Developers' Code Context Models for Change Tasks

Thomas Fritz† fritz@ifi.uzh.ch
David C. Shepherd* david.shepherd@us.abb.com
Katja Kevic† kevic@ifi.uzh.ch
Will Snipes* will.snipes@us.abb.com
Christoph Bräunlich† christoph.braeunlich@uzh.ch

†Department of Informatics
University of Zurich
Switzerland

*Industrial Software Systems
ABB Corporate Research
Raleigh NC USA

ABSTRACT

To complete a change task, software developers spend a substantial amount of time navigating code to understand the relevant parts. During this investigation phase, they implicitly build context models of the elements and relations that are relevant to the task. Through an exploratory study with twelve developers completing change tasks in three open source systems, we identified important characteristics of these context models and how they are created. In a second empirical analysis, we further examined our findings on data collected from eighty developers working on a variety of change tasks on open and closed source projects. Our studies uncovered, amongst other results, that code context models are highly connected, structurally and lexically, that developers start tasks using a combination of search and navigation and that code navigation varies substantially across developers. Based on these findings we identify and discuss design requirements to better support developers in the initial creation of code context models. We believe this work represents a substantial step in better understanding developers' code navigation and providing better tool support that will reduce time and effort needed for change tasks.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Human Factors

Keywords

Context models, search, navigation, change task, user study

1. INTRODUCTION

Software developers spend substantial time searching and navigating through code to understand relevant parts of a system for a particular change task [24, 37]. During this process of understanding and then changing code, developers implicitly build *code context models* that consist of the relevant code elements and the relations between these elements, often more generally referred to as task context. Since these models mainly stay implicit in developers' heads and are not persistent, developers have to continuously spend a significant amount of their time creating context models for newly assigned change tasks from scratch [26].

Researchers have suggested that more explicit context models for change tasks[1] can be used to support developers in their work [29]. To form these explicit context models, existing approaches have used methods ranging from a developer manually specifying the context (e.g., [33]) to automatically inferring the context from a developer's interaction with a development environment (e.g., [22]). While these approaches have been shown to support developers with change tasks, little is understood about the implicit code context models that developers build and their characteristics. With a better understanding of the characteristics of developers' code context models, we might be able to help developers in the initial creation of these models for change tasks, saving time and effort.

To investigate the characteristics that code context models exhibit for different change tasks, we conducted an exploratory study with twelve developers on three change tasks in open source projects. To validate our observations on a broader population of developers and tasks, we conducted a second empirical analysis using data sets from several hundreds of change tasks and eighty developers working on open and closed source projects. Amongst other results, our studies show that developers' context models are highly connected, structurally and lexically, that the code navigation can differ substantially by individual even for the same change task, and that developers start change tasks using a combination of search and navigation and then frequently revisit code elements. Based on our findings we infer design requirements to support developers in the creation of code context models and discuss the design of such an approach.

This paper makes the following research contributions:

- It identifies important observations on the characteristics of code context models based on an exploratory study with 12 developers on three open source projects.
- It provides an empirical analysis of the findings on data collected from eighty developers working on a variety of change tasks on open and closed source projects.
- It identifies design requirements and discusses the design of an approach to support developers in the creation of code context models.

[1]We use the term change task to refer to both modification task and bug.

FSE'14, November 16–21, 2014, Hong Kong, China
ACM 978-1-4503-3056-5/14/11
http://dx.doi.org/10.1145/2635868.2635905

This work represents a substantial step in better understanding developers' code navigation and implicit context models for change tasks that will help to provide better tool support with the potential for real-world impact on the development process, in particular on reducing the time and effort required for change tasks.

2. EXPLORATORY STUDY

In the process of performing a change task, developers build up *code context models*[2]—code elements and relationships between these elements that are *relevant* to the change task. In this study, we investigate these code context models based on three specific concepts of relevance: (a) relevance as perceived by the developer, (b) relevance as defined by the actual code change and (c) relevance as defined by the explicit navigation activity of the developer. Since our ultimate goal is to support code context creation we also investigated developers' code navigation tendencies to understand how code context models could be created. In particular, we wanted to address the following questions:

(1) What are common characteristics that code context models exhibit?

(2) How do code context models vary based on different definitions of relevance?

(3) How do developers' code context models and navigation behavior vary for different tasks?

To investigate these questions, we conducted an exploratory study with a blocked subject-project study setup [13] with twelve software developers. Each developer worked on one of three different change tasks for open source projects.

2.1 Study Method

For this experiment we chose three open source systems in Java that all have an open task repository, recent development activity and a code base big enough to preclude a systematic understanding of the entire system. Specifically, we chose FreeMind [6], Java PasswordSafe (JPass) [7] and Rachota [8]. For each system we chose one open change task that two of the authors were able to perform in less than one hour. Furthermore, we chose tasks for which the change could be observed in the graphical user interface. All three change tasks were reported as bugs, however, the JPass and FreeMind tasks could have been categorized as enhancement or modification task. Thus, we will mostly refer to all three tasks with the more general term *change task*.

Each developer who signed up for the study was randomly assigned to one of the three tasks. We then provided each participant a document with instructions and access to a virtual machine that was set up with an Eclipse IDE[3] and a workspace that contained the assigned change task description and project. We decided to set up a virtual machine for each participant on the Amazon Elastic Compute Cloud [2] to allow for remote and independent access using his own computer setup and thus to affect the "normal" behavior as little as possible. The participants were instructed to first run the application and observe the current behavior to be changed before looking at the code and trying to perform the change. Furthermore, the instructions told the participants to answer a set of questions after either completing the change task successfully or after 75 minutes to limit the total time required of a participant to 90 minutes—75 minutes for the change task and 15 minutes for the questions.

In the questions, the participants were asked to sketch a model of the source code elements, such as classes, methods and fields, and the relationships they considered relevant for understanding and making the change. For the sketch, the participants were allowed to use pen and paper or their favorite drawing tool and they were encouraged to use any notation or form they wanted to. The rest of the questions in the set addressed the experience of the participants.

To make sure that the tasks are solvable in the given time and the questions are understandable by the participants, we conducted pilot studies with three graduate students, each performing one of the three tasks. The pilots confirmed our assumption on the timing and we only slightly altered the question on the model sketching part to explicitly state that developers are allowed to use pen and paper for the sketch.

2.2 Subjects

We recruited subjects through email and personal contact. To be eligible, subjects had to have experience programming in Java. We ended up with 12 participants that we randomly assigned to one of the three tasks, four for the FreeMind task (F1-F4), four for the Java PasswordSafe task (J1-J4) and four for the Rachota task (R1-R4). Of these twelve developers, five worked in a company, four were graduate students and three faculty members in Computer Science, all with a background in software engineering. The subjects' programming experience ranged from 8 to 16 years (average of 11.8) with between 0 to 12 years (average of 4.5) of professional programming experience. For each task we made sure to have at least two developers with professional development experience and one graduate student to report on. Two of the subjects were female, ten male. On a five point Likert-scale with 1 (strongly disagree) to 5 (strongly agree), all subjects agreed or strongly agreed that Java is one of their primary programming languages (average of 4.75), and were very familiar with the Eclipse IDE (average of 4.17).

2.3 Projects and Change Tasks

FreeMind. The FreeMind project (version 0.9.0 RC 15) is an open source mind map editor consisting of 52.5k non-commented lines of code (NCLOC), 439 top level classes, and 45 packages. We selected a task for this project (ID 3420227, [10]) that was still open and observable. This change task addressed FreeMind's failure to save a map after an encrypted node was added as well as the inadequate notification upon failure. We limited the scope of this potentially large change by asking the subjects to add a reasonable explanation to the "Save Failed" dialog. This change required users to propagate exception information from the `save` method of `EncryptedMindMapNode` to the user action (`actionPerformed` in `SaveAction`) and finally to display the improved message to the user. The call chain between `actionPerformed` and `save` is relatively long (11 method calls in total) and can be challenging to follow. Fortunately, when reproducing the failure, which we asked all subjects to do before making the change, a stack trace was printed to the console that contained the relevant call chain.

Java PasswordSafe. The Java PasswordSafe (JPass) project (version 0.8 final) is an open source password management system consisting of 13.5k NCLOC, 167 top level classes,

[2]Murphy *et al.* ([29]) introduce the broader term *task context* that extends our notion to arbitrary artifacts.

[3]Eclipse IDE for Java Developers, version 3.7, `eclipse.org`

and 18 packages. We again selected a change task (ID 2933526, [11]) that was still open at the time and observable. This task addressed the lost selection and undesired scrolling that occurs when the application is unlocked after coming out of the sleep state. For this change task, we expected subjects to save the selection index in order to reselect and center the appropriate item after the application was unlocked. While classes `UnlockDbAction`, `LockDbAction`, and `PasswordSafeJFace` were all relevant for this task, only one to two methods in `PasswordSafeJFace` needed to be changed. During this task, subjects familiar with the Standard Widget Toolkit [3] may have benefitted, although prior knowledge was not necessary. This application also made extensive use of console logging. Observant developers could use these log messages as a starting point for searches.

Rachota. The Rachota project (version 2.4) is an open source time tracking utility where users can track the time spent on each task. It consists of 18k NCLOC, 53 top level classes, and three packages. We selected a task (ID 2658881, [9]) that was open and observable. This task addressed the problem of newly created tasks failing to show in the 'History' tab. For this task, we expected subjects to trigger an update of the History tab's underlying model upon task creation. Three classes that are relevant to the task are extremely large (`HistoryView` has 1800 NCLOC with 42 methods, `MainWindow` has 1125 NCLOC with 28 methods and `DayView` has 1807 NCLOC with 50 methods). These large classes, along with the fact that the application offered no logging or relevant stack trace made it more difficult for users to find a starting point in the code base.

2.4 Data Collection and Analysis

We used a combination of qualitative and quantitative methods motivated by the ones described by Seaman [34]. We used participant observation by recording each participant's screen and having access to their actual workspace after the session, in addition to asking the participant a set of questions. From the participants' sessions we collected three types of data: patches for the successful completion of the change task, videos capturing the developers' screens during their work on the task and the artifacts that contained the answers to the questions, including the sketched models. To record a developer's screen, we automatically started a screen recording application at the beginning of a developer's session. For the questions, we asked developers to send us their answers by email after they finished. We transcribed and coded the patches, the screen recordings and the collected answers. The transcripts together with further study artifacts are available at [5].

From the videos we determined the time that each participant took to complete a task. We chose the point at which a participant validated the correctness of his change in the user interface of the application as the finish time. Even though the instructions stated that participants should move onto the questions part after 75 minutes to limit the total amount of effort spent, three participants chose to continue. Two of these three participants, J4 and R4, did not succeed in performing the appropriate change and at some point stopped working on it. Both participants closed Eclipse at the end which we used as the finish time. Table 2 presents the time participants took to complete the task or until they stopped.

[4]This was only used in the FreeMind task and opened up the parent class.

Table 1: Developer Navigation Steps Transcribed from the Screen-Captured Videos (several of these refer to tool support provided in the Eclipse IDE).

Structured Navigation Steps	
navigation aids	call hierarchy, type hierarchy, find references
debugger	step into, step return, stacktrace click
editor	quick documentation, open declaration, quick fix[4]
Unstructured Navigation Steps	
package explorer	expand item, open item
search	Java, file, find in file, outline view
editor working set	back, forward, open from editor tab
editor	scan

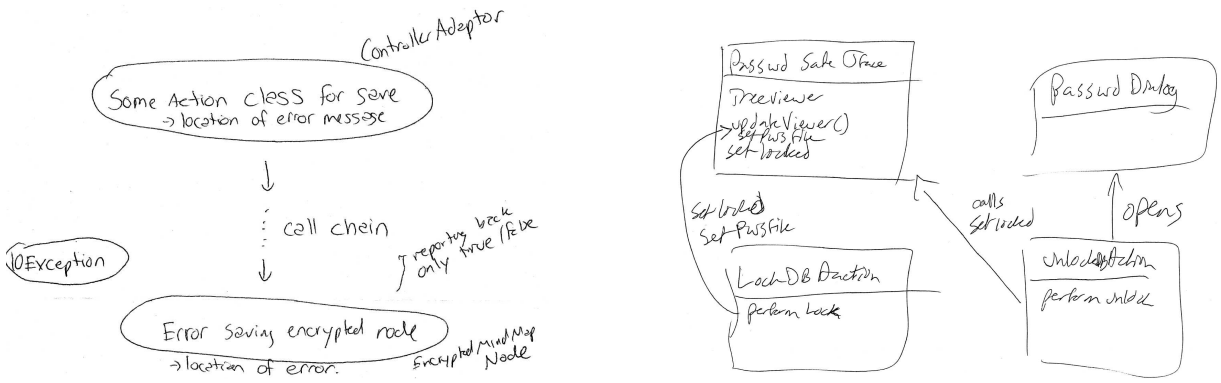
For investigating and comparing the three different code context models, we determined the source code elements and relations in these models from the data collected.

Developer Models. For the code context models based on the developer's relevance definition, which we refer to as *developer models* in the following, we coded the models sketched by the developers. We acknowledge that these sketches may not be a complete or accurate representation of developers' implicit context models, thus necessitating the use of complementary models, in particular the code navigation model. We believe, however, that these sketches encode important or prominent features of these implicit models of which developers are conscious. For each sketch, we determined the code elements the sketches explicitly referred to. Since all twelve models contained references to classes but four did not contain any methods and three did not contain any fields, we only examine the classes used in these models for a fair comparison in the following.

Patch Models. For the code context models from the actual patch, which we refer to as *patch models*, we determined the classes and methods that were changed as well as the types that were used and the methods that were called in the actual change.

Code Navigation Models. For the models defined by the developer's explicit navigation behavior, which we refer to as *code navigation models*, we transcribed the screen recordings and coded the resulting transcripts. Since we are interested in the navigation of a developer through the program code, in particular the classes and methods, we transcribed the structured and unstructured navigation steps a developer took. We considered a navigation structured if a developer explicitly navigated from a code element A to a code element B along a structural relation using some tool support in the IDE, where code elements were defined as classes, methods or fields and structural relations referred to call, implements and usage relations. Table 1 presents a summary of all transcribed navigation steps. For each step we recorded the step, the target element and its type as well as, in case of a structured step, the source element, its type and the relation followed.

The navigation steps that we transcribed do not explicitly capture the code editing by a developer. However, since we think it is reasonable to assume that a developer has an understanding of the elements he uses or calls in his code change, we added these elements to the code navigation model if they were not yet in it, which was rarely the case.



(a) F2's Developer Model

(b) J1's Developer Model

Figure 1: Developer Models for FreeMind and JPass.

For transcribing code navigation one has to determine which code elements a developer is examining at any point in time, which is challenging as described by [32]. While transcribing the video, we used the mouse pointer as a clue and examined the actual code base to determine which method a developer was in. We also used the keyboard events to determine the tool support a developer used. Observational studies are subject to observer bias which may lead to omitting navigation instances or characterizing them incorrectly. To mitigate this risk, we had an initial phase in which three investigators transcribed one video and cross-validated the results to make sure no major differences occurred. After this phase, one investigator transcribed all videos and random samples were picked for cross-examination by the other two investigators revealing no major differences.

2.5 Study Results

Based on the analysis of the qualitative and quantitative data we gathered, we made several key observations with respect to the three questions we set out to study. Given the exploratory nature of our study, we will discuss these observations and their implications mainly alongside the presentation of descriptive statistics. In the presentation of the observations we only included the successful subjects (ten of the twelve subjects) since we did not have patches for the unsuccessful subjects. A summary of the statistics gathered is presented in Table 2.

O1—Developer models are small, abstract and highly connected. Across all subjects and tasks, developer models are consistently small with a mean (M) of 4.6 class elements (standard deviation SD of 0.8). Even though the size of patch models showed big variances for different tasks (M = 14.5 classes for FreeMind, M = 3.7 for JPass and M = 2.3 for Rachota) and the code navigation models varied widely across all subjects on the class level (overall SD = 8.5 with M = 17.4), the size of the developer models remained consistently small.

Developers generally used abstraction in their models. Instead of using concrete class names developers recorded the concepts or functionality they were interested in, e.g., subject F2 used "some action class for save, location of error message" to denote the class `ControllerAdaptor` (see Figure 1(a)) and J2 stated "Main View" and put the actual class name in brackets close by. However, the level of abstraction used in the models varied by subject and task. For instance, all four FreeMind subjects used a very high level of abstraction, whereas subjects on the JPass task included more detail in their models. An example to illustrate this difference is shown in Figure 1; Figure 1(a) shows F2's developer model for the FreeMind task which is highly conceptual, abstracting from direct call relations to transitive call chains, and Figure 1(b) shows the model of developer J1 on the JPass task which resembles a class diagram with details of the code.

All developer models were also highly connected. In fact, all models were fully connected at class-level excluding one class element in subject F2's otherwise connected model. On average, there were 5.3 relations in a developer model (SD = 2.1) and these relations mainly referred to method calls, but also to contains and inheritance relations.

O2—Patch model size has little influence on the size of code navigation models. For the three tasks we investigated, the average number of methods in the patch model had almost no influence on the number of methods in the code navigation model. The patch models for the FreeMind task were the largest and the most scattered, containing an average of 22.2 methods (SD = 3.0) over 14.5 classes. The patch models for the JPass task and the Rachota task were both much smaller (M = 6.7, SD = 4.5 and M = 4.7, SD = 4.6 respectively) as well as less scattered (3.7 and 2.3 classes). In spite of the bigger patch models, the code navigation models for FreeMind were smaller than the ones for Rachota, with an average of 35.2 methods (SD = 7.3) in the FreeMind models and 43.7 (SD = 30.1) for Rachota. A similar lack of correlation is seen when, in spite of patch models of roughly equal size, JPass's code navigation models were on average a lot smaller (M = 22.3, SD = 8.8) than Rachota's. This can also be seen in the Pearson's correlation coefficient between the patch and the code navigation model size being close to zero overall with r = .006. This observation implies that *a bigger and more scattered change does not result in a developer navigating through more method elements to make the change.*

O3—Even for concise and successful changes, code navigation models can differ substantially on class as well as method level. Code navigation models can vary substantially across developers, even for tasks that require only small changes. For example, while there was some agreement on four core classes for the JPass task, *i.e.*, all four classes were in all navigation models and three of these four were in all developer models, there was a wide variance outside of these four classes. The three subjects had between 6 and 19 classes in their navigation models with an

Table 2: Summary of Descriptive Statistics on Participants' Background and Exploratory Study (*pro* = *professional*, *grad* = *graduate student*, *fac* = *faculty*, ✓ = *success*, ■ = *failure*, *Cl* = *classes*, *Me* = *methods*, *Deb* = *debugging*).

Project	*ID*	*Job*	*Years Pr.Exp.*	*Time & Success*	*Dev. Model Cl*	*Patch Model Cl*	*Patch Model Me*	*Code Nav. Model Cl*	*Code Nav. Model Me*	*Navigation Steps All*	*Navigation Steps Structured*	*Navigation Steps Revisits*	*Navigation Steps Deb*
Freemind	**F1**	pro	10	39.7min ✓	4	16	25	37	42	116	101	47	26
	F2	pro	11	22.0min ✓	4	15	23	16	25	106	57	40	54
	F3	grad	8	59.7min ✓	5	11	18	19	36	341	229	250	219
	F4	grad	9	70.9min ✓	4	16	23	22	38	177	41	112	11
JPass	**J1**	pro	12	8.6min ✓	4	2	2	6	12	67	46	30	33
	J2	pro	12	64.5min ✓	4	6	11	19	28	408	279	305	250
	J3	pro	11	62.0min ✓	5	3	7	12	27	140	64	88	17
	J4	fac	12	101.1min ■	4	-	-	19	32	570	459	453	441
Rachota	**R1**	fac	15	53.8min ✓	6	5	10	20	31	349	101	248	78
	R2	grad	11	100.6min ✓	4	1	2	13	78	553	42	396	39
	R3	pro	15	36.6min ✓	6	1	2	10	22	326	130	241	160
	R4	fac	16	114.4min ■	9	-	-	16	35	282	76	155	6

average overlap of elements with at least one other subject's model of only 52.6%. On method level, the variance was larger as models ranged from 12 to 28 elements with only 3 methods that all three subjects had in common. In class `PasswordSafeJFace`, one of the core classes for this change task, the three subjects inspected 21 different methods but only one of these 21 methods was inspected by all subjects.

O4—Code Navigation Models are highly connected (structural cohesion). Upon inspecting all code navigation models for all successfully completed change tasks we found that, on average, 73% of the class elements in a code navigation model are connected with at least one other class through a call, usage or implements relation. Six of the ten code navigation models centered around one large connected group of classes and zero or more additional classes with no connections. By cross referencing these unconnected classes with the transcripts we observed that the unconnected elements were often visited towards the beginning of the task using unstructured navigation steps, prior to developers finding a point of reference to start a deeper, more structured investigation. For example, for the code navigation models of the three subjects on the JPass task, there are nine classes that are not connected to more than one other class and seven of these nine were navigated to within the first few steps. Figure 2 illustrates an example of a code navigation model for JPass, including a numbering to show the order in which elements were navigated to. In this example, most elements are connected except for some that the developer navigated to in the beginning and two elements from seemingly random selections later on (30 and 31).

O5—Navigation Sequences are largely determined by lexical similarities (lexical cohesion). In our transcripts we observed that developers often subsequently visited code elements that share identifying terms within their identifiers, in particular in the beginning of the investigation. Upon inspecting all subsequent visits, either from one class to another, one method to another one within the same class, or one method to a method in another class, we found that over all subjects and tasks, 43% (±17%) of the elements visited subsequently are lexically similar. In this paper, we define two identifiers as lexically similar if, after splitting up identifiers according to camelCase notation, they have at least one term in common. This result supports the observations made by other researchers (e.g. [27, 24]). We also found that developers pay overall more attention to lexical similarities when they switch from a method to a method in another

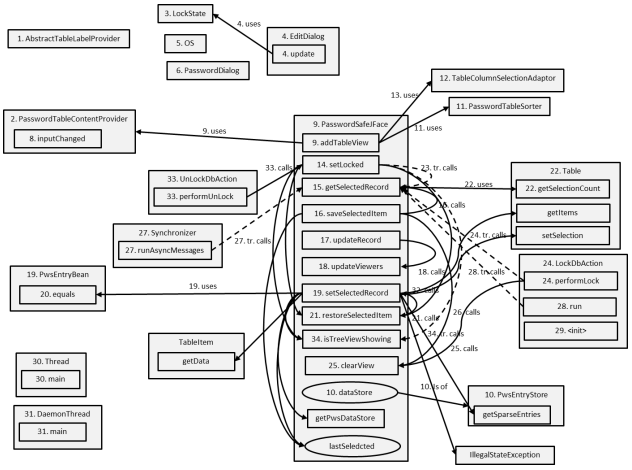


Figure 2: Code Navigation Model for Subject J2.

class. On average 69% of the method switches to a different class are lexically similar, whereas only 29% of the method switches within the same class are lexically similar.

O6—Developers start with a combination of search and structural navigation. In a study on a small project with 500 lines of code, Ko *et al.* [24] found that developers first search for information, then engage with the information to decide whether it is worth continuing by navigating the relationships between information, before finally editing the code. Similarly, Sillito *et al.* [36] identified that developers exhibit a behavior of 'finding initial focus points' and then 'building on those points' through navigation and exploration. Our exploratory study on the three projects corroborates these initial findings. Of the twelve subjects, nine performed an explicit search within the first 8 steps, and the other three found an initial starting point in the code by scanning the package structure rather than explicitly searching. Seven of the nine subjects that performed searches found starting points from the search. In these cases, within the next 10 steps they spent an average of 4.1 steps following call and execution relations from one of the search results and an average of 4.3 steps scanning one of the search results. More qualitatively, from the code navigation models generated, one can see that developers explored call, declaration and execution relationships a couple of steps out from search results, often revisiting the results and the intermediate elements once determining the relevancy of the element. This suggests that developers start their tasks with

a combination of a global search for information and then navigate the structural relations, in particular call relations, to comprehend more about the context of the elements.

O7—Developers frequently revisit code and take less time if their navigation is more structured. In their navigation, developers revisit elements more often than they navigate to new code elements. Over all subjects and tasks and including debugging steps, 68% of all navigation steps were revisits, with a mean revisit rate of 61.4% (SD = 14.7%) per subject. Not surprisingly, the more revisit steps a developer performed in his navigation, the more time he spent on the whole change task (Pearson's r = .72). Furthermore, the higher the ratio of structured versus unstructured navigation was, the less time a developer spend on the change task (Pearson's r = −.49). This result supports the observation that Robillard *et al.* [32] made on successful developers performing more structurally guided searches than unsuccessful ones, only that we look at the time of completing a change task rather than success.

3. EMPIRICAL ANALYSIS

To further investigate the characteristics and variations of code context models and validate the findings of our exploratory study, we conducted an empirical analysis of two data sets from several hundreds of change tasks and eighty developers working on open and closed source projects.

3.1 Data Sets

We used two data sets, denoted as Blaze and Mylyn data, for the empirical analysis. Both data sets capture developers' interactions, such as selects and edits, with an IDE. We used these two data sets for the different aspects they capture about a developers' work within an IDE, the different populations and IDEs they capture and their availability. While the Mylyn data contains information on the specific tasks developers worked on and the changes they committed for resolving these tasks, the Blaze data contains information on search instances and the exact order of events.

Mylyn Data.
The Mylyn data consists of change tasks, task contexts capturing interaction data for a change task, and patches of the Mylyn project. We chose to analyze the Mylyn project as it is a reasonable-sized project (466k NCLOC) that provides information on developers' interaction with the Eclipse IDE for a reasonable number of change tasks. The interactions are thereby captured using the Eclipse Mylyn project [12, 22]. From the Bugzilla [1] repository for the Mylyn project, we retrieved 9920 change tasks reported between 07/18/2007 and 02/20/2014. From this set, we filtered all change tasks that did not have a patch and a task context associated, resulting in a total of 2253 change tasks. For each change task, we extracted several features, such as the severity of the change task and the comments stored within the change task. By linking the change task ID to the commit comments in the change history of the Mylyn project, we identified the patches for each change task and extracted the changed classes and methods using ChangeDistiller [19]. In addition, for each change task we retrieved the associated task context and extracted the classes and methods a developer selected in the process of performing the change task.

For each of the 2253 change tasks in our Mylyn data, developers changed on average 5.7 (SD = 16.2) classes and 13.0 (SD = 69.2) methods and selected an average of 47.2 (SD = 139.7) classes and 18.8 (SD = 31.1) methods. The change tasks in this data set are categorized into different task types based on the severity field: 915 normal, 719 enhancement, 313 minor, 130 trivial, 127 major, 30 critical and 19 blocker. The time period of a task context, which denotes the time from the first captured code selection for the task to the last, varies a lot with an average of 18.5 days (SD = 87.3). Thus, it differs substantially from the short change tasks investigated in our exploratory study and most other similar empirical studies in related work. Overall, there were 31 developers, each working on at least one of the 2253 change tasks.

Blaze Data.
Blaze is a Visual Studio extension that logs interaction data, in particular all actions a developer performs within the IDE or that are executed by Visual Studio itself [38]. Blaze acts as a global event handler, listening for all GUI events within Visual Studio. For each event, Blaze records the key attributes, such as the name and type of the event. If existent, Blaze also records the file name and the currently selected line number. All events are registered along with an anonymized unique identifier for each Blaze user that allows to investigate differences between developers.

The Blaze data used in our analysis contains data recorded from 59 developers and over 8000 hours of development activity. It was collected from developers in ABB's globally distributed industrial software development community that volunteered to share their data.

3.2 Data Analysis and Results

To examine the characteristics of code context models and validate the findings of our exploratory study, we performed a set of analysis over the two data sets.

Mylyn Data.
To examine if the size of the patch influences the size of the task context (***O2***), we compared, for each change task, the changed classes and methods of the patches with the selected classes and methods within the task contexts. Figure 3 illustrates the high variance and lack of a general trend between the number of changed methods and the number of selected methods which is similar on class level. A Pearson's correlation coefficient between the number of changed and selected elements of r_{class}=.257 (p<.001) on class and of r_{method}=.202 (p<.001) on method level supports this lack of a trend. Both correlation coefficients are of small effect sizes (r<.3). The coefficient of determination, $R^2{}_{class}$=.066 for r_{class}, and $R^2{}_{method}$=.041 for r_{method}, measures the amount of the variability in the number of changed classes, respectively methods, that is shared by the number of selected classes, respectively methods. $R^2{}_{class}$ implies, that the number of changed classes and the number of selected classes share only 6.6% of variability and that 93.4% cannot be explained. On method level, only 4.1% of the variability is shared, denoting that 95.9% of the variability cannot be explained. These results support ***O2***.

O8—Type and discussion length of a change task can significantly influence code navigation models. Since our results show that the size of a patch has little influence on the navigation behavior, we investigated the influence of other aspects of a change task. In particular, we looked at the type and the discussion length of a change task, since these might be reflective of the complexity of a change task, assuming that complicated or unclear concepts often require

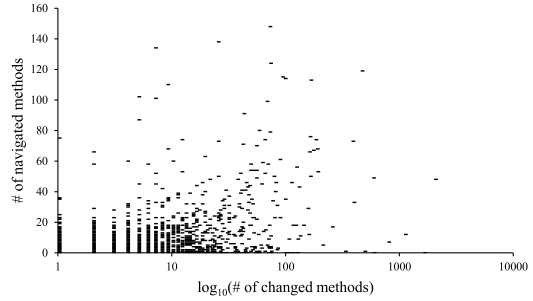


Figure 3: Number of Selected Methods Plotted Against Number of Changed Methods.

longer discussion threads. Therefore, we examined if we can predict the task context size as small or big on the basis of a change task type using multinomial logistic regression. We defined a task context as small if it contains less than the median size of all task contexts (8 different classes and 3 different methods) and big otherwise. A chi-square test results in *chi square*= 40.624 (p<.000 with df=6), showing that the predictive power significantly increases when the change task type is added to the model. Looking at the parameter estimates for all seven change task types allows us to interpret effects by comparing two change task types against each other. For example, the odds of having a small task context for a "trival" change task are 2.34 times higher than for an "enhancement" (with p<.001). When comparing all pairs of types, significant effect sizes exists for enhancement and trivial, enhancement and minor, enhancement and major, and enhancement and normal, showing that a change task of type enhancement usually results in a bigger task context. Our comparison of the number of comments within a change task and the navigation behavior, represented by the number of selected elements, shows that there is a correlation with a medium effect size with a Pearson's r=.30, p<.001 on class and Pearson's r=.38, p<0.001 on method level. At the same time, the number of comments and the patch size (number of changed elements) are only correlated with a small effect size with Pearson's r=.11, p<.001 on class and Pearson's r=.27, p<.001 on method level.

We also used the Mylyn data to further examine the observation that even for concise changes code navigation model can differ substantially (***O3***). Since in the history of a project, there are usually no two change tasks that are exactly the same and no two developer complete the same task, as we had it in our exploratory study, we used an approximation for concise changes. In particular, we approximated concise changes as the ones in the Mylyn data that changed at most 2 classes and 3 methods (the median of changed classes, respectively methods over all change tasks). After extracting these changes from the Mylyn data, we retrieved all pairs of changes that had at least 1 method in common to approximate for similar changes as we had in our exploratory study. Overall, we identified 49 pairs of change tasks that changed at most 2 classes and 3 methods and that had at least 1 method in common in their changes. When comparing the task contexts of each of these pairs, we found substantial differences on class and method level. Out of the 49 pairs, only 13 pairs have an overlap on class level within their task contexts and out of the 13 pairs, 12 pairs share 1 class element and 1 pair shares 2 class elements within their task contexts. On the method level, out of the 49 pairs, only 7 pairs share exactly one method within their task contexts. Since the task contexts of all change tasks of the 49 pairs have an average of 5.35 (SD = 4.57) classes and 5.49 (SD = 4.67) methods, the overlap in task contexts is relatively small. While it is impossible to recreate the study in which multiple developers perform the same change task and this analysis only approximates it, our results show that even if the changes of different concise changes overlap, the code navigation models are substantially different, providing some evidence that ***O3*** holds.

For the observation that code navigation models are highly connected (***O4***), we examined whether the elements within a task context are structurally connected. We considered elements as structurally connected if they either belong to the same class or if there is a call relationship between the elements on the method level. Disregarding 69 change tasks with only one element in their task context, we found that on average 68% (SD = 29%) of the task context elements are structurally connected, supporting ***O4***.

For the observation that code navigation models exhibit a high lexical cohesion (***O5***), we examined whether the elements within a task context are lexically related. To determine the lexical cohesion within task contexts, we split the elements according to the camelCase rule (e.g. SetupHelper is split into "Setup" and "Helper"). Resulting stop words, such as "get" or "set" were removed. We then consider an element as lexically connected, if it shares terms with at least 50% of the task context's elements. Disregarding again the change tasks which only contain one element, we found that on average 61% (SD = 35%) of the task context elements are lexically connected, supporting ***O5***.

Blaze Data.

Our exploratory study as well as a small study by Ko *et al.* [24] show that developers often begin a task by searching for an initial point and then move outwards using structured navigation (***O6***). To examine if this pattern also holds for professional developers in the field, we analyzed the Blaze data that captures interaction data from 59 professional developers, since the Mylyn data does not contain information on the searches performed. Since the Blaze data does not contain task boundary information, we can not examine whether developers start a task with searches, but we focus our analysis on whether there is an increase in structured navigation right after a search. Therefore, for each of the 59 developers within the Blaze data, we compared the general use of structured navigation over all of a developer's work in the IDE with the structured navigation in the minute immediately following a search. This again is only an approximation, but should provide evidence whether the observation also holds in the field.

We consider the following events as structured navigation in the Blaze data:

- Go To Definition (F12): brings up the code that defines the selected identifier.
- View Call Hierarchy (Ctrl+K Ctrl+T): provides a two way analysis of an identifier's dependencies and uses.
- Class View (Ctrl+W, C): provides a browser and search function for classes and class hierarchy.
- Find All References (Ctrl+K,R): provides a list of lines that reference an identifier.
- Navigate to Event Handler: brings up the event handler for an object in the XAML editor.
- View Class Diagram: generates a class diagram.
- View Object Browser: is a search tool and browser.

Unstructured navigation events include selecting a file in an explorer window, selecting the tab for a file, using arrow and page up/down keys to go up/down through a file, scrolling and clicking on a file element.

To examine whether there is an increase in structured navigation after a search, we define the metric structured navigation events per minute (SNM). For each developer, we counted the number of structured navigation steps overall sessions recorded and calculated the average SNM per developer ($SNM_{General}$). In addition, for each developer we identified all searches the developer performed, counted the number of structured navigation steps in the minute after the search and calculated the average SNM over all searches by the developer ($SNM_{AfterSearch}$). In this analysis, we consider two kinds of search that we treat separately: the execution of the command "Find in Files" and the use of the Sando code search [4]. Since all 59 developers used the command "Find in Files" (FiF), we collected 59 pairs of SNMs ($SNM_{General}$,$SNM_{AfterSearch}$) for FiF. Only 36 developers used the Sando search tool in their work. Therefore, we only collected 36 pairs of SNMs for Sando. A Wilcoxon Signed Ranks test showed that for both searches, there is a significant difference in the use of structured navigation immediately after a search compared to the general use of structured navigation with p=.006 (T=442, r=-.25) for FiF and p=.015 (T=166, r=-.29) for Sando, providing further evidence for ***O6***.

To examine if developers frequently revisit code (***O7***), and since the Blaze data does not contain information on when the work on a task started or ended, we performed a processing step to partition the Blaze data into sessions, each of one hour length. We chose one hour since this is a few minutes more than the average time participants used in our exploratory study to complete a change task. Per session, we then counted the number of class visits ($\#ClassesVisited$), i.e. the number of times a developer selected a class different to the one that was selected beforehand, and the number of distinct classes visited per session ($\#DistinctClasses$). The percentage of classes revisited can then be calculated as $\frac{(\#ClassesVisited - \#DistinctClasses)}{\#ClassesVisited}$. Over all participants and sessions, a developer visited an average of 16 classes per hour, with 6 distinct classes, resulting in an average percentage of revisiting of 62.5%, and supporting ***O7***.

4. THREATS TO VALIDITY

Exploratory Study. By applying a blocked subject-project study setup with developers from various backgrounds and three different change tasks of three active open source systems we tried to limit the threats to the external validity of our exploratory study and the experiment. To study change tasks representative of realistic situations, we used change tasks from active open source systems with a size big enough to preclude systematic understanding of the entire code base. A limitation of our study is that all tasks were solvable in less than two hours and thus might not represent the broad range of tasks that exist. We tried to mitigate this risk by choosing the change tasks as randomly as possible (see Section 2). Another threat is the limited size of our subject sample and the small number of change tasks which limits our study's generalizability. We tried to mitigate this risk by cross-sectioning full-time developers and researchers from different companies and universities with multiple years of programming experience.

In our exploratory study we focused on Eclipse and Java since they are amongst the most commonly used IDEs and programming languages. Navigation might differ depending on the tools provided in the IDE and language structure.

By screen capturing the participants we could only tell which elements they selected, but not which ones they looked at. This process misses elements and relations that were not explicitly followed through navigation steps, but our focus was on an obvious set of elements rather than an approximation of everything developers might have looked at. In future studies, we plan to explore the use of eye-trackers to also gather information on where developers look.

Empirical Analysis. To increase the external validity of our observations from the exploratory study, we conducted an empirical analysis on data sets from the field. Since the data sets used do not capture the same kind of data captured in our exploratory study, we used approximations, such as partitioning the Blaze data into one hour sessions rather than task sessions, or using a general metric on structured navigation rather than per task. These approximations pose a threat to the construct validity of our results. We tried to mitigate this risk by using two different data sets that captured different aspects of developer's interaction data to better approximate for the analysis of the observations. Furthermore, we never claim to fully validate our observations, but point out that the empirical analysis provides further evidence strengthening the support for the observations.

For the analysis on the Mylyn data, we were only able to analyze 2253 out of 9920 change tasks of the project, which implies that the observations are only denotative for 23% of the project's change tasks. Since developers manually start and stop the capturing of the task context for a change task, this empirical analysis is also threatened by possibly polluted task contexts. Also, we only take into account call relationships and class affiliation when analyzing structural relations between code elements for ***O4***, and do not include type hierarchy. However, this only results in our result being lower than it could be. Finally, the correlation analyses for ***O2*** and ***O8*** suffer the third-variable problem, which means that we cannot argue about the causality of the correlation.

5. DISCUSSION

The exploratory study (Section 2) and the empirical analysis (Section 3) revealed unique characteristics about the behavior of developers when working on a change task. Table 3 summarizes key observations along with inferred design requirements for tool support. While there are several approaches to explicitly or implicitly capture task context, such as Mylyn [22] or CodeBubbles [14] (see Section 6), these approaches are limited in the support of the presented design requirements. In the following, we discuss design considerations for tool support for change tasks that explicitly captures developer's code context model.

A Code Context Model Tool.

To support a developer in a change task, a tool should not only try to best depict the developer's current code context model—his representation of the relevant code elements and their relations—, it should also provide proactive and relevant context to the developer in all activities while working on the change task, from the search to the navigation and the editing of code. This will allow a developer to resume more easily from interruptions and speed up the navigation and search in the first place.

Table 3: Observations from Studies and Inferred Design Implications.

Empirical Observation	Design Requirement	#
Developer models are small, abstract and highly connected	Provide adequate abstraction with limited size, focus on highly connected parts and indicate relations	**R1**
Code navigation models are hardly influenced by the size of the actual change, but differ substantially by developer	Provide personalized navigation support	**R2**
Code navigation models exhibit a high structural & lexical cohesion; developers take less time if their navigation is more structured	Combine structural and lexical navigation support and provide proactive structural and lexically context	**R3**
Lexical cohesion of code navigation models is stronger at beginning of exploration	Adapt support and proactive context over time	**R3.1**
Developers use a combination of search and structural navigation	Combine search and navigation	**R4**
Developers frequently revisit code	Indicate and keep track of what was already visited	**R5**
Change task completion time can be several days	Provide a summarization/ abstraction and persistency for previous code context models	**R6**
Change task type influences size of code navigation model	Tailor support to change task type	**R7**

Combined View for Search and Navigation.
As found in our studies (**R4**) and by other researchers (e.g., [36, 24]), developers use a combination of search and navigation when performing a change task. Current tool support in IDEs, however, generally either support search or navigation, requiring developers to switch between views and loose track of dependencies. An initial approach that allows a single query and structural navigation to expand the search results was presented by Janzen et al. [21]. Since developers usually perform multiple queries for a change task over time and their code context model expands, a combined view should provide support for presenting multiple query and navigation instances at the same time, possibly in a time line view. In addition, to speed up the exploration, search results should be presented with relevant proactive structural context (structural recommendations) and the results should be ranked based on their cohesiveness with respect to the current developer's code context model (**R3**). Initial results on a limited form of structural context for search results already received positive feedback in a study by McMillan et al. [28].

Given the small size and high abstraction of developer's models (**R1**), such a combined view also has to provide an adequate abstraction and summarization. We plan to investigate approaches that aggregate and abstract information over time and yet provide enough and proactive context on the current selection.

Integration of Lexical Dependencies.
Along with the split between search and navigation in current IDEs also comes a split between support for following lexical and structural dependencies. While views such as the Package Explorer or the Call Hierarchy in Eclipse allow a developer to follow structural dependencies, the many lexical similarities that exist—and often present a semantic dependency—are neglected in these views. Given the strong lexical cohesion of code context models, tools should more explicitly support this kind of dependency (**R3**) without requiring to switch views and loosing the current working set. Recent research to recommend subsequent navigation steps already leverages the combination of structural and lexical information (e.g [20, 27]) to a certain extent. A view of the current code context model should provide explicit cues for these lexical dependencies, indicating the pivotal part of these dependencies, and thus easing the assessment of the relevance of elements and providing a rationale for certain elements in the model. Similarly, when presenting search results or the selected elements, the view should provide proactive lexical context and integrate it with structural context.

Adaptation to Developer and Task.
While generally more structural navigation might lead to a better performance, the specific elements and relations that are being explored for a given change task differ substantially by developer and depend on a lot of factors, such as the developer's experience, and preferences as well as the program comprehension strategy used, such as bottom-up or top-down (e.g. [15, 16, 30]). As also mentioned by Storey et al. [40], tools should support a wide variety of comprehension and navigation activities. So while a tool should provide support for the patterns that exist across developers such as frequent revisitations (**R5**) and the advantage of structural navigation (**R3**), it should also take into account the individual preferences of developers (**R2**). By learning from a developer's past and possibly an interactive component to adjust one's preferences, a tool might be able to provide better recommendations and also a more accurate representation of a developer's code context model of the past.

Since code navigation also varies with the kind of change task as well as over time, an approach should adapt the provided navigation or, more generally, context recommendations based on these factors (**R3.1,R7**). For instance, when a developer starts working on an enhancement and performs a search, the view could provide more structural and lexical context for each result than for a trivial change task. Later on in the task when the developer performs another search, the lexical context would be reduced adapting to the changing behavior as seen in ***O5***. Also, given that certain change tasks are performed over a series of days, for example, with communication happening in between developers to clarify parts of it, there needs to be an option to

persist code context models, similarly to the way that Mylyn stores task context, and summaries should be available to ease resuming the task or communicating about it with other developers (**R6**). We plan to conduct studies to further investigate the impact the task type has on the code context models and how to best summarize code context models to resume or share them.

6. RELATED WORK

Related work can be categorized into two areas: empirical studies on software developers performing change tasks and tool support for explicit task context.

Empirical Studies. Researchers have extensively observed and studied the program investigation behavior of software developers during maintenance tasks. Ko *et al.* [23, 24] conducted an exploratory study to determine patterns of navigation. They report on 10 developers working on simplistic tasks in a very small system, where they found patterns such as developers starting with a search and then navigating to related elements, collecting small fragments of task-relevant code. Robillard *et al.* [32] conducted an exploratory study to look at the differences in the program investigation behavior of successful and unsuccessful developers. From observing five developers performing a maintenance task on a reasonably-sized system, they found that successful developers reinvestigate methods less frequently and mostly performed structurally guided searches. LaToza *et al.* [25] observed 13 developers working on two tasks on a bigger system, to study how experience affects the program comprehension. They found that experienced developers visit less methods, thus wasting less time on understanding irrelevant methods. Sillito *et al.* [36, 35] conducted a laboratory and an industrial study with 25 developers in total working either on a given or on one of their own change tasks. They observed that developers first search for an initial focus point and then explore relationships from these points, also revisiting elements. Based on their observations they identified four types of questions developers ask during change tasks. Starke *et al.* [39] focused on the initial investigation phase of a change task and had ten developers perform tasks for 30 minutes. Their observations mainly focused on how participants form search queries and then skim through the results. Wang *et al.* [42] conducted an exploratory study with 38 students performing feature location tasks. They focused their study on the initial feature location process and identified search patterns, such as execution-based and exploration-based search. While our results support some of the observations made in the earlier studies, our two studies focus on the actual context models that developers built implicitly for a variety of different tasks and systems and how they overlap with the actual changes they perform for these tasks. In addition, different to most studies mentioned above, we conducted a combination of an exploratory study with real change tasks on three open source systems and an empirical analysis of data collected from open and closed source developers to validate our findings.

Other studies have observed developers to investigate the process and characteristics of program comprehension. Mayrhauser and Vans [41] used protocol analysis to explore the program comprehension of professional developers working on industrial maintenance tasks. Based on the results of their study, they formulated an integrated model, combining top-down and bottom-up strategies found by other researchers (e.g., [15, 16, 30]), to describe the cognitive processes of program comprehension. Corritore and Wiedenbeck [17] looked at the differences of the mental representation of expert procedural and object-oriented programmers carrying out maintenance tasks on very small systems. Their results show that expert programmers build a mixed mental representation of a program that includes detailed program knowledge as well as domain-based knowledge. Piorkowski et al. [31, 27] build upon the theory of information foraging, exploring how developers use *information scent* emitted from *cues* to guide program exploration, and especially study how quickly developers' goals evolve. These approaches focus on the cognitive process of program comprehension, while we investigate the actual code context models, the implicit knowledge, that developers build and retain during comprehension and changing the code.

Explicit Task Context Support. Several approaches provide support for explicitly keeping track of task context—the set of files or code elements a developer works with during a maintenance task. An early tool, Concern Graphs supports developers in recording task context in the form of concern graphs, but requires the developer to manually identify and add relevant elements [33]. Code Bubbles alters the usual IDE editor interface so that each code element a developer navigates to for a task is represented by its own bubble and relations that a developer followed between these bubbles are made explicit [14]. This way, the context for a task is automatically created when stepping through or editing code. Mylyn, a task-focused UI approach, differs to these approaches in that it automatically creates an explicit task context from a user's interaction with the development environment [22]. Similarly, DeLine *et al.* proposed to use a user's interaction history for a task to recommend where to navigate next in the code [18]. All of these approaches specialize in saving task context for elements *after* they have been discovered. We investigate the creation of task context and the design requirements for explicitly supporting developers in the creation and representation of their mental models.

7. CONCLUSION

Software developers currently spend much of their time on change tasks, partially due to the large cost of creating a code context model for each new task assigned to them. In this paper, we have presented the results of two studies, an exploratory study with 12 developers performing change tasks and an empirical analysis of data sets capturing work of professional developers on open and closed source projects. In these two studies we found, amongst other results, that the code navigation models of developers exhibit a high structural and lexical cohesion and that they differ by developer and task. We inferred design implications and presented design consideration for a tool to support the creation and explicit capturing of developer's code context models. In particular, we discuss the combination of search and navigation, the integration of lexical dependencies into the views that currently are predominantly focused on structural dependencies, and the adaptation of such tools to the developer's preferences and the current task. In future work, we plan to develop such an approach, paying particular attention to the presentation and summarization of code context models as well as providing proactive context to speed up the developer in performing change tasks.

References

[1] bugs.eclipse.org/bugs/.

[2] http://aws.amazon.com/ec2.

[3] http://eclipse.org/swt.

[4] http://sando.codeplex.com/.

[5] https://github.com/abb-iss/study-artifacts-for-code-context-models.

[6] http://sourceforge.net/projects/freemind/.

[7] http://sourceforge.net/projects/jpwsafe/.

[8] http://sourceforge.net/projects/rachota/.

[9] http://sourceforge.net/tracker/?func=detail&aid=2658881&group_id=144949&atid=760391.

[10] http://sourceforge.net/tracker/?func=detail&aid=3420227&group_id=7118&atid=107118.

[11] http://sourceforge.net/tracker/index.php?func=detail&aid=2933526&group_id=243370&atid=1122415.

[12] www.eclipse.org/mylyn/.

[13] V. R. Basili, R. W. Selby, and D. H. Hutchens. Experimentation in software engineering. *IEEE Trans. on Softw. Eng.*, 1986.

[14] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proc. of CHI'10*, 2010.

[15] R. Brooks. Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd international conference on Software engineering*, ICSE '78, pages 196–201, Piscataway, NJ, USA, 1978. IEEE Press.

[16] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543 – 554, 1983.

[17] C. L. Corritore and S. Wiedenbeck. Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *Int. Journal of Human-Computer Studies*, 50(1):61 – 83, 1999.

[18] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *Proc. of SoftVis'05*, pages 183–192. ACM, 2005.

[19] B. Fluri, M. Wursch, M. Pinzger, and H. Gall. Change distilling:tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, 33(11):725–743, 2007.

[20] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with dora to expedite software maintenance. In *Proc. of ASE'07*, 2007.

[21] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187. ACM, 2003.

[22] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proc. of FSE'06*, 2006.

[23] A. J. Ko, H. H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 126–135. IEEE, 2005.

[24] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during soft. maintenance tasks. *IEEE Trans. on Soft. Eng.*, 32:971–987, 2006.

[25] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *Proc. of ESEC/FSE'07*, 2007.

[26] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proc. of ICSE'06*, 2006.

[27] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2008.

[28] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Proc. of ICSE'11*.

[29] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranić. The emergent structure of development tasks. In *Proc. of ECOOP'05*, pages 33–48, 2005.

[30] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3):295 – 341, 1987.

[31] D. Piorkowski, S. Fleming, C. Scaffidi, C. Bogart, M. Burnett, B. John, R. Bellamy, and C. Swart. Reactive information foraging: an empirical investigation of theory-based recommender systems for programmers. In *Proc. of Human Factors in Computing Sys.*, 2012.

[32] M. Robillard, W. Coelho, and G. Murphy. How effective developers investigate source code: an exploratory study. *IEEE Trans. on Softw. Eng.*, 30(12):889 – 903, dec. 2004.

[33] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proc. of ICSE'02*, 2002.

[34] C. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Trans. on Soft. Eng.*, 25(4):557 –572, jul/aug 1999.

[35] J. Sillito, K. De Voider, B. Fisher, and G. Murphy. Managing software change tasks: An exploratory study. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 10–pp. IEEE, 2005.

[36] J. Sillito, G. C. Murphy, and K. D. Volder. Questions programmers ask during software evolution tasks. In *Proc. of FSE'06*, 2006.

[37] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proc. of Centre for Adv. Studies on Collaborative Research*, 1997.

[38] W. Snipes, A. Nair, and E. Murphy-Hill. Experiences Gamifying Developer Adoption of Practices and Tools. In *Software Engineering, 2014. ICSE 2014. IEEE 36st International Conference on*, 2014.

[39] J. Starke, C. Luce, and J. Sillito. Searching and skimming: An exploratory study. In *Proc. of ICSM'09*, 2009.

[40] M.-A. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.

[41] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *Proc. of ICSE'94.*, 1994.

[42] J. Wang, X. Peng, Z. Xing, and W. Zhao. An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. 2011.