A case study of paired interleaving for evaluating code search techniques

Conference Paper · February 2014

DOI: 10.1109/CSMR-WCRE.2014.6747225

3 authors, including:

Kostadin Damevski
Virginia Commonwealth University
59 PUBLICATIONS **514** CITATIONS
SEE PROFILE

David Shepherd
ABB
35 PUBLICATIONS **598** CITATIONS
SEE PROFILE

Empirical Software Engineering manuscript No.
(will be inserted by the editor)

A Field Study of How Developers Locate Features in Source Code

Received: date / Accepted: date

Abstract Our current understanding of how programmers perform feature location during software maintenance is based on controlled studies or interviews, which are inherently limited in size, scope and realism. Replicating controlled studies in the field can both explore the findings of these studies in wider contexts and study new factors that have not been previously encountered in the laboratory setting. In this paper, we report on a field study about how software developers perform feature location within source code during their daily development activities. Our study is based on two complementary field data sets: one that reflects complete IDE activity of 67 professional developers over approximately one month, and the other that reflects usage of an IR-based code search tool by nearly 600 developers. Analyzing this data, we report results on how often developers use which type of code search tools, on the types of queries and retreival strategies used by developers, and on patterns of developer feature location behavior following code search. The results of the study suggest that there is (1) a need for helping developers to devise better code search queries; (2) a lack of adoption of niche code search tools; (3) a need for code search tool to handle both lookup and exploratory queries; and (4) a need for better integration between code search, structured navigation, and debugging tools in feature location tasks.

Keywords Code Search, Feature Location, Field Studies

1 Introduction

Many studies of developers performing software maintenance and evolution tasks have highlighted the challenges of performing effective feature location (Wang et al., 2011; Ko et al., 2006), which constitutes identifying the parts of the code that implement a specific functionality. Similar studies have also shown that poor feature location can affect both the speed with which the maintenance task is completed as well as the quality of the resulting code changes (Robillard et al., 2004). While such controlled laboratory studies have uncovered a great deal about feature location, they all observe developers in synthetic circumstances where the code bases are completely unfamiliar to the study participants while the time window to perform each task is artificially limited. In the field, developers commonly examine code at their own pace and possess some knowledge of the code base, albeit incomplete when the scale of the project is large enough. Conducting a field study of feature location, where software developers are monitored during their day-to-day work can validate laboratory studies at scale, under realistic conditions, and without observational bias. In this paper, we define field studies as observational studies of software developers, conducted during their daily work, with minimal or no interference to the developers' usual activity.

In general, there are relatively few large scale research field studies in software engineering, and there are no field studies of feature location that we are aware of. While the difficulty in finding developers who are willing to allow their activities to be monitored by researchers is likely the greatest challenge to conducting field studies, analyzing the resulting data can also pose challenges. The main problem in analyzing field IDE data is interpreting high-level developer behaviors from low-level logged events, without the assistance of other sources of information (e.g. screen capture, video, or questionnaires) typically available in laboratory experiments. This problem is exacerbated by noise in field data, which occurs as developers can often be interrupted during their daily work or frequently switch between several maintenance tasks.

In this paper, we describe a field study of feature location that we conducted to validate existing knowledge about this software maintenance activity and discover other interesting usage patterns exhibited by developers in the field. Specifically, we investigated developer behavior along several dimensions, including the frequency and types of code search tools used by developers, developer behavior before and after search, types of queries issued by developers, and complex feature location patterns. To analyze field data, we clustered relevant events to form time-based sessions, which we qualitatively determined were representative of developer behaviors. When possible, we also used existing models of developer behavior during feature location, constructed during controlled laboratory observations of developers, as a starting point for analyzing field data.

The resulting field study consists of two datasets, one consisting of all IDE actions by 67 developers at one company over a period of two months, which enabled examination of the feature location process in general, including tool types and complex actions. The other dataset consists solely of logs gathered by an information retrieval based code search tool, which includes 596 developers and 8,052 queries from developers that downloaded the tool anonymously. The IDE dataset provides information on many aspects of developer behavior, while the search tool dataset provides information focused on behavior in code search, commonly considered to be the initial activity performed by developers during a feature location task.

The contribution of this paper is in both the data analysis approach as well as the results from this analysis. The paper makes the following main contributions based on the two field studies of software developer activity. First, we confirm several commonly held beliefs about how developers perform feature location in the field:

– Developers frequently and consistently use code search tools in the field, averaging several uses of these tools per workday.
– Most queries consist of a single word.
– Developers often reformulate their query, and they do so by adding, removing, or changing a term.
– After starting with code search, developers' next step is most often navigation.

Second, we discovered several unexpected patterns in the field data:

– Developers rarely use code search tools that are applicable to overly specific and limited information in the code.
– Developers create many queries semi-automatically via copy/paste or editor text selection.
– Developers tend to issue queries immediately after opening a software project (or solution[1]).
– When performing complex feature location tasks, developers tend to switch modalities several times (e.g., *search* to *navigate* to *search* to *navigate*).

Both the confirmed and unexpected behaviors can shed light on developer behavior during software maintenance, expose potential deficiencies and thus improvements in laboratory experimental designs, and provide valuable suggestions to toolsmiths.

The organization of this paper is as follows. Section 2 describes the related work, while Section 3 describes the data sets we used and the type of analyses we performed. In Section 4, we introduce the findings of our field study, and, finally, in Section 5 we discuss the implications of those findings.

2 Related Work

The related work for this paper can be grouped into two parts: (1) feature location developer studies, and (2) large-scale field studies in software maintenance. We are unaware of any previous large-scale field study of feature location that intersects these two categories.

2.1 Feature Location Developer Studies

There have been a number of laboratory studies that have influenced our understanding of how programmers perform feature location during software maintenance. We outline the most influential of these here, highlighting mental models of developer behavior during feature location that were proposed by some of the studies. These mental models describe the thought process of a developer performing feature location, consisting of a set of steps mapping to a specific sequence of actions performed within the IDE. We relied on such a model in our field study to provide a basis for interpreting developer IDE actions.

An influential laboratory study of feature location was conducted by Ko et al. (Ko et al. (2006)). The 10 developers (experienced graduate students) in this study

[1] In this paper, we use the Visual Studio term solution to refer to a software project or a code base. A solution is a container consisting of one or more Visual Studio projects, which, in turn, contains a number of source code files.

were observed while performing a number of maintenance tasks in a 70 minute session on a Java paint application that they were completely unfamiliar with. The 5 assigned maintenance tasks were invented by the researchers and consisted of minor feature requests and bug reports (e.g., a button in the UI does not work). Periodic interruptions to the developers' work were simulated to more closely resemble real-world development scenarios. The study reported on the distribution of time on various parts of a maintenance task, highlighting the importance of feature location and comprehension, which dominated task time. The study further found that developers had difficulty keeping track of all the relevant parts of the code base, exposing opportunities for toolsmiths in maintaining task contexts for software maintenance. Another contribution is the construction of a high-level mental model of developers performing software maintenance tasks, which was not very specifically instantiated with IDE actions.

Sillito et al. conducted a study in which they used both code bases that the participants were familiar and unfamiliar with (Sillito et al. (2006)). The part of the study that used an unfamiliar code base consisted of 9 students that worked in pairs in 45 minute sessions on 4 tasks selected from an open source project's history. The part of the study that used a familiar code base included 16 industrial software engineers who were asked to think aloud while working for 30 minutes on maintaining their own code bases. Voice and video of the participants actions were recorded, coded and analyzied with the goal of identifying a comprehensive list of program comprehension questions that developers ask during software maintenance tasks. The questions identified by this study can be grouped into several categories, ranging from simple questions about individual program elements to complex relationships between groups of structurally interconnected methods.

A recent laboratory study performed by Wang et al. (Wang et al., 2011) monitored 20 experienced (18 graduate students and 2 industry developers) and 18 novice (3rd and 4th year undergraduate) developers in performing 6 short feature location tasks in two 60 minute sessions. The tasks were selected from the maintenance histories of open source projects. While observing the experienced developers, the researchers constructed a detailed mental model of feature location, based on debugging, information retrieval based search, and structured program navigation. The novice developers were taught this model, after which a noticeable uptick in the quality of feature location performed by these developers was observed. We use Wang et al.'s mental model in our analysis, exploring how the abstract IDE actions described in the model are realized by developers in the field.

With the exception of the second half of Sillito et al.'s study, all of the studies described here used mostly student developers working on unfamiliar code bases, although often choosing realistic tasks extracted from software project histories. All of the studies used short, fixed time intervals for the developers to perform the tasks. In most cases, developers were observed in disruptive ways, using video and audio monitoring and subsequent questionaires. In this paper, we conduct a study that is different from previous studies of feature location, which is based on IDE datasets collected unobstrusively while developers worked at their own pace on their own code bases. Our study also consists of larger groups of developers monitored for longer periods of time.

2.2 Large-Scale Field Studies in Software Maintenance

While relatively few large scale field studies have been conducted by software maintenance researchers, all of the ones that have been performed have uncovered interesting developer behaviors, which were previously unreported by researchers.

Murphy-Hill et al. conducted a large-scale field study of refactoring using several relevant datasets (Murphy-Hill et al. (2009)). One dataset contained IDE data of 41 developers, with an average of 66 hours of development time per developer. Another dataset used by Murphy-Hill et al. was the Eclipse Usage Collector dataset, which is extremely large (13,000 developers), but only consists of frequency counts for each command within the Eclipse IDE. The datasets used in our paper are similar to Murphy-Hill et al.'s in that they combine both detailed usage logs of a number of developers with a larger, but limited, dataset reflecting a much larger set of developers in the field.

A study of Eclipse usage based on IDE data was conducted by Murphy et al. (Murphy et al., 2006). The study used 41 developers' interaction histories gathered using the Mylar Monitor Eclipse plugin (Kersten and Murphy, 2005), which is a popular tool to gather usage data in this IDE. The study analyzed the frequency of use of a number of Eclipse features, including refactoring, navigation and search, and general commands and views (or windows), informing future development of this popular software development environment. Similar to Murphy-Hill et al., the dataset used by Murphy et al.'s study is large but extremely limited in the depth of information it can provide.

There have been a few large-scale studies of code search behavior when developers are searching on the web across a number of software projects (Bajracharya and Lopes, 2012). They find interesting results including that developers use source code on the web for various purposes, and use different forms of queries to express their information needs, including natural language and names of code entities they are aware of. We have observed similar types of queries in previous small-scale studies using Sando (Damevski et al., 2014) as well as in the larger collection of data analyzed in this paper.

3 Field Study Description

Our field study of programmers performing feature location is based on two data sets: (1) the Blaze dataset consisting of all the IDE interactions of 67 developers at ABB, Inc. over a period of about 2 months; (2) the Sando dataset consisting of 8,052 queries using the Sando search tool by 596 unknown developers in the field that downloaded the tool and issued more than one query[2]. Sando users that issued only one query were presumed to have only tried the tool and not performed any serious feature location, and therefore their interactions with the tool were removed from this dataset. We leveraged these datasets to shed more light on developer behavior during feature location, focusing mostly on observations during, before, and after the interaction of developers with code search tools.

[2] Sando usage data spanning from 05/2013 to 06/2014 was included in the dataset.

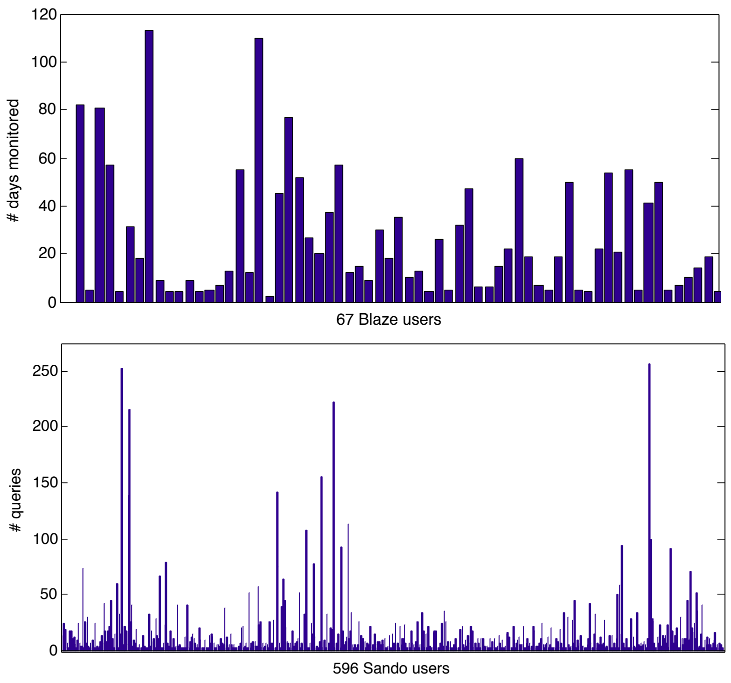

Fig. 1 Usage characteristics for the Blaze dataset (top) and the Sando dataset (bottom).

3.1 Data Collection Procedure

The Blaze dataset was collected from 67 volunteer developers at ABB, Inc. who installed a Visual Studio interaction monitoring extension[3]. This extension monitors and logs clicks and keypresses within the Visual Studio IDE, capturing timestamped information indicating each IDE command that was invoked by a developer. The data collection timespan differed for most developers as some of them installed the data collection tool later than others, some stopped collection data earlier than others, while a third group stopped and resumed data collection after a several day pause. All of the data was collected in the second half of 2013, and it consists of an average of 25.6 days of activity per developer with a standard deviation of 25.4 days. The usage characteristics of the Blaze dataset, in terms of work-days per user, is shown in the top part of Figure 1.

The Sando dataset consists of data describing developer interaction with the Sando open-source code search tool (Shepherd et al., 2012), implemented as a plugin to Visual Studio. Sando retrieves a set of program elements (e.g., methods, field, classes) in response to a user supplied text query. It uses information retrieval techniques, such as word stemming, and the vector space model to rank retrieved results based on relevance to the query. Sando is commonly downloaded from the

[3] The interaction monitoring extension, called Blaze, is implemented by researchers at ABB, Inc. Its name is the reason we refer to this dataset as such.

Visual Studio Gallery Site[4], which houses numerous Visual Studio extensions and tools. When installing Sando, users are asked to participate in volunteer data collection, and if they agree anonymous usage data is periodically uploaded from their machines to Sando servers. The Sando dataset spans 13 months and consists of 596 users and 8,052 queries. The number of queries for each of the users in the Sando dataset are shown in the bottom part of Figure 1. This figure shows that, in both the Blaze and Sando datasets, while the usage quantities vary significantly between users, neither dataset is dominated by few users and therefore should allow for widely generalizable observations.

3.2 Data Description

3.2.1 Blaze Dataset

The Blaze dataset contains most user clicks or keypresses in the Visual Studio IDE. Certain events are condensed for performance reasons, to reduce both the impact of data collection on Visual Studio response time as well as the space overhead of the gathered logs. For instance, keypresses on the Visual Studio editor window are logged only if they move the cursor down a line, ignoring edit commands that do not cause a line break and horizontal movements of the cursor. Certain user actions within Visual Studio are impossible to record as the IDE did not allow us to register a listener for that type of event, for instance closing or dragging and dropping a window within the IDE. Despite this, the number of logged events in the dataset is substantial, consisting of over 3.2 million entries. Below is an illustrative listing of events in the Blaze dataset where a developer started the debugger (`Debug.Start`) , then moved the cursor on the editor window (`View.File` and (`View.OnChangeCaretLine`), stopped on a breakpoint (`Debug.Debug Break Mode`), opened the call stack for a method (`View.Call Stack`), and used the Find in Files tool to search the code base (`View.Find and Replace`, `Edit.FindinFiles`, and `View.Find Results 1`).

```
2013-11-18 14:50:23.000, Debug.Start
2013-11-18 14:50:23.000, View.File
2013-11-18 14:50:23.000, View.OnChangeCaretLine
2013-11-18 14:50:23.000, Debug.Debug Break Mode
2013-11-18 14:50:33.000, View.Call Stack
2013-11-18 14:51:08.000, View.Find and Replace
2013-11-18 14:51:08.000, Edit.FindinFiles
2013-11-18 14:51:08.000, View.Find Results 1
```

All developers whose actions were captured in the Blaze dataset had access to similarly configured versions of Visual Studio with a few exceptions. Some developers used JetBrains' ReSharper toolkit for Visual Studio (ReSharper :: The Most Intelligent Extension for Visual Studio, 2014), which offers advanced program navigation and UI capabilities. Developers also had access to a custom program navigation tool from ABB, Inc. called Prodet, which allows exploration of a code base by following the call graph.

[4] `http://visualstudiogallery.msdn.microsoft.com`

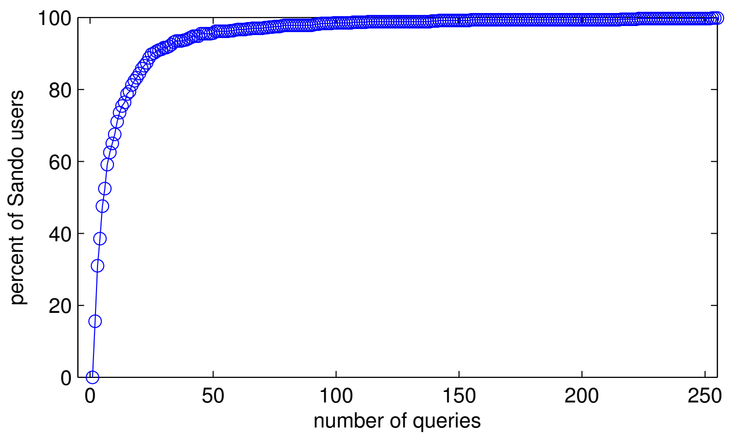

Fig. 2 A cumulative distribution function showing the number of queries issued by proportions of Sando's 596 users.

The developers had three code search tools available to them: Quick Find, Find in Files, and Sando. Quick Find, launched using the Ctrl+F key press, performs searches local to the currently open file in the editor using string matching or optional regular expression input. Find in Files also uses string matching or regular expressions, but produces results across the entire code base by default. The results produced by Find in Files are displayed in a separate window, using one line to describe the file containing the match, the matching line number, and a brief snippet of the code on that line. The matches from files currently open in the editor appear first, with the line number as the second order sorting term. Sando uses sophisticated Information Retrieval algorithms, based on the Vector Space Model, to index identifiers in the source code and retrieve relevant results. The results are ordered according to their relevance (using tf*idf scoring). A several line long snippet of code corresponding to each result can be previewed in a popup window using a single click. Double clicking on a Sando result opens it in the Visual Studio editor. Based on the scope of the search, we consider Quick Find a file-scope search tool, while we consider Find in Files and Sando project-scope search tools.

3.2.2 Sando Dataset

As developers issue queries about their software projects, the Sando code search tool records and collects anonymous and private data on developer interactions with the tool. Sando provides instructions that describes its use, displayed on install and when clicking a help button. The Sando dataset contains several types of information about each query, including its length and similarity to the previously submitted query, the retrieved results, including their number and type, and clicks on the retrieved results, including their rank and similarity to the query. The distribution of the number of queries that each Sando user issued, as a cumulative distribution function is shown in Figure 2. Below is a representative snippet of the Sando dataset, describing a user opening a Visual Studio solution, issuing a query, receiving 11 results, and clicking on the 5th result in the list.

```
2013-10-24 10:06:41,094 Solution opened: SolutionHash=522943624
2013-10-24 10:10:09,845 Query submitted by user: QueryDescription=Plain;
                        DiceCoefficientToPreviousQuery=0
2013-10-24 10:10:10,006 Sando returned results: NumberOfResults=11;
2013-10-24 10:11:31,268 User double-clicked a result: TypeOfResult=XmlElement;
                        ResultLanguage=OtherLang; ResultScore=0.051;
                        ResultRank=5
```

The Sando dataset contains little contextual information on the size of the code base, the language its implemented in, or other activities that the developer performs to locate features in the code outside of code search. Such information is not gathered due to privacy concerns, which could limit wider adoption of Sando, especially in the industrial setting. These threats are mitigated, however, by the size and diversity of the dataset which should produce statistically relevant measurements of developer behavior when performing code search in the field. A listing of all of the events used in the Sando and Blaze dataset is provided in Appendix A.

3.3 Data Analysis

Our main goal of the field study was to investigate developer behavior in regard to feature location as developers complete their normal daily activities. Specifically, we investigate frequency and types of code search tools used, developer behavior before and after search, types of queries issued by developers, and complex feature location patterns. Since both the Blaze and Sando datasets are large, spanning millions of log messages, and our interest is in understanding feature location developer behavior, we need an approach for identifying these user behaviors in the dataset. Simple pattern recognition, where certain sequences of clicks represent particular developer behaviors, is difficult to perform in the Blaze dataset because of the presence of ambiguous sub-patterns of clicks, which can correspond to several types of behavior.

The key strategy to gain insight into developer behavior from such low-level logged event data is to use the time of a specific click as an additional clarification parameter that can help resolve some of the ambiguity in the dataset. Therefore, based on sets of key log messages corresponding to relevant developer actions in time, our goal was to extract feature location *sessions*, where a specific set of key messages occur one or more times within a short, on the order of seconds, timespan. Based on the choice of the key messages that we use, we can identify various types of sessions, for instance, search sessions, structured navigation sessions, or debugging sessions. Also, by using only the log messages that belong to a specific tool we can produce, for example, Sando sessions or Quick Find sessions.

To automatically extract feature location sessions in the Blaze dataset, we use hierarchical agglomerative clustering (Manning et al., 2008) of log events according to their time distribution. This clustering algorithm produces a clustering tree, which can be cut at a variety of places to produce the desired number of clusters. We use a natural cut, which uses the ratio of time differences of the key messages, to choose the best number of clusters, constrained to between a 30-second to 5-minute interval. We used this interval to select a cut between the clusters as it reflects general expectations of user search behavior. This is to say that any two log messages that are less than 30 seconds apart are required to belong in the same session, while two messages with a distance of over 5 minutes are assumed not to be in the same session. If the distance is between these two limits, then the algorithm automatically determines whether or not the two messages should be grouped in the same session based on the distribution of other Sando clicks in a specific developer's dataset.

For instance, consider three clicks on the Sando code search tools at some relative times: 0 seconds, 15 seconds and 300 seconds. The clustering algorithm would definitely group the first two clicks, which are 15 seconds apart into the same session as they are below the 30 second lower threshold. Since the time between the second and third click is 285 seconds (or 4 min and 45 seconds), which is below the 5 minute upper threshold, they may or may not be grouped together depending on the distribution of other Sando clicks in a specific developer's dataset. If the ratio of 285/15 seconds is largest, then the second and third click would belong to different sessions; however, if a larger ratio between two clicks exists for this developer, then they would be in the same session.

Extracting groups of related developer actions in the Sando dataset is less difficult, as interactions with the tool are based on a query, and the developer's issue of a new query can be used as a delimiter in extracting related log events. While we often analyze the Sando dataset at the query level, we also rely on lexical similarity between query terms to group sets of queries into reformulation sequences. In the context of code search, such sequences represent the developer interacting with the tool to fulfill a single information need. This sometimes requires several queries, reformulating previously issued queries until a desired program element is located.

3.4 Threats to Validity

The goal of our field study was to improve on the realism, size and scope of previous studies in feature location. However, the challenge of a field study is in interpreting developer behaviors from low-level log data. This challenge is a particular source of threats to validity to the findings in this paper.

A significant threat to validity arises from our lack of knowledge of the tasks the developers in our study were trying to accomplish. For instance, we believe that they were performing software maintenance, though they could have easily engaged in "greenfield" software development for some of the time. This threat is mitigated partially in that we make no behavioral assumptions in our analysis and look solely at the events generated by the developers to form our conclusions.

Our analysis is largely based on clustering individual events into high level behaviors. While the clustering approach we used was simple and straightforward, a possible threat is that this mapping could be inaccurate. To mitigate this threat, we hand analyzed a sample of the clustered behaviors for validity and determined them to be reasonable.

Our results may not generalize due to several facts: (1) the Blaze dataset reflected developers from one company; and (2) all of the developers used Visual Studio with a few specific plugin extensions. The size of our Blaze dataset, consisting of over 60 developers, mitigates these concerns, as well as the fact that most

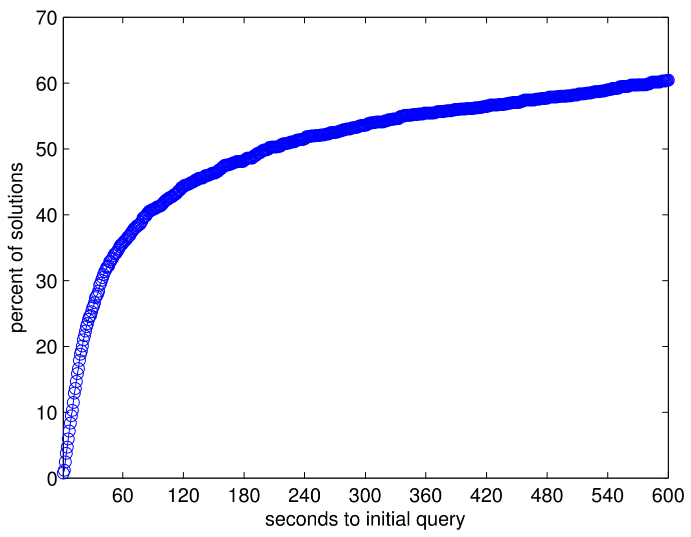

Fig. 3 Percent of code bases queried with Sando within a specific time interval (in seconds) of opening.

IDEs support plugins and many developers use at least some of them, making this a realistic scenario.

4 Findings

The findings of our study are intended to answer questions that would improve our understanding of feature location, including (1) how often developers use code search tools in general; (2) which tools are used, how often, relative to each other; (3) the types of queries issued to code search tools; and (4) the overall context following and preceding code search.

We present our findings grouped into two categories: (1) findings on the use of code search tools and (2) findings on complex feature location sessions. The first category focuses on examining developer use of code search tools, which are influential in feature location and commonly represent the first step developers take in performing feature location. Findings from analyzing the Sando dataset are presented in this category. Findings from analyzing the Blaze dataset also provide insights into developer behavior when performing a complex feature location task, which spans a significant period of time and uses a variety of IDE modalities (e.g., search and structured navigation).

4.1 Findings on the Use of Code Search Tools

Developers tend to issue project-scope queries within a few minutes after opening a new code base.
The Sando dataset tracks the open solution event in Visual Studio, representing when a developer opens a new code base to work on. There are many code bases

that never receive a query, presumably because the developer is already familiar with the code base. However, our analysis shows that solutions that receive a Sando query tend to get an initial query soon after the solution is opened.

The Sando dataset contains 1595 queried solutions. Figure 3 shows the distribution of time between a newly opened code base and the initial query. A large proportion of solutions, more than 40%, are queried within the first 2 minutes after being opened in Visual Studio.

Developers use both file-scope and project-scope code search tools frequently and consistently.

We differentiate two types of code search tools: project-scope and file-scope. Project-scope tools, like Find in Files and Sando in the Blaze dataset, are used to query the entire code base. They are commonly used to find a starting point in discovering a particular feature in the code, but can also have other uses, such as a lookup of an known identifier in a code base. File-scope code search tools, such as Quick Find (Ctrl+F) in Visual Studio, search only the currently open file. This search modality is common across a number of applications (e.g., browser or word processor) and is commonly used to navigate within an open page of text. Since it is possible in Visual Studio to produce project-scope search results with the file-scope search tool (Quick Find) by clicking the Find All drop down button, we explicitly looked for such sessions and categorized them as project-scope search as long as they performed at least one such search.

Based on our clustering of log events into sessions and subsequent identification of sessions as search sessions, each developer in the Blaze study performed an average of 3.32 search sessions per day. Out of these searches, there were 1.7 project-scope search sessions per user per day, using either Find in Files or Sando, and there was an average of 1.57 file scope searches that used Quick Find. As a reference, there was an average of 7.95 editing sessions [5] per developer per day.

Figure 4 shows the overall number of search sessions per day for each of the 67 developers in the Blaze study, and the proportion of those sessions that used project-scope vs. file-scope search tool(s). For instance, the first user (leftmost bar in Figure 4) performed a total of 3.4 search sessions per day, with an average of 1.9 being project-scope sessions. There were 4/67 users who did not use any search tools at all.

Almost all developers (60/67) used a file-scope search tool at some point during our data collection, and, similarly, a large proportion of the developers used one of the two project-scope search tools (50/67). Most (47/67) developers used both file-scope and project scope tools, while 28/67 used a combination of the two during the same code search session.

Search sessions that combined file-scope and project-scope search tools followed either an expanding scope or a reducing scope pattern. In expanding scope, developers started with a file-scope search that did not retrieve satisfactory results, and they expanded the scope by issuing a project-scope search query. In reducing scope searches, the developers started with a project-scope search and used file-scope search only when they had found the file containing the code of interest and needed to drill down further to the statement level. There was a roughly equal

[5] Editing sessions were identified by applying the session clustering algorithm on editing events.

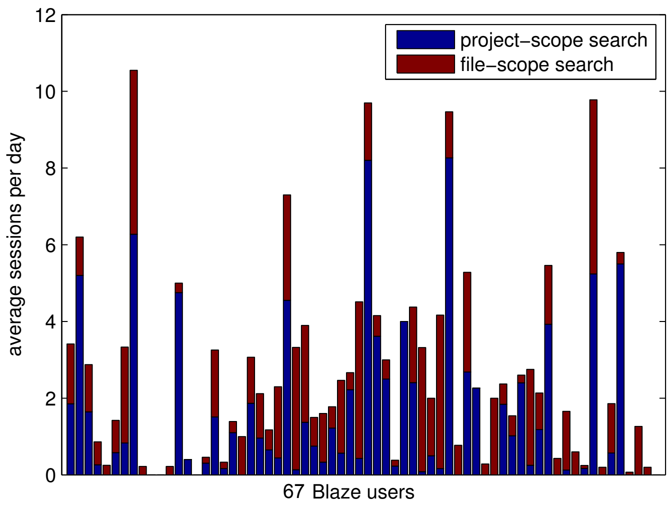

Fig. 4 Number of project-scope and file-scope search sessions per user in the Blaze dataset.

number of expanding and reducing scope search strategies observed in the Blaze study.

To investigate the consistency of usage of search tools over time, Figure 5 displays a timeline of the use of search tools for individual users and on average. The dashed lines in the image represent the number of search tool usage sessions per day, averaged over a 3-day window, for each individual developer, while the red solid line denotes the average usage over all developers in the Blaze dataset. The Sando dataset indicates a similar conclusion to the Blaze dataset and thus is not shown. The graph indicates a fairly consistent usage of search tools on a daily basis, with a few outliers who performed considerably more search at times. There was an average of 3.53 queries and 2.29 sessions per user per day.

We investigated the choice of individual search tools over time per developer to see if developers used the same tool consistently, switched tools consistently, or used various tools throughout the monitored time period. Sando is a more advanced project-scope code search tool than Find in Files: it uses a more sophisticated information retrieval algorithm instead of simple string matching and also ranks the retrieved results instead of returning a lengthy list. Thus, one might expect that once developers tried using Sando, they would replace Find in Files usage with this improved tool. Out of the 39 developers in the Blaze study who used Sando, only 13 of 39 (one third) never used Find in Files after using Sando. However, 26 of 39 (two thirds) developers used both tools interchangeably. The most likely explanation for this is Sando's limitation in indexing all languages in the Visual Studio ecosystem, such as Visual Basic and Javascript. Find in Files, on the other hand, works consistently on all files. The remaining set of developers (28/67) used only Find in Files, either because they were unaware of Sando or because they preferred Find in Files.

To examine how effectively feature location tools were used by developers, we used the Sando dataset, which differentiates types of user behavior with a retrieved result set. If we consider double-clicking on a result as an indicator of

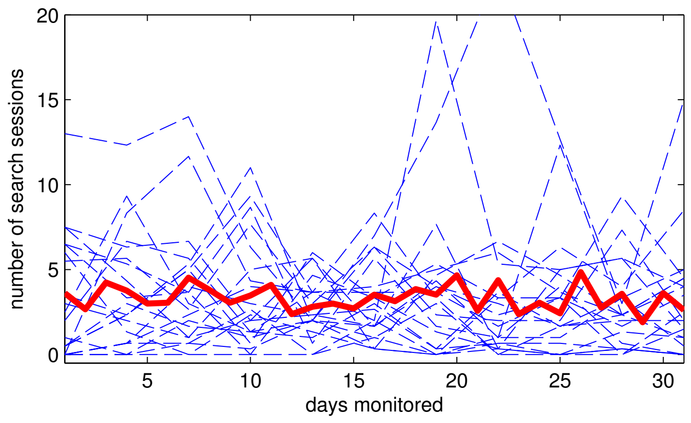

Fig. 5 A timeline of code search tool usage in the Blaze dataset. Individual users are shown as dashed lines, while the average across all users is shown as a red line.

retrieval effectiveness, 31% (2505/8052) of the queries had at least one click. The median rank of the highest double-clicked result in the retrieved result set was 2, with a standard deviation of 11.9, following an exponential distribution. This is indicative of the fact that while most developers clicked on the first couple of results, there were some developers who examined results deep in the retrieved set.

Developers perform lookup style searches fairly often.
The most frequent interaction prior to and following a click on a code search tool was a click on the editor window, constituting well over 50% of pre and post code search events. While we did not capture the actual editor selections from cut/copy due to the associated slow downs for users, we monitored copy/cut and paste commands being performed. The logs show that 9% of messages immediately following the opening of a Quick Find window were a paste; 14% of messages immediately following Find in Files window opening were a paste; and 5% of messages immediately preceding any Sando interaction were copy and paste. The use of copy-paste queries suggests that developers are searching for exact parts of code, i.e., performing lookup searches. However, they are not the only indicator of lookup style searches, as FindinFiles and Quick Find automatically copy selected items into the query window and those events are not monitored, as well as there are queries that are typed directly by the developer that could also be lookup style searches without using copy and paste commands. Thus, these percentages are a minimum estimate of lookup searches, and thus indicate that lookup searches occur fairly often with all these search tools.

NavigateTo, generally believed to be the most effective tool for performing lookup-style searches, was rarely used.
In Visual Studio, the NavigateTo tool uses simple string matching to retrieve type names (e.g., classes), method names, file names, or project names. This kind of tool is commonly available in many IDEs, such as Eclipse where a similar tool is called OpenType. Since it only indexes a selected group of items, NavigateTo/OpenType

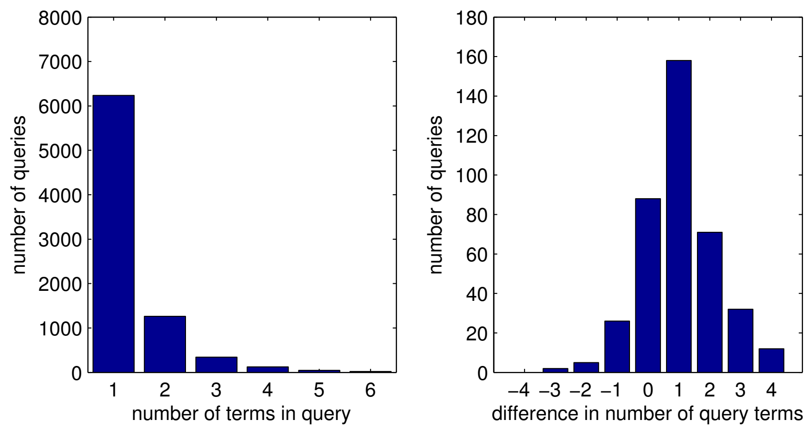

Fig. 6 Distribution of the number of terms per query in the Sando dataset (left). Number of terms difference between an initial and a reformulated Sando query (right).

can be very effective, producing very few false positive or false negative results for a developer query.

In the Blaze dataset, we detected only two instances of developers using NavigateTo, each by two separate developers. Both of these developers used Sando and Quick Find fairly consistently, but neither of them used Find in Files. Murphy et al.'s study of command usage in Eclipse found similar usage characteristics for OpenType, ranking the tool as the least used navigation command by developers using the Eclipse IDE (Murphy et al. (2006)). The fact that not many developers used NavigateTo indicates that likely many of them are not very familiar with the tool; unfamiliarity has been reported as a problem with many IDE features in Murphy-Hill et al. (2012). The fact that the two developers used NavigateTo in one isolated instance, while using other search tools both before and after, is perhaps indicative of the lack of flexibility of this tool.

To investigate the amount of searches in the Sando dataset that could have been answered by NavigateTo, we computed whether the query string could be found verbatim within the name (e.g., method name or class name) of a result opened in the Visual Studio editor, which is triggered by a double click on a result in Sando. 1074/8052 (or only 13.34%) of Sando queries retrieved a result that exactly matched the query that was provided by the user.

Most project-scope queries consist of one plain word term.

The left graph in Figure 6 presents the distribution of number of terms per query in the Sando dataset. The graph shows that when searching code, developers who used Sando for code search overwhelmingly issued simple, single plain term queries. We define a single plain term as a space-delimited sequence of characters that does not contain any punctuation symbols or a camelcase pattern (i.e. a lower case followed by an upper case letter). This query writing behavior is similar to that of Internet search engine users, where one term queries are also the norm (Baeza-Yates and Ribeiro-Neto (1999)). One explanation for this behavior is provided by the berry-picking model for Internet search, which suggests that searchers often issue a shorter general query followed by a refinement of that query by adding more terms if the initial result set was unsatisfactory (Bates (1989)). Another explanation is that users are searching for a known place of the code, that they have visited before, and therefore are constructing a specific query that is unlikely to yield many ambiguous results (Baeza-Yates and Ribeiro-Neto (1999)). In certain cases, code search tool researchers have assumed that developers issue longer queries than what we have observed in the field. This is exemplified by the frequent use of commit messages as a proxy for queries when researchers evaluate code search tools (Dit et al. (2011)), even though they are commonly sentence-like in length and structure.

About one in ten project-scope queries were part of a query reformulation sequence, where the developer usually followed a strategy of adding a single term.
To investigate query reformulation behavior, we examined the Sando dataset to identify query sequences where the user reformulated a previous query by adding or removing terms. We used a word similarity measurement between consecutive queries, where we computed the number of words that were similar in the two queries. Out of the 8,052 queries, 672 (or 8.35%) were part of a reformulation query sequence. Most of the query reformulation sequences were short, consisting of one reformulated query while a few reformulation sequences extended to 4-5 query reformulations.

The right graph in Figure 6 shows the distribution of the number of terms difference between an initial and a reformulated Sando query. A larger proportion of reformulated queries followed a strategy of increasing the number of terms in the query than reducing or maintaining the same number of terms, most often by one more term. However, there was still a number of reformulated query sequences where one term was modified (88), leaving the total number of terms in the query unchanged.

4.2 Findings on Complex Feature Location Sessions

There is no single feature location tool available to software developers, so developers often exhibit complex patterns of tool usage when locating features in the code, commonly relying on combinations of code search, structured navigation and debugging tools in the IDE. To discover complex feature location sessions that incorporate a variety of IDE tools and last a significant amount of time, we leverage a feature location mental model constructed by Wang et al. in a controlled laboratory setting (Wang et al. (2011)). This mental model is detailed enough to be used as a basis for the interpretation of the data we collected in the Blaze study. The model includes four different stages: *Search*, *Extend*, *Validate*, and *Document*. During the *Search* phase, the developer locates an initial point in the source code. The developer explores the surroundings of that initial point by exploring the call graph using static or dynamic means in the *Extend* phase, building an understanding of the code feature, which can be confirmed during the *Validate* phase. The *Document* phase records in writing the results of feature location. This phase is not crucial to the model and irrelevant to the field data that we analyze.

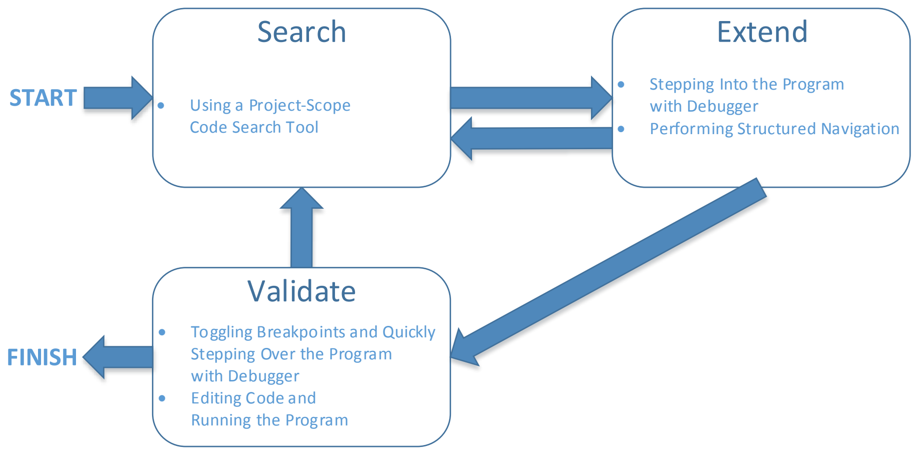
Search
START
Using a Project-Scope Code Search Tool
Extend
Stepping Into the Program with Debugger
Performing Structured Navigation
Validate
FINISH
Toggling Breakpoints and Quickly Stepping Over the Program with Debugger
Editing Code and Running the Program

Fig. 7 The feature location model proposed by Wang et al., adapted to field data by removing ambiguous paths through the model.

Figure 7 shows an adaptation of Wang et al.'s model to field data that takes into account, and removes, feature location behavior that is ambiguous and may confound analysis (e.g., debugging could be used to fix bugs or for comprehension so we remove sessions that use debugging in the *Search* phase). We use this adapted model to classify feature location behavior in the field, while trading off the ability to recognize certain feature location patterns that are more ambiguous and difficult to determine.

A developer may or may not progress through all of the model's stages for any feature location task. For instance, simple lookup-type feature location can usually be accomplished using only a single, rapid code search tool interaction. However, the complex exploratory tasks that we are interested in here require progressing to several stages in Wang et al.'s model, which in turn may require using several different IDE action modes, including searching, navigation and debugging.

We analyzed our Blaze dataset for such complex sessions, extracted using the previously used hierarchical agglomerative clustering algorithm, constrained to last for longer than one minute. We explored the extracted sessions to investigate recurring patterns in how these complex feature location tasks were performed by the developers in the Blaze dataset.

Our analysis produced 206 complex feature location sessions by 33 of the original 67 developers in the Blaze dataset. Sessions that proceeded past the *Search* phase were deemed complex in this analysis. The majority of such sessions 196/206 (95%) proceeded to the *Extend* phase of the feature location model in Figure 7. Only 10/206 (5%) of these sessions proceeded to the *Validation* phase.

Structured navigation is the most common *Extend* phase strategy during feature location.

Structured navigation, including viewing the call hierarchy of a method, viewing the type hierarchy of a class, and navigating to the definition of a method or class from their use, is the most frequent strategy taken by developers following

Table 1 Distribution of navigation commands in the *Extend* phase of feature location sessions.

Command	Number of Occurrences	Percent of Occurrences
Go To Definition	315	47%
*Prodet Call Graph Navigator	202	30%
Find All References	115	17%
ReSharper Find Usages	27	4%

a search in the analyzed complex feature location sessions. There were 190/196 (97%) sessions that used navigation, while the remaining 6/196 (3%) sessions relied on debugging in the *Extend* feature location phase.

We further analyzed the distribution of the kinds of navigation commands employed by the developers. Table 1 shows the results for commands with more than 1% of overall occurrences, indicating that `Go To Definition`, which navigates from a use to a definition of a program element, gets the most use. The `Prodet Call Graph Navigator`, a call graph exploration tool proprietary to ABB, Inc., is second most frequent. However, its popularity relative to other navigation commands is deceptive, since exploring a call graph in Prodet often requires several clicks for a single session. `Find All References` is also a popular IDE command that displays a list of uses for a specific identifier. Finally, the `ReSharper Find Usages` command is the `Find All References` command equivalent in the popular ReSharper toolkit for Visual Studio.

Developers in complex feature location sessions often switched between two different tool modalities.

A number of complex feature location sessions (41/206 or 20%) exhibited repetitive use of several tools and commands from distinct modalities (i.e. code search, structured navigation, debugging). For instance, there were 35 sessions where the developers frequently alternated between project-scope search and structured navigation and 6 sessions where the developers alternated between search and debugging. The alternating pattern was significant, consisting of an average of 2.9 repetitions of search and navigations and 3.6 repetitions of searching and debugging.

One possible reason for these patterns is that the developer realizes after entering the *Extend* phase of feature location that the *Search* phase was inadequate. Another possibility is that since the search, navigation, and debugging tools do not share information, the developer has to repeat previous steps when forgetting specifics about the current feature location task.

5 Implications of Study Findings

Our field study has focused on the frequency and types of code search tools used, developer behavior surrounding code search, types of queries issued by developers, and complex feature location patterns. The results that we have presented validate several prior studies in software maintenance. For instance, the brief nature (containing only one query term) of code search queries was first reported in Bajracharya and Lopes (2012) and Damevski et al. (2014). We confirmed this finding in the IDE setting, whereas Bajracharya and Lopes (2012) used a web-based, many-code base search engine, and at a much larger scale than Damevski et al. (2014). Helping developers easily produce longer queries is beneficial to code search tool developers. One possibility for this is using a recommender to suggest related terms to the developer based on the currently written term, an approach previously suggested by Yang and Tan (2012) and Howard et al. (2013). Similarly, the CONQUER tool by Roldan-Vega et al. (2013) uses a variety of natural language based suggestions for query refinement. An encompassing strategy for pre and post code search recommendations that includes a natural language approach, among others, was previously suggested by Ge et al. (2014) and implemented in recent versions of Sando. An automated approach to query reformulation which does not involve developer feedback was also shown to be an effective strategy in Haiduc et al. (2013). The developer's intent when issuing a brief, single-term query can be one of the following: 1) looking up an existing code element whose name is completely or partially known; or 2) exploring a completely unknown concept in the code base. An area of future work could be to automatically detect which of these actions is performed by the developer and target recommendations for that specific purpose, instead of displaying many recommendations that may polute the interface.

The consistent usage of code search tools in daily developer work indicates that increasingly developers are unable to remember all of the code for the projects they work on. Since developers are using such tools with significant consistent frequency, continued efforts by the research community to produce helpful support for code search are justified. To our knowledge, our study was the first to track developer behavior over longer periods of time and establish the frequency of use of code search tools by developers in the field.

Good code search tools are important to successful feature location. However, many instances of developers using code search tools are not centered around conventional feature location tasks, but smaller lookup-type developer activity intended to navigate to a known place in the code. In our study, this can be observed by the overwhelming frequency of one term queries in Sando and copy and paste command usage prior to all code search tools. Though this type of behavior was prevalent, a tool specifically crafted for a large class of those lookups (i.e. lookup of method and class names) was almost never used by the developers in our study. The likely causes for this behavior can be several. One may be unawareness of the existence of this tool. Many IDE tools are rarely used by developers, often because they are unaware of their capabilities or potential productivity improvements until receiving a recommendation about a tool from a fellow developer (Murphy-Hill et al., 2012). Another potential cause is the availability of a number of code search tools to the developers in our study that are more flexible than NavigateTo. The developers may be biased towards code search tools that can handle many different types of queries, rather than specific and highly-tailored ones. Since the two developers that used NavigateTo once, never used it again while making use of other search tools, the inflexibility of NavigateTo seems like a more likely reason for its lack of use.

Our study showed that query reformulation occurred in nearly 10% of submitted Sando queries. The implied "berry-picking" search behavior starts with more general queries and, if unsuccessful, are followed by more specific queries that may further limit the search tool's retrieved result set. Many code search tools proposed by researchers are not designed with the flexibility to handle this behavior, but are optimized for retrieval quality on idealized sets of queries that are very specific in nature. An example of this is the evaluation strategy researchers have often used for proposed feature location tools, which relies on one-shot retreival quality measurements, like precision and recall. Unlike many others, the MFIE tool by Wang et al. (2013) proposes an approach that integrates multiple facets extracted from the code (using the inheritance hierarchy, package structure, call graph, etc.) that allow for the iterative exploration of the results of an initial keyword query and the query's subsequent reformulation or extension. The challenge of approaches like MFIE is to select the right type and amount of information to display to the developer that allows for the quick and effective examination of the search results, without excessive cognitive load.

In complex feature location sessions, it is common, and known to researchers, that developers use multiple tools to comprehend the source code. Developers often switch between these tools (e.g. between search and navigation), interacting with both tools in their information quest. In most IDEs, these tools have no integrated capabilities that could speed up the developers' workflow. Such integrative behaviors would likely benefit developers in reducing effort as well as the cognitive load of remembering relevant information when transferring between the separate tools. Tools such as Mylar (Kersten and Murphy (2005)) and Eclipse's Mylyn partially solve this issue by maintaining a task context consisting of relevant program elements. However, tool developers can do more to integrate code search and navigation into a similar interface, such as providing structural information or the ability to perform structural exploration directly from the search engine interface.

6 Conclusions and Future Work

In this paper, we investigated developer behavior during feature location using two field datasets, one reflecting developer actions in the IDE and the other developer interactions with a code search tool. Both datasets presented unique challenges for analysis as the developers were not directly observed during their daily work, which however offered a unique perspective.

Our analysis found that developers often rely on code search tools, but that they most often issued quick lookup-style queries to navigate to a presumably known place in the code base. When performing more complicated feature location, searching for a previously unknown set of program elements, developers often used a "berry picking" approach of continuously reformulating their query by adding or removing terms. They also sometimes relied on structured navigation or the debugger following a textual search to better comprehend a specific area of the code base. Our study suggests more work in query recommendation specifically targeted to either lookup style or exploratory style code search, work in studying the relevant information necessary to developers when evaluating results during an exploratory search session, and work on better integration of code search with navigation and debugging tools within the IDE.

Future empirical work includes analysis of the specific patterns developers exhibited when using a navigation or debugging tool following code search. It also includes further data collection to support the conclusions of this study and explore other micro-patterns exhibited by developers during software maintenance.

Acknowledgements The authors gratefully acknowledge developers at ABB, Inc. and users of the Sando search tool who allowed anonymous data collection during their daily work. We also acknowledge Will Snipes for collecting and sharing the Blaze dataset and data collection tool with us.

References

Baeza-Yates RA, Ribeiro-Neto B (1999) Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Bajracharya SK, Lopes CV (2012) Analyzing and mining a code search engine usage log. Empirical Software Engineering 17(4-5):424–466

Bates MJ (1989) The design of brosing and berrypicking techniques for the online search interface. Online Information Review 13(5):407–424, DOI 10.1108/eb024320, URL http://ci.nii.ac.jp/naid/80004823012/en/

Damevski K, Shepherd D, Pollock L (2014) A case study of paired interleaving for evaluating code search techniques. In: Proceedings of the IEEE Conference on Software Maintenance and Reengineering - Working Conference on Reverse Engineering (CSMR-WCRE)

Dit B, Moritz E, Poshyvanyk D (2011) A tracelab-based solution for creating, conducting, and sharing feature location experiments. In: IEEE International Conference on Program Comprehension

Ge X, Shepherd D, Damevski K, Murphy-Hill E (2014) How the sando search tool recommends queries. In: Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on, pp 425–428

Haiduc S, Bavota G, Marcus A, Oliveto R, De Lucia A, Menzies T (2013) Automatic query reformulations for text retrieval in software engineering. In: International Conference on Software Engineering (ICSE)

Howard MJ, Gupta S, Pollock L, Vijay-Shanker K (2013) Automatically mining software-based, semantically-similar words from comment-code mappings. In: Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press, Piscataway, NJ, USA, MSR '13, pp 377–386, URL http://dl.acm.org/citation.cfm?id=2487085.2487155

Kersten M, Murphy GC (2005) Mylar: A degree-of-interest model for ides. In: Proceedings of the 4th International Conference on Aspect-oriented Software Development, ACM, New York, NY, USA, AOSD '05, pp 159–168, DOI 10.1145/1052898.1052912, URL http://doi.acm.org/10.1145/1052898.1052912

Ko AJ, Myers BA, Coblenz MJ, Aung HH (2006) An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans on Soft Eng 32(12):971–987

Manning CD, Raghavan P, Schütze H (2008) Introduction to Information Retrieval. Cambridge University Press, New York, NY, USA

Murphy GC, Kersten M, Findlater L (2006) How are java software developers using the eclipse ide? IEEE Software 23(4):76–83, DOI 10.1109/MS.2006.105, URL http://dx.doi.org/10.1109/MS.2006.105

Murphy-Hill E, Parnin C, Black AP (2009) How we refactor, and how we know it. In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '09, pp 287–297, DOI 10.1109/ICSE.2009.5070529, URL http://dx.doi.org/10.1109/ICSE.2009.5070529

Murphy-Hill E, Jiresal R, Murphy GC (2012) Improving software developers' fluency by recommending development environment commands. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, New York, NY, USA, FSE '12, pp 42:1–42:11, DOI 10.1145/2393596.2393645, URL http://doi.acm.org/10.1145/2393596.2393645

ReSharper :: The Most Intelligent Extension for Visual Studio (2014) `http://www.jetbrains.com/resharper/`

Robillard M, Coelho W, Murphy G (2004) How effective developers investigate source code: an exploratory study. IEEE Transactions on Software Engineering 30(12):889–903

Roldan-Vega M, Mallet G, Hill E, Fails JA (2013) Conquer: A tool for nl-based query refinement and contextualizing code search results. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, ICSM '13, pp 512–515, DOI 10.1109/ICSM.2013.84, URL http://dx.doi.org/10.1109/ICSM.2013.84

Shepherd D, Damevski K, Ropski B, Fritz T (2012) Sando: an extensible local code search framework. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE, pp 15:1–15:2

Sillito J, Murphy GC, De Volder K (2006) Questions programmers ask during software evolution tasks. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, SIGSOFT '06/FSE-14, pp 23–34, DOI 10.1145/1181775.1181779, URL http://doi.acm.org/10.1145/1181775.1181779

Wang J, Peng X, Xing Z, Zhao W (2011) An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. In: Software Maintenance, IEEE Int. Conf. on, IEEE, pp 213–222

Wang J, Peng X, Xing Z, Zhao W (2013) Improving feature location practice with multi-faceted interactive exploration. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '13, pp 762–771, URL http://dl.acm.org/citation.cfm?id=2486788.2486888

Yang J, Tan L (2012) Inferring semantically related words from software context. In: Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on, IEEE, pp 161–170

A Appendix: List of Relevant Events in Blaze and Sando Datasets

Table 2 Blaze Dataset Events

Category	Event	Number of Occurrences
Navigation	Edit.GoToDefinition	9681
	View.Find Symbol Results	4866
	View.Code Navigation Window	4175
	Edit.FindAllReferences	1719
	ReSharper.ReSharper_FindUsages	634
	View.Call Hierarchy	100
	View.CodeMap1.dgml	30
	EditorContextMenus.CodeWindow.CodeMap	3
	Debug.ShowCallStackonCodeMap	1
Edit	Edit.BreakLine	67592
	Edit.Delete	60685
	Edit.Paste	46284
	Edit.Undo	24632
	Edit.InsertTab	13996
	Edit.Cut	8428
	Edit.CommentSelection	1787
	Edit.FormatDocument	989
	Edit.FormatSelection	185
Debug	Debug.Debug Run Mode	181290
	Debug.Debug Break Mode	174110
	Debug.StepOver	102566
	Debug.Start	57689
	Debug.StepInto	35402
	Debug.ToggleBreakpoint	7536

Table 3 Sando Dataset Events

Category	Event	Number of Occurrences
Solution Level	Solution opened	12839
Query Level	Recommendation item selected	54027
	Query submitted by user	5856
	Sando returned results	5856
Result Set Level	User previewed a result	21933
	User clicked a result	12172