already know, where you have to go to. "Navigating the call stack" means, navigating from a method implementation to either the declaration of a method, which is used, or to the caller of the method. "Navigating variable access" describes navigation from any occurrence of a variable to other methods in the source code that read or write the same variable.

Q12 **How frequently do you use each navigation technique while bug fixing?**
1=frequently, 4=seldom, 5=never

	1	2	3	4	5
Navigating to a known part in the source code	○	○	○	○	○
Navigating the call stack	○	○	○	○	○
Navigating variable access	○	○	○	○	○
Navigating between poster and recipient of a notification	○	○	○	○	○
Navigation between interface and implementation	○	○	○	○	○
Navigating between objects and their delegates	○	○	○	○	○
Other navigation	○	○	○	○	○

Q13 **How frequently do you use each navigation technique while refactoring?**
1=frequently, 4=seldom, 5=never

	1	2	3	4	5
Navigating to a known part in the source code	○	○	○	○	○
Navigating the call stack	○	○	○	○	○
Navigating variable access	○	○	○	○	○
Navigating between poster and recipient of a notification	○	○	○	○	○
Navigation between interface and implementation	○	○	○	○	○
Navigating between					

objects and their delegates	○	○	○	○	○
Other navigation	○	○	○	○	○

Q14 **How frequently do you use each navigation technique while adding new features?**
1=frequently, 4=seldom, 5=never

	1	2	3	4	5
Navigating to a known part in the source code	○	○	○	○	○
Navigating the call stack	○	○	○	○	○
Navigating variable access	○	○	○	○	○
Navigating between poster and recipient of a notification	○	○	○	○	○
Navigation between interface and implementation	○	○	○	○	○
Navigating between objects and their delegates	○	○	○	○	○
Other navigation	○	○	○	○	○

Q15 **Please rank the top 5 most frequently used types of navigation in you workflow.**

	1	2	3	4	5
Navigating to a known part in the source code	○	○	○	○	○
Navigating the call stack	○	○	○	○	○
Navigating variable access	○	○	○	○	○
Navigating between poster and recipient of a notification	○	○	○	○	○
Navigation between interface and implementation	○	○	○	○	○
Navigating between objects and their delegates	○	○	○	○	○
Other navigation	○	○	○	○	○

Q16 **If you do "other navigation" at all, please explain which relationships in the source code, other than those listed above, you explore by navigation.**

« Back   Continue »

Powered by Google Docs

Report Abuse - Terms of Service - Additional Terms

Working with source code

Support

Q17 **Please rate how well Xcode supports each navigation action.**
1= very useful support, 5= no support

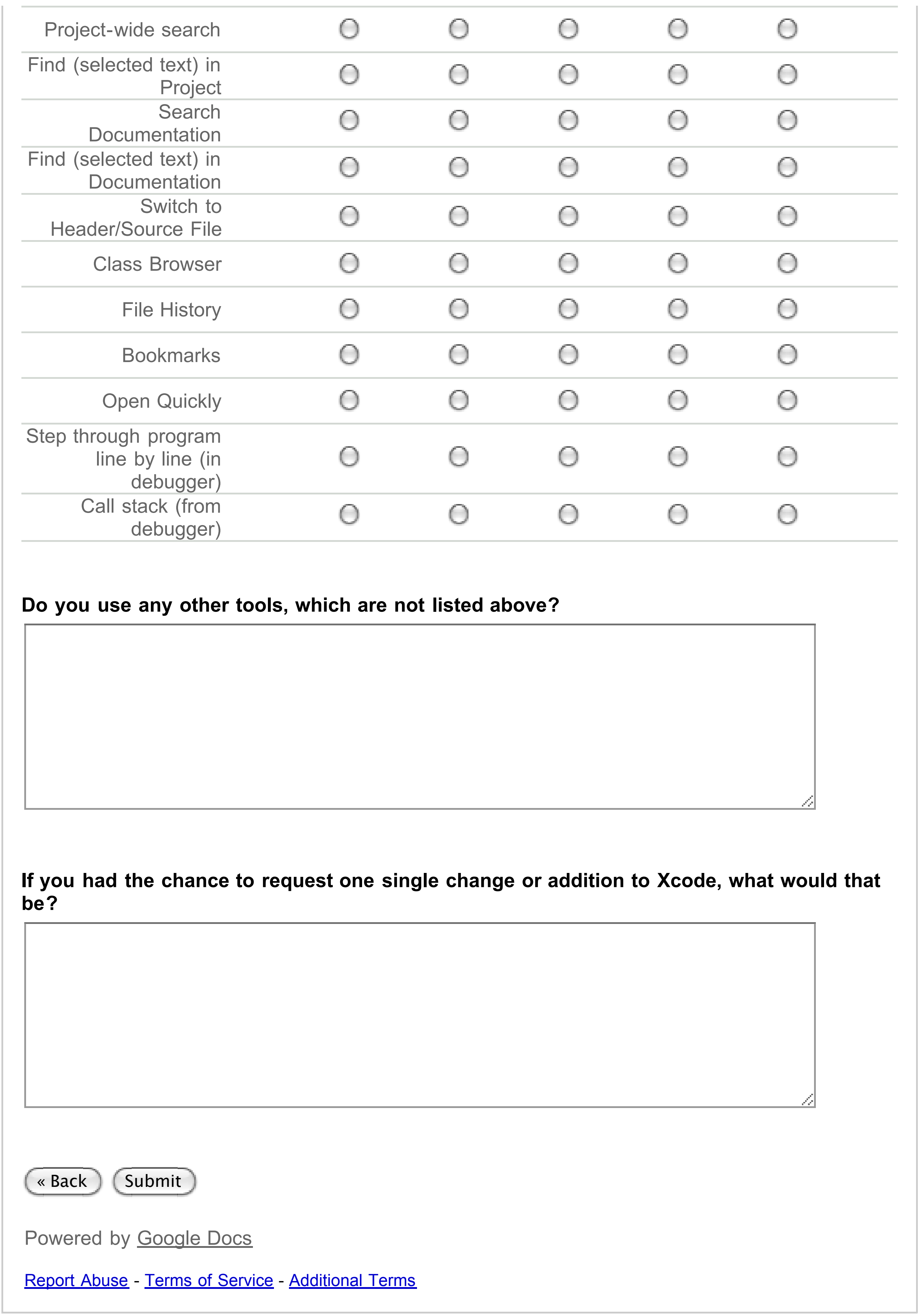
	1	2	3	4	5
Navigating to a known part in the source code	○	○	○	○	○
Navigating the call stack	○	○	○	○	○
Navigating variable access	○	○	○	○	○
Navigating between poster and recipient of a notification	○	○	○	○	○
Navigating between interface and implementation	○	○	○	○	○
Navigating between objects and their delegates	○	○	○	○	○
Other navigation	○	○	○	○	○

Q18 **Can you think of something you find particularly annoying while exploring source code?**

Tools

Q19 **Please rate the importance of each tool listed below.**
1= essential, 5= unnecessary

	1	2	3	4	5
File Browser	○	○	○	○	○
Jump to Definition	○	○	○	○	○



Project-wide search	○	○	○	○	○
Find (selected text) in Project	○	○	○	○	○
Search Documentation	○	○	○	○	○
Find (selected text) in Documentation	○	○	○	○	○
Switch to Header/Source File	○	○	○	○	○
Class Browser	○	○	○	○	○
File History	○	○	○	○	○
Bookmarks	○	○	○	○	○
Open Quickly	○	○	○	○	○
Step through program line by line (in debugger)	○	○	○	○	○
Call stack (from debugger)	○	○	○	○	○

Q20 **Do you use any other tools, which are not listed above?**

Q21 **If you had the chance to request one single change or addition to Xcode, what would that be?**

Appendix C

User Test: Task Descriptions

Task 1.1

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 1.2

One of your colleagues suggests implementing the change from 1.1 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.

Task 2.1

For a (hypothetical) trial version of BibDesk, the BibTeX output should be changed to contain “Exported by BibDesk” in the notes field whenever a BibTeX file is saved. Where should this change be implemented? You do not need to consider that the added note will, of course, show up in BibTeX when opening the created BibTeX file.

Hint: BibDesk’s document format is also BibTeX.
From Apple’s NSDocument documentation:

Commonly Used Methods

`dataOfType:error:`
Returns the document’s data in a specified type.

`readFromData:ofType:error:`
Sets the contents of this document by reading from data of a specified type.

`writeToURL:ofType:error:`
Writes the document’s data to a URL.

`readFromURL:ofType:error:`
Reads the document’s data from a file.

`windowNibName`
Returns the name of the document’s sole nib file (resulting in the creation of a window controller for the window in that file).

`makeWindowControllers`
Creates and returns the window controllers used to manage document windows.

Task 2.2

The “search for” command in apple script should also consider the name of the journal when matching the string. Where should this change be implemented and which other feature would be affected by that change?

Hint: Each Apple Script command is implemented in a separate class.

Appendix D

User Test: Post Session Questionnaire

Stacksplorer - 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					

	Strongly Disagree				Strongly Agree
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					

Bibliography

Apple. Objective-C Runtime Programming Guide, 2009.

Apple. Cocoa Fundamentals Guide, 2010a.

Apple. Garbage Collection Programming Guide, 2010b.

Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, 2008.

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

Hugh Beyer and Karen Holtzblatt. Contextual Design. *interactions*, 6(1):32–42, 1999.

Barry W Boehm. Software engineering. *IEEE Transactions on Computers*, 25(12):1226–1242, November 1976.

Barry W. Boehm. *Characteristics of Software Quality*. 1978.

Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 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 - CHI '10*, New York, 2010. ACM Press.

John Brooke. SUS-A quick and dirty usability scale, 1996.

Ruven Brooks. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

Kunrong Chen and Václav Rajlich. Case study of feature location using dependence graph. In *Proceedings of the 8th International Workshop on Program Comprehension*, page 241. IEEE Computer Society, 2000.

Michael J Coblenz, Andrew J Ko, and Brad A Myers. JASPER : An Eclipse Plug-In to Facilitate Software Maintenance Tasks. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 65–69, Portland, Oregon, 2006. ACM Press.

M.J. Coblenz. *JASPER: Facilitating Software Maintenance Activities With Explicit Task Representations*. PhD thesis, 2006.

B Curtis. Substantiating programmer variability. *Proceedings of the IEEE*, 69(7):846–846, 1981.

Brian de Alwis, Gail C. Murphy, and Martin P. Robillard. A Comparative Study of Three Program Exploration Tools. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 103–112. IEEE, June 2007.

Robert DeLine, Mary Czerwinski, and George Robertson. Easing Program Comprehension by Sharing Navigation Data. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 241–248. IEEE, 2005.

Robert Deline, Mary Czerwinski, Brian Meyers, Gina Venolia, Steven Drucker, and George Robertson. Code Thumbnails: Using Spatial Memory to Navigate Source Code. In *Proceedings of the Visual Languages and Human-Centric Computing*, pages 11–18. IEEE, 2006.

Pierre Deransart, Laurent Cervoni, and Abdel Ali Ed-Dbali. *Prolog: the standard reference manual*. Springer-Verlag, London, UK, 1996.

L.P. Deutsch and A.M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, pages 297–302. ACM, 1984.

Martin Fowler, Kent Beck, John Brandt, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Longman, Amsterdam, 1999.

G.W. Furnas. Generalized fisheye views. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '86)*. ACM, 1986.

K.B. Gallagher. Visual impact analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 52–58. IEEE, 1996.

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.

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

Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD '03)*, pages 178–187. ACM, 2003.

A.C. Kay. The early history of Smalltalk. *ACM SigPlan Notices*, 28(3):69–69, 1993.

Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for IDEs. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168. ACM, 2005.

Henning Kiel. *Reducing mental context switches during programming by*. Diploma thesis, RWTH Aachen University, 2009.

Won Kim and Frederick H. Lochovsky. *Object-Oriented Concepts, Databases and Applications*. 1989.

Andrew Ko, Brad Myers, Michael Coblenz, and 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.

Andrew J Ko and Brad A Myers. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *International Conference on*

Software Engineering (ICSE '08), pages 301–310. IEEE, 2008.

Jan-Peter Krämer, Thorsten Karrer, Jonathan Diehl, and Jan Borchers. Stacksplorer: understanding dynamic program behavior. In *Adjunct proceedings of the 23nd annual ACM symposium on User interface software and technology*, pages 433–434. ACM, 2010.

G.E. Krasner and S.T. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system, 1988.

Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM, 2006.

Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and Software*, 7(4), 1987.

Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. 1981.

J Lewis and Jeff Sauro. The Factor Structure of the System Usability Scale. *LNCS: Human Centered Design*, 5619:94–103, 2009.

John A. Lewis, Sallie M. Henry, Dennis G. Kafura, and Robert S. Schulman. An empirical study of the object-oriented paradigm and software reuse. *Conference proceedings on Object-oriented programming systems, languages, and applications (OOPSLA '91)*, pages 184–196, 1991.

Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2): 111–122, 1993.

Bennet P. Lientz. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations*. Addison-Wesley, 1980.

Jochen Ludewig and Horst Lichter. *Software Engineering*. dpunkt.verlag, Heidelberg, 2007.

B. Meyer. *Object-oriented software construction*. 1997.

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

J. Nielsen and T.K. Landauer. A Mathematical Model of the Finding of Usability Problems. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 206–213. ACM, 1993.

Donald A. Norman. *The Design of Everyday Things*. 1988.

Object Management Group. OMG Unified Modeling Language Specification, 1997.

Object Management Group. Unified Modeling Language, Infrastructure, 2010.

Rob Van Ommering, Frank Van Der Linden, and Jeff Kramer. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

Thomas E. Potok, Mladen Vouk, and Andy Rindos. Productivity analysis of object-oriented software developed in a commercial environment. *Software: Practice and Experience*, 29(10):833–847, August 1999.

M P Robillard and G C Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, pages 406–416. ACM, 2002.

Martin P. Robillard. Automatic generation of suggestions for program investigation. *ACM SIGSOFT Software Engineering Notes*, 30(5):11, September 2005.

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

Jonathan Sillito, Gail C Murphy, and Kris De Volder. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

Janice Singer, Robert Elves, and Margaret-Anne Storey. NavTracks: Supporting Navigation in Software Maintenance. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 325–334. IEEE, 2005.

M Storey, K Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2-3):183–207, March 2000.

Frank Tip, Jong-Deok Choi, John Field, and G. Ramalingam. Slicing class hierarchies in C++. *ACM SIGPLAN Notices*, 31(10):179–197, October 1996.

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

Mark Weiser. Programmers Use Slices When Debugging. *Communications of the ACM*, 25(7):446–452, 1982.

Dennis Wixon, Karen Holtzblatt, and Stephen Knox. Contextual design: an emergent view of system design. In *Conference on Human Factors in Computing Systems*, pages 329 – 336. ACM, 1990.

Index

Typeset January 18, 2011