CodeMap: A Graphical Note-Taking Tool Cooperating with an Integrated Development Environment

Rikito Taniguchi
rtaniguchi@prg.is.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

Hidehiko Masuhara
masuhara@acm.org
Tokyo Institute of Technology
Tokyo, Japan

ABSTRACT

Program comprehension is an important yet difficult activity in a software development process. One of the main causes of the difficulty is its cognitive overhead for maintaining the *mental models*, which consist of roles of program elements and relationships between them. Though researchers have been working on tools to help maintaining the mental models, existing tools have high adoption barrier and support only a few programming languages, which hinders wide-range of programmers from using the program comprehension tools. We propose CodeMap, a graphical note-taking tool for offloading the mental models onto a visual representation. We designed CodeMap by considering familiarity and availability for practitioners. Our tool allows programmers to extract interested information into a graphical note with a few keyboard/mouse operations, and support many programming languages by using the language server protocol. In this paper, we present the design and implementation of CodeMap, and discuss possible features that could be useful for program comprehension.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

program comprehension, code navigation, software engineering

ACM Reference Format:
Rikito Taniguchi and Hidehiko Masuhara. 2022. CodeMap: A Graphical Note-Taking Tool Cooperating with an Integrated Development Environment. In *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming (‹Programming› '22 Companion), March 21–25, 2022, Porto, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3532512.3535225

1 INTRODUCTION

1.1 Program Comprehension in Software Development

Program comprehension is an important yet difficult task in software development. It is an activity to construct a *mental model* of a software system, which consists of the roles of programming elements such as modules and functions and the relationships among those elements [6, 12]. It is mandatory to do this when we add a new feature, fix a bug, and refactor a software system.

Program comprehension is a difficult and time-consuming task, especially for a large software system. Xia et al. identified that professional programmers spend 58% of their development time for program comprehension [15].

One of the sources of the difficulty is the cognitive overload to maintain a mental model when exploring a large software system [14]. Since the construction of a mental model is an exploratory process, the programmers have to visit many programming elements regardless of the relevance of the elements to their specific task. As a result, when a programmer visits one element after a long course of exploration, he/she can forget the initial motive of the exploration and tends to visit the same elements again and again.

1.2 Graphical Note-Taking Tools for Drawing Mental Models

Taking a graphical note, which illustrates program elements and their relationships, is a common approach to keep track of the course of exploration during a program comprehension activity. It can be done by using paper and a pen, or by using a software tool, which can be a stand-alone tool, or an integrated with a software development environment. Though those tools use different graphical representations, they commonly incorporate selected program elements during exploration, and graphically organize those elements based on the relations between the elements.

Mylin [7] is an IDE integrated tool that creates and maintains a task specific view of program elements. Its graphical representation reuses the package explorer view by only showing relevant elements in a tree structure. When the programmer explores program elements in the code editor, it automatically incorporates the visited elements into the view, while the programmer can give a name to the view, and switch the saved views in the past.

JASPER [4], Code Bubbles [3] and Synectic [1] are IDEs that have a large space (i.e., *canvas*) that contains small editor windows, each of which shows a small program element (e.g., methods). In those systems, the systems and the programmer can freely layout those elements so that related elements come closer to each other. We call such a system *a canvas-based IDE*. The programmers explore by using common IDE functions, such as "go to definition," which will create a new element on a canvas. Not only program elements, those systems allow the programmer to place non-code artifacts such as documentation, issue reports, sample code, and design rationale on the canvas.

‹Programming› '22 Companion, March 21–25, 2022, Porto, Portugal

ACM ISBN 978-1-4503-9656-1/22/03…$15.00
https://doi.org/10.1145/3532512.3535225

CodePad [11] is a stand-alone note-taking tool for program comprehension. It is motivated by the observation that the programmers use paper and other media to maintain their mental model [10]. Though it is a stand-alone tool (it is even designed to be used on a different console alongside with a console for coding), it is designed to work with an IDE by retrieving the currently browsing program elements from the IDE, and by navigating the IDE to program elements that was recorded in a CodePad's memo.

1.3 Familiarity and Availability

We argue that the existing note-taking tools are limited in familiarity and availability, which hinders from being used by wide-range of programmers.

In terms of familiarity, some of the aforementioned tools require the programmers to use different user-interface from that they are usually using for daily development. In fact, many of code navigation features, such as "go to definition", provided by those tools have different user-interface from those in common IDEs. Since modern IDEs are rapidly evolving to embrace many useful features, it is almost impossible for those tools to catch up. Therefore, this can lead to lower acceptance from practitioners as pointed out by Roehm et al. [13].

In terms of availability, some of the tools support one or a few programming languages. For example, Code Bubbles and Synectic only support Java. This makes those tools difficult to be used in the age where many industrial projects use more than one programming language in recent years [9].

1.4 Our Approach: IDE Plugin with Language Servers

To address the familiarity and availability issues, we advocate to build a note-taking tool as an IDE plugin connected to language servers, and designed and implemented a prototypical tool called *CodeMap*. Our contributions in this paper are summarized as follows.

- We presented a user-interface design of CodeMap that enables the programmer to explore source code by using existing IDE functions, while extracting interested information into a graphical note with a few keyboard/mouse operations. This will offer better familiarity compared to the tools with their own editor functions.
- We identified a set of IDE-plugin API that is required to implement CodeMap. Although our implementation currently assumes only Visual Studio Code (VSCode), we believe that it can also be easily implemented for other IDEs that offer the required APIs.
- We resolved dependency on programming languages by using the *language server protocol* (LSP)[1]. Even though CodeMap requires semantic level information of the programs being explored, the LSP-based implementation reduces the cost of adapting the tool to new languages and will improve availability of the tool. CodeMap is also an interesting use of LSP for constructing a non-trivial IDE feature.

[1] https://microsoft.github.io/language-server-protocol/

2 CODEMAP

We set the following two design goals for CodeMap.

- It uses a graphical representation that can intuitively match the programmers' mental model of a system, and can relate back the graphical elements onto the program elements in the code editor.
- Its implementation exploits existing code editors and language systems as much as possible so that it increases familiarity and availability.

In this section, we first summarize the characteristics of the programmer's mental model during the code comprehension and how we designed CodeMap based on it. Then, we introduce the user-interface design of CodeMap, and the implementation to achieve familiarity and availability. Finally, we illustrate programmers' code comprehension activity with a CodeMap using a scenario.

2.1 Graphical Note-Taking for Program Comprehension

2.1.1 Program Comprehension Model. Ko et al. proposed a program comprehension model consists of the three phases: Search, Relate and Collect [8]. In this model, programmers search for a starting point of program comprehension, and move back and forth between those three phases until they collect all the necessary information to complete the program comprehension task.

In the Search phase, programmers look for a source program element that is relevant to the task as a starting point of the program comprehension. For example, if the programmer attempts to understand a specific feature's behavior, the starting point would be an API endpoint or a specific method. To find such an element, they use a lexical search on various information through the development environment, such as identifier names, comments, and documentation, to identify their relevance to the task.

After finding a starting point for the program comprehension, in the Relate phase, they attempt to understand the program element by exploring related program elements from the starting point, such as dependent functions and modules. To do so, they use a structural code navigation function (like "go to definition" and "find references" provided by IDE) or a lexical search function (like grep).

During the program exploration, programmers collect the necessary information for the program comprehension and construct a mental model in the Collect phase. While the mental models for program comprehension can be different between programmers, Ko et al. argue that the mental model consists of the roles of program elements such as function and modules, how they work, and their relationships, such as how they interact with each other. In this paper, we design CodeMap based on Ko's model.

2.1.2 Design and Implementation. Based on Ko's program comprehension model, programmers construct a mental model during program exploration. However, maintaining a mental model in their head would be a huge cognitive overhead, and he/she can forget the initial motive of the exploration.

CodePad supports programmers maintaining the mental model by allowing them to take graphical notes integrated with source code. However, programmers have to arrange the graphical notes

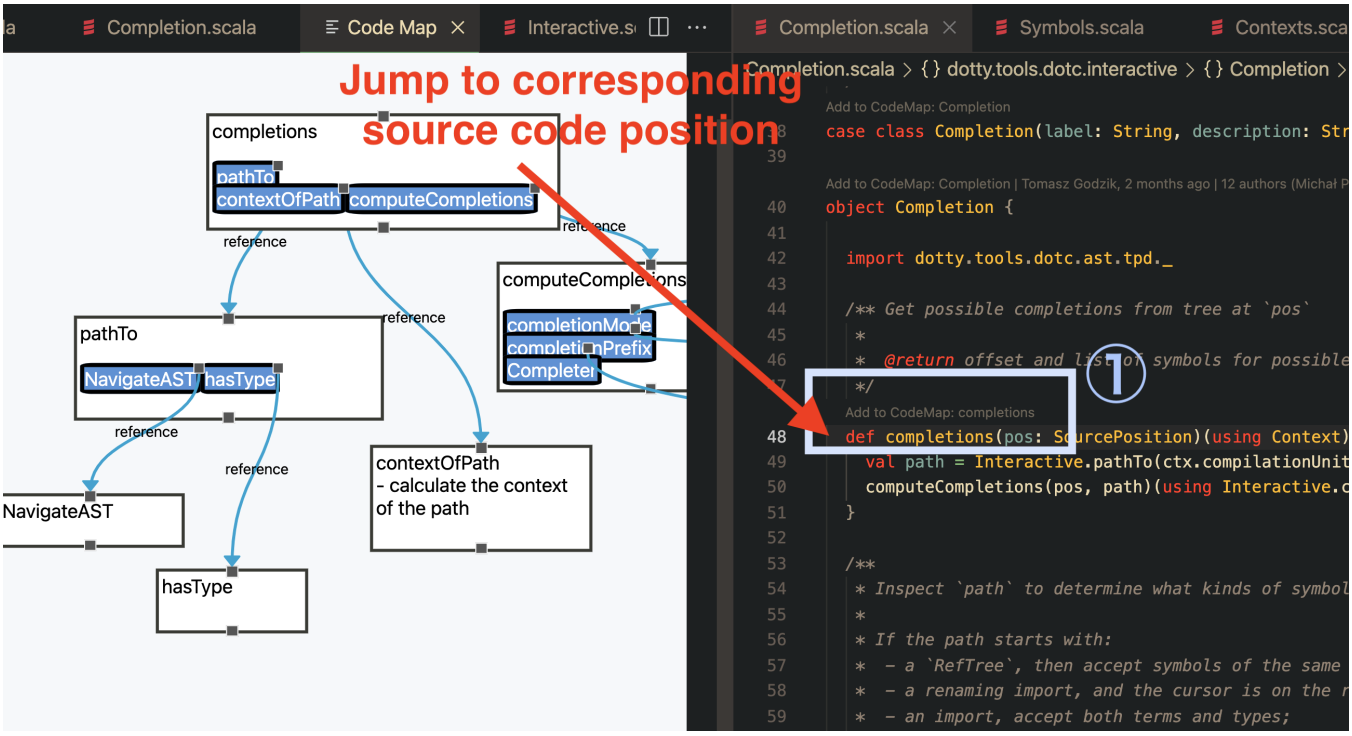

Figure 1: Screenshot of CodeMap

by hand, and they have to switch back and forth between IDE and CodePad's display. This can distract programmers from code reading and exploration.

Using CodeMap, programmers can concentrate on code reading and exploration as we can arrange nodes in CodeMap with a few keyboard/mouse operations. Figure 1 shows the overview of CodeMap. The left-hand side depicts the canvas where programmers can freely arrange the graphical notes. We can add a node to the canvas that corresponds to the program element by clicking the CodeLens [2] that shows above the program elements as shown in Figure 1-1, and we can click the nodes to add a custom comment. On the other hand, when we want to recall the details of a node, we can jump to the corresponding source code position by double-clicking the node. Thus, we can move back and forth between the editor and canvas without a cognitive overhead. On top of that, CodeMap captures the IDE's navigation events and automatically constructs relationships between nodes on the canvas. Hence, we do not need to take care of arranging the relationships on the canvas during the program exploration.

2.2 Design for Familiarity and Availability

To achieve familiarity, we developed CodeMap as an extension to an existing IDE VSCode, and enabled programmers to use VSCode editor for code reading, editing, and exploration as they do for daily development.

We use LSP functions for various features such as showing CodeLens and capturing the code navigation events. For example, we retrieve the symbol information using *textDocument/documentSymbols* to show the CodeLens on program elements such as class and function, and we use the combination of the symbol information and *textDocument/definition* request, *textDocument/references* request, and so on for capturing the code navigation events in VSCode.

When programmers use code navigation, CodeMap automatically adds nodes that correspond to the symbols we jump from and to. LSP does not have an interface to specify which symbols programmers navigate from and to, they only know the positional information such as file path, line, and column of where it comes from and jump to. To specify them, we search for the minimum bounding symbols that contain the navigation positions, from all the symbols in the files given by *textDocument/documentSymbols* request.

Thus, even though CodeMap features require semantic level information of programming languages, CodeMap is available in any programming language as CodeMap implements those features based on LSP.

2.3 Program Comprehension Scenario

To illustrate how CodeMap supports programmers' comprehension tasks, we use a scenario where we attempt to understand how Dotty [3] compiler's code completion API works. In this scenario, we

[2] https://code.visualstudio.com/blogs/2017/02/12/code-lens-roundup

[3] https://dotty.epfl.ch/

Figure 2: Programmers can add a node corresponding to the completion method by clicking the CodeLens

.

explore the `lampepfl/dotty` [4] repository [5] using Metals [6] language server implementation for Scala programming language.

```
def completions(pos: SourcePosition)(using Context) = {
   val path = Interactive.pathTo(
     ctx.compilationUnit.tpdTree,
     pos.span
    )
   computeCompletions(pos, path)(
     using Interactive.contextOfPath(path))
}
```

Listing 1: completions method in Completion.scala

Following the Ko's program comprehension model (Section 2.1), we begin with searching for a starting point. In this case, we found an endpoint of the code completion API by lexical search (Listing 1). After finding the starting point, to take a note about how `completions` method works, we added a node corresponding to the method on the canvas by clicking the CodeLens (Figure 2).

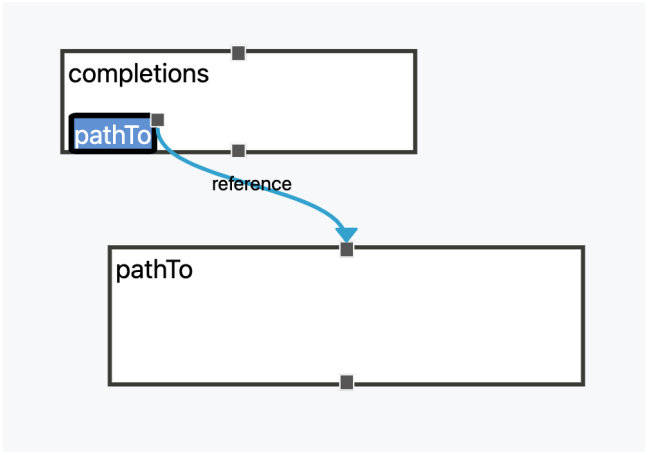

Figure 3: CodeMap automatically adds a node for the pathTo method and connects it with the node for the completions method.

Following the program comprehension model, we explore linked program elements by accessing definition of `pathTo` method to understand how this method works and what it returns. When we access the `pathTo` method using "go to definition" feature, CodeMap automatically adds a node for `pathTo` method and connects the nodes with an edge (Figure 3). CodeMap automatically adjust a position of the new node using a graph layout algorithm. Therefore, we do not need to consider the layout of the nodes during code exploration.

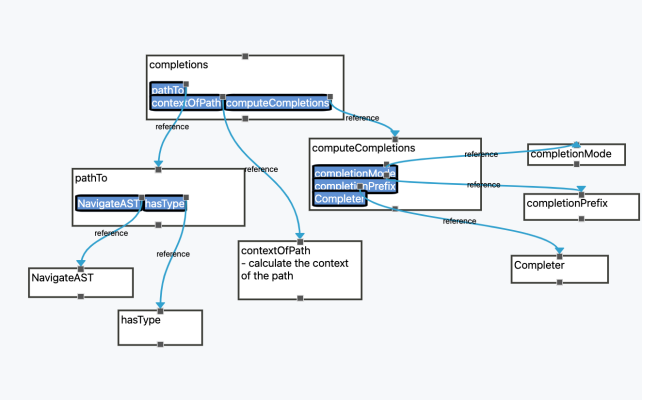

Figure 4: CodeMap constructs a graphical network.

According to the program comprehension model, we construct a mental model during the code exploration in our head, which leads to a cognitive overhead. Using CodeMap, the relationships of the program elements related to the `completions` method are visualized in the canvas (Figure 4), and this visual representation supports maintaining the mental model. In addition, we can instantly access the relevant program elements for the program comprehension tasks by double clicking the node in the canvas (Figure 1). This feature reduces the cognitive overhead for reconstructing the forgotten elements of the mental model in our head.

3 DISCUSSION

In this section, we discuss challenges that we learned from our initial prototype.

3.1 Canvas Space Scarcity

As Bradley et al. pointed out in their work on Code Bubbles, desktop space scarcity limits the number of nodes displayed [2]. As CodeMap shows its canvas in a Visual Studio Code's tab, the canvas space, especially its width, is too small to display many nodes. This limitation may prevent programmers from simultaneously viewing all the information necessary for a task, which is one of the main causes of difficulty in program comprehension [5].

One possible solution for this limitation would be to place canvas in a separate window instead of a VSCode tab. For example, we can provide a wider canvas by implementing the canvas user-interface as a separate web application that runs on a web browser and communicates with an IDE. However, this solution requires programmers to have an additional setup for CodeMap.

[4] https://github.com/lampepfl/dotty
[5] Revision: b472e2662ed1da9c896246adf69b07f61d7fe2f6
[6] https://scalameta.org/metals/

3.2 Geometric Information of the Graph Layout

Since CodeMap automatically places nodes by using a graph layout algorithm, their positions reflect only the distance with respect to program relations (e.g., caller/callee), but not the proximity in the source code. It could be meaningful if CodeMap calculates the position of the new node based on the proximity in the source code.

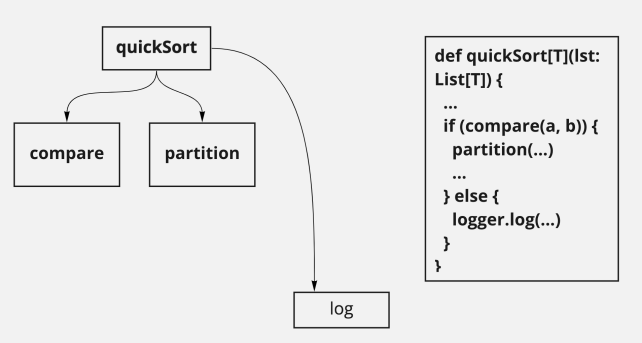

Figure 5: Placing node closer if two nodes have close proximity

For example, Figure 5 shows a layout after explored callees from `quickSort` function, where `quickSort`, `compare` and `partition` are placed close to each other, and `log` is placed a bit far away. We believe this layout is more informative as it reflects the proximity in terms of modularity. Even though our current implementation does not use information other than the edges on the graph, there should be a number of techniques that estimate semantic proximity and structure of program elements that can be used for better layouts in the future.

3.3 Degree of Interest Model

Programmers can freely change the size and color of the nodes. This will allow us to emphasize the importance of nodes, but it depends on programmers' manual involvement. It could be beneficial for programmers if CodeMap automatically resizes or sets color based on the degree of interest score for the node [7], which is calculated based on the frequency of interaction with the element and an interactions' recency.

For example, as shown in Figure 6, CodeMap would highlight the `partition` method if programmers frequently access the method, which means they are interested in `partition` method. Meanwhile, CodeMap would automatically lighten and shrink the `log` node if CodeMap calculates that programmers' low interest in this method. With this feature, programmers can recall the context about the important parts of the system and where are unimportant details at a glance.

4 CONCLUSION

We proposed a prototype tool called CodeMap, a VSCode extension for supporting program comprehension in a large software system. The design goals of CodeMap are twofold: use a graphical representation that can intuitively match the programmers' mental model of a system and design considering familiarity and availability for practitioners. To achieve the first design goal, CodeMap

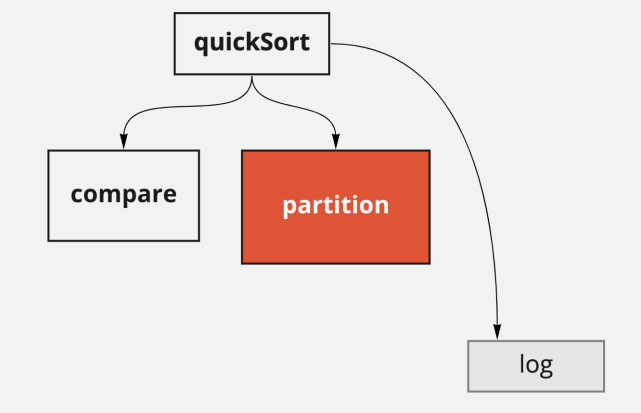

Figure 6: partition node will be highlighted as a programmer frequently accesses it

allows programmers to take graphical notes integrated with source code with a few keyboard/mouse operations. For the second goal, CodeMap exploits existing code editors and language systems as much as possible. Thus, programmers can use VSCode editor for code reading, editing, and navigation which they usually use for daily development. Furthermore, we resolved dependency on programming languages by using the LSP. Therefore, CodeMap is available in many programming languages, which shall encourage programmers to adopt our tool in an environment that consists of multiple programming languages. We believe CodeMap to bridges program comprehension research and industrial needs.

REFERENCES

[1] Marjan Adeli, Nicholas Nelson, Souti Chattopadhyay, Hayden Coffey, Austin Henley, and Anita Sarma. 2020. Supporting code comprehension via annotations: Right information at the right time and place. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–10.

[2] Daniel R Bradley and Ian J Hayes. 2013. Visuocode: A software development environment that supports spatial navigation and composition. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 1–4.

[3] Andrew Bragdon, Robert Zeleznik, Steven P Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. 2010. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2503–2512.

[4] Michael J Coblenz, Amy J Ko, and Brad A Myers. 2006. JASPER: an Eclipse plug-in to facilitate software maintenance tasks. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*. 65–69.

[5] Brian De Alwis and Gail C Murphy. 2006. Using visual momentum to explain disorientation in the Eclipse IDE. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. IEEE, 51–54.

[6] Vikki Fix, Susan Wiedenbeck, and Jean Scholtz. 1993. Mental representations of programs by novices and experts. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*. 74–79.

[7] Mik Kersten and Gail C Murphy. 2006. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 1–11.

[8] Amy J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering* 32, 12 (2006), 971–987.

[9] Philip Mayer and Alexander Bauer. 2015. An empirical analysis of the utilization of multiple programming languages in open source projects. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. 1–10.

[10] Chris Parnin and Robert DeLine. 2010. Evaluating cues for resuming interrupted programming tasks. In *Proceedings of the SIGCHI conference on human factors in*

computing systems. 93–102.

[11] Chris Parnin, Carsten Görg, and Spencer Rugaber. 2010. CodePad: interactive spaces for maintaining concentration in programming environments. In *Proceedings of the 5th international symposium on Software visualization*. 15–24.

[12] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology* 19, 3 (1987), 295–341.

[13] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How do professional developers comprehend software?. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 255–265.

[14] Jonathan Sillito, Kris De Voider, Brian Fisher, and Gail Murphy. 2005. Managing software change tasks: An exploratory study. In *2005 International Symposium on Empirical Software Engineering, 2005*. IEEE, 10–pp.

[15] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (2017), 951–976.