Thomas Fritz
David Shepherd
Christoph Bräunlich

Supporting Search and Navigation through Code Context Models

2012

TECHNICAL REPORT – No. IFI-2012.07

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

Thomas Fritz, David Shepherd, Christoph Bräunlich
Supporting Search and Navigation through Code Context Models
Technical Report No. IFI-2012.07
Software Quality
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
null

Supporting Search and Navigation through Code Context Models

Thomas Fritz
Department of Informatics
University of Zurich
fritz@ifi.uzh.ch

David Shepherd
Industrial Software Systems
ABB Corporate Research
david.shepherd@us.abb.com

Christoph Bräunlich
Department of Informatics
University of Zurich
christoph.braeunlich@uzh.ch

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. Our study uncovered that code context models are highly connected and that developers start tasks using a combination of search and navigation. Building upon our findings and drawing from related studies, we developed an approach to automate the generation of code context models that combines the previously distinct phases of search and navigation. We evaluated our approach, CoMoGen, against the study data. CoMoGen performed significantly better than state-of-the-art and state-of-the-practice approaches for locating initial code elements necessary for generating code context models. We believe this work represents a substantial step towards providing automated code context models that will reduce the time and effort needed for change tasks.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

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 the system for a particular change task [19, 40]. 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 [22], developers have to continuously spend a significant amount of their time creating context models for newly assigned change tasks from scratch.

Researchers have suggested that more explicit context models for change tasks[1] can be used to support developers in their work [26]. To form these explicit context models, existing approaches have used different methods ranging from the developer manually specifying the context (e.g., [35]) to automatically inferring the context from a developer's interaction with a development environment (e.g., [13, 18]). While these approaches have been shown to support developers with change tasks, little is understood about the actual code context models that developers build and, in particular, which elements are relevant to developers. With a better understanding of the characteristics of developers' code context models, we can help developers in the creation of these models for change tasks, saving them time and effort.

To investigate the characteristics that code context models exhibit for different change tasks and different concepts of relevance, we conducted an exploratory study with twelve developers on three change tasks in three open source projects. For each change task session, we collected three context models: a *developer model* based on the developer's definition of relevance, a *patch model* representing the code changes and their relations, and a *code navigation model* inferred from transcribing all navigation steps of a developer. We found that developers' context models are highly connected, that the code navigation models can differ substantially for a task, and that developers start tasks using a combination of search and navigation and then frequently revisit code elements.

Based on our findings, we developed an approach that combines search with structured navigation to support developers in the creation of code context models. While existing search approaches focus on identifying a list of possibly relevant starting points [17, 23], our approach uses the search results as seeds, expands these by exploring their structural relations and automatically groups and ranks the resulting structural graphs. By combining the search results with structural navigation context and presenting them in diagrams, we may be able to get more relevant search results and eliminate navigation steps and revisits developers would otherwise perform.

To evaluate how well this approach can capture the context navigation models of developers we performed an evaluation, comparing our search and navigation approach against state-of-the-art and the state-of-the-practice search techniques for the three tasks from our exploratory study. The evaluation shows that combining search and navigation to generate

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

initial context models for change tasks outperforms existing techniques on both the class and method level. This suggests that the grouping of results and the visualization of code elements together with their call relations can support developers in building their code context models, eliminate navigation steps in the comprehension process of the code and also reduce the frequently occurring revisits in developer's navigation.

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 introduces the first approach that combines search with navigation to automatically generate code context models.
- It provides an evaluation of the approach showing that it is significantly more effective at generating the initial context relevant for change tasks than both the state-of-the-art and the state-of-the-practice.

This work represents a substantial step towards providing automated code context models 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?
(4) How do developers navigate code during change tasks?

To investigate these questions, we conducted an exploratory study with a blocked subject-project study setup [10] 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 [3], Java PasswordSafe (JPass) [4] and Rachota [5]. 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 and validated 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 task description and project. We decided to set up a virtual machine for each participant on the Amazon Elastic Compute Cloud[4] 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

For this study, we recruited subjects through email and personal contact. To be eligible, subjects had to have experience with programming in Java. To solicit participation, we invited more than 30 developers working in companies or at a university and we ended up with 17. Out of these 17 subjects we only had four who worked on the Rachota task. To ensure an equal number of subjects per task, we randomly chose four subjects for each of the other two tasks. We report on the selected twelve subjects, 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 years to 16 years (average of 11.8 years) with between 0 to 12 years (average of 4.5 years) of full-time/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.

[2]Murphy *et al.* ([26]) introduce the broader term *task context* that extends our notion to arbitrary artifacts.

[3]Eclipse IDE for Java Developers, version 3.7, `eclipse.org`.
[4]aws.amazon.com/ec2

On a five point Likert-scale with 1 representing strongly disagree to 5 representing strongly agree, all subjects agreed or strongly agreed that Java is one of their primary programming languages (average of 4.75), and were predominantly familiar with the Eclipse IDE (average of 4.17).

2.3 Projects and Change Tasks

We chose three active open source projects and one task for each of these projects.

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, [7]) 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, and 18 packages. We again selected a change task (ID 2933526, [8]) 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[5] 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, [6]) 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.

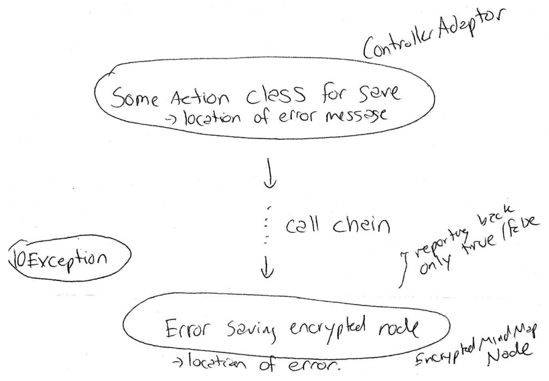
[5]`eclipse.org/swt`

2.4 Data Collection and Analysis

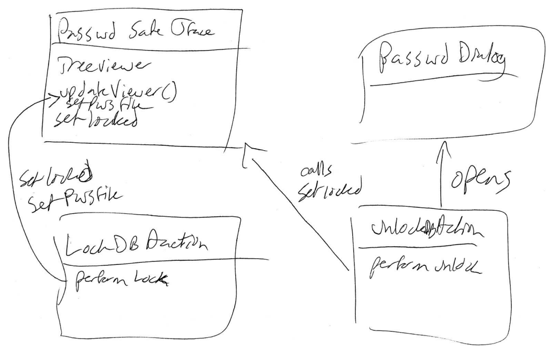
We used a combination of qualitative and quantitative methods motivated by the ones described by Seaman [36]. We used participant observation by recording each participant's screen and having access to their actual workspace after the session, in addition to asking the participant to answer a set of 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 artifacts collected in the study are available at [2].

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.

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. 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 that 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. For the code context models from the actual patch, which we refer to as *patch models* in the following, 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. For the models defined by the explicit navigation behavior of the developer, which we refer to as *code navigation models* in the following, we transcribed the screen recordings and coded the resulting transcripts. Since we were 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 that a developer took. We considered a navigation structured if a developer explicitly


ControllerAdaptor
Some Action class for save
→ location of error message
call chein
reporting back only true / false
IOException
Error saving encrypted node
→ location of error.
EncryptedMindMap Node

(a) F2's Developer Model



(b) J1's Developer Model

Figure 1: Developer Models for FreeMind and JPass.

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. A table of all transcribed navigation steps is presented in Table 1. 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.

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, and we used the keyboard events for determining some of the tool support a developer used. Observational studies are subject to observers' biases which may lead to omitting instances of the navigation 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 initial 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 four 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.

[6]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[6]
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

Developer models are small, abstract, highly connected and exhibit tree-like structures. Across all subjects and tasks, developer models are consistently small with an average[7] (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 (14.5 classes on average for FreeMind, 3.7 for JPass and 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 would record 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 subjects assigned to the FreeMind task 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 and abstracts from direct call relations to transitive call chains and Figure 1(b) shows the model of developer

[7]We use the mean as the average.

Table 2: This table presents a summary of the descriptive statistics collected on the developer's background and from the exploratory study ($pro = professional$, $grad = graduatestudent$, $fac = faculty$, ✓ = $success$, ■ = $failure$, $Cl = classes$, $Me = methods$, $Deb = debugging$).

Project	*ID*	*Job*	*Years Pr.Exp.*	*Time & Success*	*Dev. Model*	*Patch Model*		*Code Nav. Model*		*Navigation Steps*			
					Cl	*Cl*	*Me*	*Cl*	*Me*	*All*	*Structured*	*Revisits*	*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

J1 on the JPass task which resembles a class diagram with significant details about the code.

All developer models were also highly connected. In fact, all developer models were fully connected at the 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.

Finally, even though developers had the freedom to draw arbitrary graphs and the nodes were highly connected, they drew mental models that had tree-like structures with mainly directed edges and without cycles in the model. For the models from the ten successful subjects, three depicted a simple path, four a tree and another three depicted poly trees.

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 correlation coefficient between the patch and the code navigation model size being close to zero overall with r = 0.006. 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.*

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 instance, 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 wide variance outside of these four classes. The three subjects included between 6 and 19 classes in their navigation models with an average overlap of elements with at least one other subject's model of only 52.6%. On a 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 method of these 21 was inspected by all three subjects.

Code Navigation Models are highly connected. Upon inspecting all code navigation models for all successful 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. 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 elements that the developer navigated to in the beginning and two elements from seemingly random selections later on (30 and 31).

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.* [19] found that developers first search for information, then engage with the information to decide whether it is worth continuing to comprehend it by navigating the relationships between information, before finally editing the code. Similarly, Silito *et al.* [39] identified that developers exhibit a behavior of 'finding initial focus points' and then 'building on those points' in terms of navigation and exploration. Our exploratory study on the three

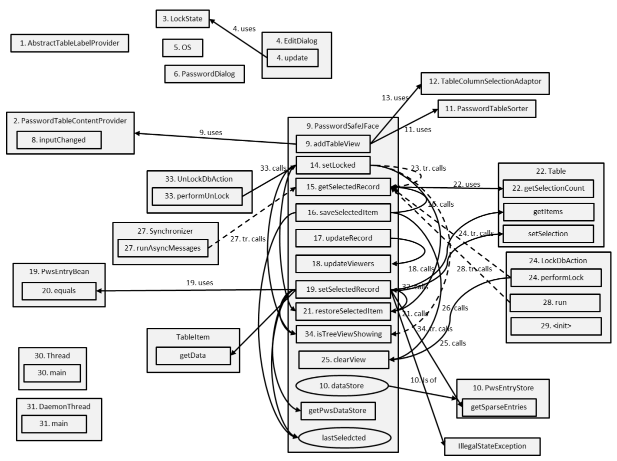


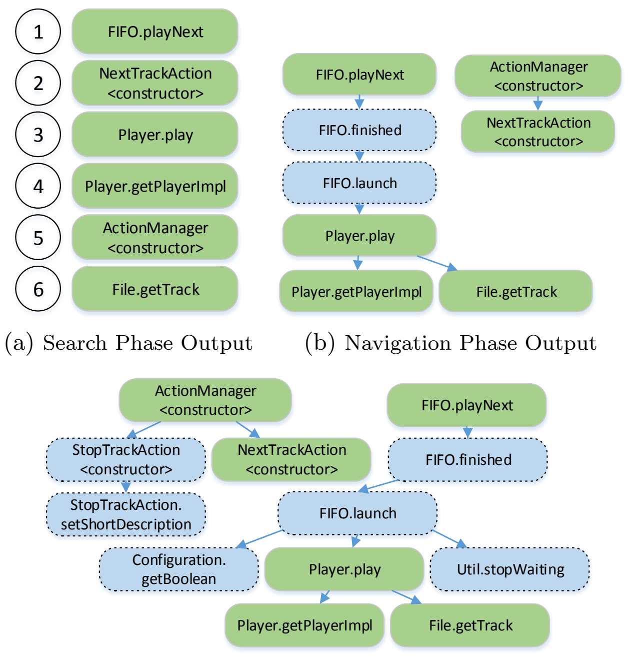
Figure 2: Code Navigation Model for Subject J2.

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.

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 $r = 0.72$). Furthermore, the higher the ratio of structured versus unstructured navigation was, the less time a developer spend on the change task (Pearson's $r = -0.49$). 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. GENERATING CODE CONTEXT MODELS

To support developers in their search and navigation for change tasks, we were interested in developing and evaluating an approach that generates initial code context models.



1
FIFO.playNext
2
NextTrackAction <constructor>
3
Player.play
4
Player.getPlayerImpl
5
ActionManager <constructor>
6
File.getTrack
(a) Search Phase Output
FIFO.playNext
FIFO.finished
FIFO.launch
Player.play
Player.getPlayerImpl
File.getTrack
ActionManager <constructor>
NextTrackAction <constructor>
(b) Navigation Phase Output
ActionManager <constructor>
StopTrackAction <constructor>
NextTrackAction <constructor>
StopTrackAction. setShortDescription
FIFO.playNext
FIFO.finished
FIFO.launch
Configuration. getBoolean
Player.play
Util.stopWaiting
Player.getPlayerImpl
File.getTrack
(c) Snapshot of Navigation Step

Figure 3: CoMoGen Output at Different Stages of the Search and Navigation Process (green methods are seeds, blue methods with dashed edges are non-seeds).

3.1 Approach

As observed in our exploratory study and supported by other research studies [19, 39, 43], developers tend to perform global text searches to locate a starting point and then navigate structural relations from the starting point for comprehension. If developers focused on structural relations, they tended to spend less time for completing a change task. Also, generally all developer models, whether sketched or inferred from developer's navigation, were highly connected.

Inspired by these observations, we developed an approach that combines the search and navigation and focuses on generating structurally connected code context models. Our approach, which we call CoMoGen, takes as input a user query and outputs a ranked list of connected call trees that can be visualized as sequence diagrams. The approach is composed of three phases: search, navigation, and visualization.

Search Phase. The search is split into three steps, the expansion, the execution of the search and the synthesis of the search results. The expansion step takes an initial query and creates a set of queries that contains all possible subsets of terms from the original query, transforming "play next" into the set {"play", "next", "play next"}. This idea is based on the observation by Ko *et al.* [19] that a lot of user searches fail to return anything of interest and approximates the backoff algorithm—a systematic removal of words from the initial query—that users employed during our exploratory study.

The next step takes the created queries and executes them using code search technology [38], outputing lists of relevant methods. A query for "play next" locates methods such as `FIFO.playNext`. This technique is used because researchers have observed that relevant code will often contain identifiers that hint at its purpose [23] and that users will leverage these information *scents* to guide their search [29]. Code search extracts these identifiers from each method, creating information retrieval documents. These documents are later compared against a query and the most similar methods are returned. While our query expansion and result synthesis are novel contributions, search execution is simply an invocation of a state-of-the-art code search technique.

The final step in the search phase collects all search result lists, one for each query, and combines them together into a single ranked list. The "play next" result list may contain `FIFO.playNext` while the"next" result list may begin with `FIFO.next` but also contain `FIFO.playNext`. To fairly combine these possibly overlapping result lists we use Team-Draft Interleaving [31]. This approach operates as a draft, choosing the highest remaining item from each result list during each round, resulting in a list similar to Figure 3(a).

Navigation Phase. The navigation phase is split into three steps, the navigation outwards from seeds to discover new relevant methods, the grouping of all found methods into call trees, and the ranking of these call trees. The navigation step takes search results as seeds and adds methods that are found via navigation. For instance, in our simple example, starting at the seed `FIFO.playNext`, shown as a green node in Figure 3(c), CoMoGen explores structurally related elements and finds that the blue non-seeds `FIFO.launch` and `FIFO.finished` build a path to another seed `Player.play`, thus discovering that `launch` and `finished` are relevant. The navigate step explores at most five call edges outwards from seeds and, if it discovers a path from one seed to another seed, it adds all methods along that path to the final result.

The grouping step takes the expanded list of methods and groups them into connected call trees. The previously unconnected search results in Figure 3(a) plus the methods discovered through navigation are transformed into Figure 3(b). Based on our observation that developers' models are highly connected and often tree-like the results are grouped into connected trees that now contain seeds along with structurally related methods. To group results into call trees, CoMoGen chooses the first method from the result list, creates a new tree and inserts it, and then adds related methods to this tree until no more methods can be added, repeating this process until all methods in the result list are grouped.

The ranking step transforms the set of groups into a ranked list. In Figure 3(b) the group starting with `FIFO.playNext` is ranked highest because the sum of its methods' relevance scores is highest. This ranking scheme favors highly connected groups, which again reflects the high connectivity of developers' context models. Note that the relevance score for a given method is its relevance with respect to its respective query from the search step.

Visualization Phase. The visualization phase uses ranked call trees as input and presents them as sequence diagrams by traversing the tree and creating a lifeline for each new class and a message for each method call. We chose sequence diagrams as an initial visualization of the ranked call trees based on our observations of developers using tree-like structures with a large number of directed call relations and on Dzidek et al's study showing developers being more effective when given access to UML diagrams [15]. For our example, the two trees in Figure 3(b) would be visualized as sequence diagrams.

3.2 Evaluation

In this section, we present an evaluation of CoMoGen's effectiveness in generating initial code context models. Specifically, we focused the evaluation on:

> **RQ:** How well does CoMoGen locate initial context—in terms of code elements—relevant for performing change tasks compared to the state-of-the-art and the state-of-the-practice?

Note that we explicitly did not choose to perform a user study for two reasons. First, user studies should almost always be proceeded with a thorough experimental analysis, as there is little value in executing a user study on an approach that cannot perform well in a controlled environment. Second, usability issues of beta quality software tools confound evaluations, and so we expect a beta quality tool to be hardened for at least a year before a realistic user study is valid (e.g., Mylyn's user study [18]). Nonetheless, we recognize the value that user studies will add in terms of confirming the benefits of this approach on change tasks, and therefore plan to conduct them as a next step.

3.2.1 Experimental Design

Variables and Measures. The *independent variable* in our study is the approach used to execute a developer's search and generate a set of program elements—the initial context. We chose Sando [1, 38] to represent the state-of-the-art and Visual Studio's built-in regular expression search to represent the state-of-the-practice.

When choosing a state-of-the-art tool to compare against we surveyed feature location tools that also leverage static information. Unfortunately, the few existing approaches were either not available, which is a well-known problem when evaluating against feature location tools [14], not search-based [16], or not effective without further refinement [37] (as also discussed in Shepherd *et al.* [38]). In our effort to compare against the most effective available solution we chose Sando, which implements a vector space model approach that performs as well as other leading IR approaches [27] and employs most known incremental improvements such as favoring certain types of identifiers and using stopwords to filter indexed text.

Since we are interested in how well an approach can locate code elements that are relevant to a change task, we choose the F-measure as the *dependent variable.* The F-measure is the harmonic mean of precision and recall. Precision represents the fraction of relevant elements that are found by an approach divided by the total number of elements found. Recall represents the fraction of relevant elements found divided by all possible relevant elements. The F-measure balances the competing needs of precision and recall, ensuring that the best possible combination of precision and recall achieves the highest score. Furthermore, F-measure is one of the most used measures for feature location and code search evaluations [14, 17, 25] and facilitates cross-study comparisons.

Subjects. To evaluate the techniques for a set of change tasks, we need to know which code elements are relevant to perform each change task. For this, we leverage the data collected in our exploratory study. Note that this data is ideal for evaluating the techniques as it stems from realistic tasks on open source projects and contains the actual elements that a developer navigated to complete a change task, information on which elements developers deem relevant and the actual code changes. Since we are interested in supporting developers in their search and navigation, we primarily focus our evaluation on the code navigation models of developers, *i.e.*, the elements a developer navigated to in order to perform the change task. The developer models are only developers' abstraction of what is relevant and they are neither complete nor consistent across developers. The substantial variation among the developer models makes them problematic to evaluate in a uniform fashion. Finally, the patch models only capture the changed code rather than the elements that a developer needs to understand and navigate to perform the change.

We chose to evaluate the results generated by the techniques against each developer's code navigation model to investigate whether we can provide relevant initial context to each developer. We did not evaluate the results against a golden set of elements determined by a set of experts as studies have shown that even experts have low agreement (about 34%) when tagging a golden set for a task [33]. The developer models from our study show a very similar low agreement rate per task with an average of 36.2% overlap on the class level. Finally, we did not pick a "best" code navigation model, since we made sure that all of our participants have sufficient programming experience and our results show that different developers solve tasks differently even if they perform equally well.

Additionally, the data collected in our exploratory study contains the actual search strings that developers used when completing their tasks. Since not every subject performed a global search, we chose the most prevalent search string from each task, specifically "Saving failed" from the Freemind task (used by all four subjects) and "Trying to unlock" from the JPassword task (used by two subjects that performed a global search). For the Rachota task there was no agreement across study participants who performed a global search, so we chose the string "history tab" which appeared in one user's query, in the original bug report, and was selected as the best search string by our pilot study subjects.

Method. Given a task and the identified search string, we first executed a search using each of the three approaches. We then compared the results against each code navigation model that we collected from the successful developers for the given task. For each approach a different number of results was returned. To fairly compare the results we bounded the number of results we considered for each approach, as is typically done during search evaluations [17]. If necessary, we decided to only consider the top 10 results from Sando and the top 5 sequence diagrams from CoMoGen. Since the state-of-the-practice tool returns unranked results we had to consider all results. Note that this detail had no practical impact on our evaluation (as no result list was ever shortened), but is noted for reproducibility and further experimentation.

3.2.2 Results

We calculated the F-measures for the approaches under study and present them in Figure 4. The top two plots represent the summaries across tasks, at the class and method

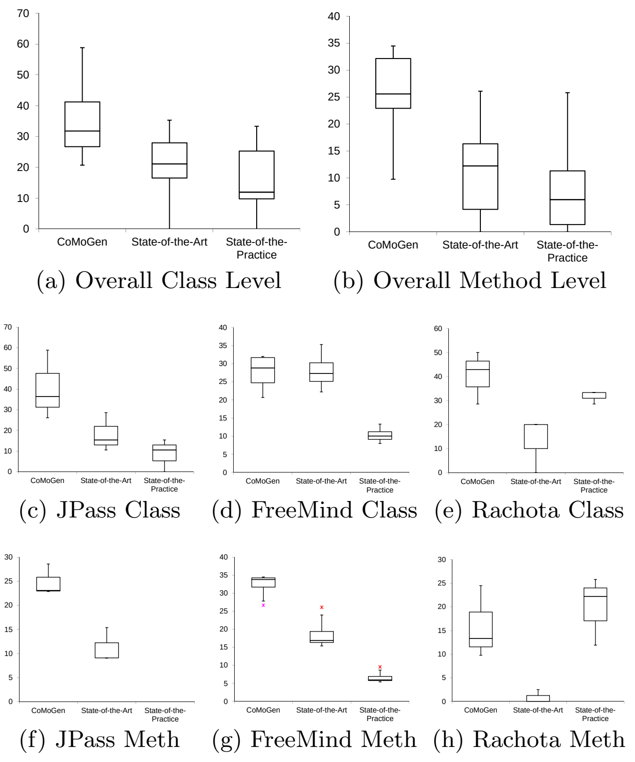


Figure 4: F-Measures of CoMoGen vs State-of-the-Art and State-of-the-Practice Tools on Class and Method (Meth) Level.

level.

We compared the results for CoMoGen with the state-of-the-practice using the Wilcoxon signed-rank test, a non-parametric paired samples test. At the class-level, CoMoGen is statistically significantly better than the state-of-the-practice with respect to the F-measures (two-tailed $p = 0.007$, mean F-measure for CoMoGen of $F_{CMG} = 35.3$ and a mean $F_{SOP} = 16.3$ for the state-of-the-practice). At the method-level, CoMoGen also performed better with statistical significance (two-tailed $p = 0.014$, mean $F_{CMG} = 25.1$ and mean $F_{SOP} = 8.7$). Thus, at both the class and the method-level *CoMoGen outperformed the state-of-the-practice tool.*

We also compared the results for CoMoGen with the state-of-the-art. At the class-level, CoMoGen is statistically significantly better than the state-of-the-art with respect to the F-measures (two-tailed = 0.03, mean $F_{CMG} = 35.3$ and mean $F_{SOA} = 20.7$ for the state-of-the-art). At the method-level, CoMoGen also performed better with statistical significance (two-tailed $p = 0.005$, mean $F_{CMG} = 25.1$ and mean $F_{SOA} = 11.1$). Thus, at both the class and the method-level *CoMoGen outperformed the state-of-the-art tool.*

Developer and Patch Models. As previously mentioned, we primarily calculate F-measures for the code navigation models due to the limitations of the other two models. However, for completeness we also present F-measures for the developer and patch models here. Due to the varying levels of abstraction in developer models, we focus on the class level. The average F-measure on the class level is higher for CoMoGen than both the SOP and the SOA approaches ($F_{CMG} = 60.2$, $F_{SOP} = 38.8$, $F_{SOA} = 55.0$), however, there is no statistical significance. This lack of significance might also stem from the small size of developer models (mean of 4.6 elements). With respect to the patch models, CoMoGen is statistically significantly higher on the class level than SOP (two-tailed $p = 0.014$, $F_{CMG} = 44.6$, $F_{SOP} = 17.5$) and on average higher than SOA ($F_{SOA} = 25.1$), but this difference is not statistically significant (two-tailed $p = 0.078$). On the method level, CoMoGen is statistically significantly higher than both the SOP (two-tailed $p = 0.03$, $F_{CMG} = 28.0$, $F_{SOP} = 13.9$) and the SOA (two-tailed $p = 0.03$, $F_{SOA} = 12.7$) tools.

4. DISCUSSION

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 approach evaluation. To study change tasks that are 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 of these tasks were completable 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 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. Since our first study was exploratory, we do not claim that the results are generalizable to a broader population. Further studies are needed to investigate generalizability of our observations.

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 environment and the structures in the language.

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.

During the experiment we calculated F-Measures using developers' navigation models as an approximation for a "gold set" rather than having experts choose a gold set. Studies have found that even with experts there is a high disagreement on what constitutes a gold set [33]. By using actual developer navigations, we might include some noise, however, we consider this a good and realistic standard for evaluation.

An open question is whether an automatically generated model affords the same robust understanding as a model generated by a developer. Much like studying a city map versus walking around in the city, the actual navigation through the code and even the revisitations might partially be needed for a better understanding. We plan to investigate this question as well as the extent to which this approach can save time and effort in a future user study.

5. RELATED WORK

Related work can be categorized into four areas: empirical studies on software developers performing change tasks, approaches for explicit task context, approaches for search and visualization, and program navigation.

Empirical Studies. Researchers have extensively observed and studied the program investigation behavior of software developers during maintenance tasks. Ko *et al.* [19] 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. 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.* [21] 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. While our results support some of the observations made in the earlier studies, we did not want to study differences, but focus on the actual context models that developers built implicitly for a variety of different tasks and systems.

Other studies have observed developers to investigate the process and characteristics of program comprehension. Mayrhauser and Vans [42] 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, to describe the cognitive processes of program comprehension. Corritore and Wiedenbeck [12] 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. [29] 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. None of these approaches directly investigate the code context models that developers built during comprehension.

Explicit Task Context. During maintenance tasks developers often work with a set of several files–a task context—which can become difficult to keep track of. An early tool, Concern Graphs, supported developers in recording task context in the form of concern graphs, but required the developer to manually identify and add the relevant elements [35]. A very recent approach, 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 [11]. 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 [18]. Similarly, DeLine *et al.* proposed to use a user's interaction history for a task to recommend where to navigate next in the code [13]. All of these approaches specialize in saving task context for elements *after* they have been discovered. We investigate the creation of task context to speed the initial discovery phase.

Search, Visualization, and Feature Location. There is a variety of software search tools, also known as textual feature location tools [14] such as Google Eclipse Search[30], SNIAFL [44], Sourcerer [9], that can provide a good starting point for a developer's navigation. In an exploratory study, Starke *et al.* have shown that general search tools provide too many results, many of them irrelevant [41]. They suggest that more contextual information would be valuable to help developers judge the relevance of the results. Portfolio, a search approach presented by McMillan et al. [24] already provided some very basic call graph visualization along with the result list. Participants in their study mentioned that they liked more context for the result visualization. While CoMoGen can be used for searches as well, the focus of our research is to explicitly understand the code context models that developers built for change tasks and use these insights in our approach to support developers in generating context models for the task, not just with understanding search results.

Code Navigation. As programmers navigate outwards from starting points they often utilize navigation tools. Researchers have extended navigation tools to recommend relevant next steps. Robillard et. al [34] uses program structure topology, Piorkowski *et al.* [29] combines several factors including structure and lexical information, as does Hill et al. [16], and Parnin et al. [28] even leverages recency information. These approaches only recommend a single step outwards, and thus would need to be adapted for use during our navigation phase. Nonetheless, we plan to investigate some of these alternative navigation techniques in future work. Reacher by LaToza et al. [20] supports developers in exploring control flow graphs. While not the primary focus of this work, Reacher allows developers to filter call graph chains via string queries. CoMoGen not only uses search strings to filter navigation, but also uses control flow information to rank search results and provide more context for the search results.

6. 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 introduced an approach that can automatically generate code context models starting from a user query. The generated context models are a combination of code elements on the class and method level that are structurally connected via call relations. These context models are motivated by the observations we made in an exploratory study in which twelve developers completed three different change tasks on open source systems. Through an evaluation of our approach, we found that our combination of search and navigation is significantly more effective than the current state-of-the-practice and state-of-the-art approaches for locating initial code elements needed in the generation of context models.

This work opens up new research opportunities in several ways. In this paper we have chosen to focus on the concept of combining search with navigation for the purposes of generating code context models. While we believe this is a substantial step, there is also additional room for improvement through innovation and research on the search and navigation algorithms themselves. Additionally, the idea of combining search with navigation has not only the potential for identifying context when starting a change task but also for supporting developers throughout a developer's work. For instance, future research could investigate the generation of relevant navigation context for any code search to support developers in their understanding, or could provide better rankings of search results by examining the overlap between a developer's navigation history and the navigation context of search results.

7. REFERENCES

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

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

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

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

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

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

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

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

[9] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An internet-scale software repository. In *Proc. of SUITE'09 at ICSE'09*, pages 1–4, 2009.

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

[11] 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*, 2010.

[12] C. L. CORRITORE and S. WIEDENBECK. Mental representations of expert procedural and object-oriented programmers in a software maintenance task.

[13] 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.

[14] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Soft. Maint. and Evol.: Research and Prac.*

[15] W. J. Dzidek, E. Arisholm, and L. C. Briand. A realistic empirical evaluation of the costs and benefits of uml in software maintenance. *IEEE Trans. Softw. Eng.*, May 2008.

[16] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with dora to expedite software maintenance. In *Auto. Soft. Eng.*, 2007.

[17] E. Hill, L. L. Pollock, and K. Vijay-Shanker. Improving source code search with natural language

phrasal representations of method signatures. In *ASE*, pages 524–527, 2011.

[18] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Int. Symp. on Foundations of Soft. Eng.*, 2006.

[19] 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. *Trans. on Soft. Eng.*, 32:971–987, 2006.

[20] T. LaToza and B. Myers. Visualizing call graphs. In *Visual Languages and Human-Centric Computing, 2011*, 2011.

[21] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *Proc. of European Soft. Eng. Conf. and Foundations of Soft. Eng.*, 2007.

[22] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Int. Conf. on Soft. Eng.*, 2006.

[23] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proc. of Working Conf. on Reverse Eng.*, 2004.

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

[25] C. McMillan, D. Poshyvanyk, and M. Revelle. Combining textual and structural analysis of software artifacts for traceability link recovery. In *Proc. of Wkshp on Traceability in Emerging Forms of Soft. Eng.*, 2009.

[26] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranić. The emergent structure of development tasks. In *Proc. of the European Conf. on Object-Oriented Prog.*, pages 33–48, 2005.

[27] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *Proc. of Int. Conf. on Program Comprehension*, 2010.

[28] C. Parnin and C. Gorg. Building usage contexts during program comprehension. In *Proc. of Conf. on Prog. Comprehension*, 2006.

[29] 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.

[30] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu. Source code exploration with google. *Proc of ICSM'06*, 0:334–338, 2006.

[31] F. Radlinski and N. Craswell. Comparing the sensitivity of information retrieval metrics. In *Research and Dev. in Info. Retrieval*, 2010.

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

[33] M. Robillard, D. Shepherd, E. Hill, K. Vijay-Shanker, and L. Pollock. An empirical study of the concept assignment problem. Technical report, 2007.

[34] M. P. Robillard. Automatic generation of suggestions for program investigation. In *Proc. of Foundations of Soft. Eng.*, 2005.

[35] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proc. of Int. Conf. on Soft. Eng.*, 2002.

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

[37] P. Shao and R. K. Smith. Feature location by IR modules and call graph. In *SE Regional Conf.*, 2009.

[38] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz. Sando: An extensible local code search framework. In *Proc. of Foundations of Soft. Eng., Tool Demo*, 2012.

[39] J. Sillito, G. C. Murphy, and K. D. Volder. Questions programmers ask during software evolution tasks. In *SIGSOFT'06/FSE-14*, 2006.

[40] 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.

[41] J. Starke, C. Luce, and J. Sillito. Searching and skimming: An exploratory study. In *Int. Conf. on Soft. Maint.*, 2009.

[42] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *Proc. of 16th Int. Conf. on Soft. Eng.*, 1994.

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

[44] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. Sniafl: Towards a static noninteractive approach to feature location. *Trans. Softw. Eng. Methodology*, 2006.