An Exploratory Study on How Can Diagramming Tools Help Support Programming Activities

Seonah Lee, Gail C. Murphy, Thomas Fritz, Meghan Allen
University of British Columbia
{salee, murphy, fritz, meghana}@cs.ubc.ca

Abstract

Programmers often draw diagrams on whiteboards or on paper. To enable programmers to use such diagrams in the context of their programming environment, many tools have been built. Despite the existence and availability of such tools, many programmers continue to work predominantly with textual descriptions of source code. In this paper, we report on an exploratory study we conducted to investigate what kind of diagrammatic tool support is desired by programmers, if any. The study involved 19 professional programmers working at three different companies. The study participants desired a wide range of information content in diagrams and wanted the content to be sensitive to particular contexts of use. Meeting these needs may require flexible, adaptive and responsive diagrammatic tool support.

1. Introduction

Programmers use diagrams to help with many activities associated with producing a software system [4]. For instance, to help understand a portion of a code base a programmer may sketch diagrams of how the code is structured and executes [4, 18]. Many tools have been built to aid programmers with their use of diagrams, including tools to depict software graphically (e.g., [23]), to help with sketching aspects of a system (e.g., [11]), and to help explore code (e.g., [10, 21]).

Despite all of these previous efforts, the majority of programming activity occurs in text-centric development environments with information often conveyed through list and tree views. If programmers worked efficiently and effectively in these environments, there may be little reason to consider how to better support programming through diagramming tools. However, there are reports that the status quo is not optimal [1]. Given that programs have been described as "several, general directed graphs, superimposed one upon another" [3] and given that programmers use sketched diagrams during program development activities [15], there is reason to believe that better tool support for diagrams might aid programmers in their daily work.

Why then do programmers not use existing diagramming tools? One theory is that since programmers can conceptualize software in their minds, diagramming tools are not necessary [8]. Others theorize that existing tools have not satisfactorily addressed cognitive issues with presenting diagrams, such as perceptual cues and layout [19, 24]. Yet others believe that creating and maintaining diagrams is a bottleneck in the process of understanding source code [6] and thus these tools do not provide a suitable cost-benefit trade-off [9, 20].

To investigate what support programmers want in a diagramming tool to aid during programming activities, we performed an exploratory interview-based study that involved 19 practicing programmers from three different companies. We interviewed the programmers about their current use of diagrams to gather a baseline of information. We then provided each programmer with a simple tool that diagrammatically depicted the relationships between pieces of code on which they were working. The intent of this tool was to get the programmers thinking about possible diagramming support: what information should the diagrams contain, when might the diagrams be helpful, and what interactive features are needed for the tools to be useful. The participants in the study had many ideas about the kinds of diagramming tool support that are needed. Interestingly, there is more breadth in the participant responses than reinforcement of particular trends. One of the few common themes of agreement was that the diagrams need to auto-configure to show *interesting* information, with many different, vague notions of what information might be interesting in a given situation. As we describe, the challenge in meeting the needs of programmers for diagramming support may require highly flexible, adaptive, interactive tool support.

In addition to providing empirical data about the scope of diagramming tool support of interest to programmers, results from our initial interviews of the participants provides data that affirms Cherubini and colleagues' findings about how developers at Microsoft use diagrams [4]. By reporting on similar findings across more corporations, our data helps generalize the earlier reported results.

We begin with a comparison to previous efforts investigating programmers and diagrams (Section 2). We then describe the study setup (Section 3) before presenting our analysis of interview data gathered from the participants (Section 4). We then discuss our findings (Section 5) before summarizing the paper (Section 6).

2. Related Work

A number of efforts have sought to identify how programmers use diagrams and to elicit their requirements for tool support for diagrams.

One approach has been to undertake field studies of programmers. For example, studies led by Cherubini [4, 5] and LaToza [17] examined how practicing developers utilized diagrams and diagramming tools in their daily work. Our study is similar as we also gathered information through interviews about how practicing developers use diagrams. Our study differs because we focused on programming activities and on considering issues that prevent programmers from using diagramming tools.

Other researchers have performed lab experiments to understand if and how diagrams can help programmers understand a software system. Cox and colleagues provided a variable dependency diagram and gathered developers' comments on the diagram [7]. Tilley and colleagues [25] and Hadar and colleagues [12] investigated which UML diagram formats were useful for program comprehension. In contrast to our study, these efforts were paper-based; the study participants did not interact with tools. Our study placed a prototype tool in the context of the development environment being used by the programmer to elicit comments from a realistic setting.

Studies have also been conducted to investigate whether particular diagramming tools help developers understand software programs: Zayour and colleagues experimented with call tree diagrams [26]; Hendrix and colleagues experimented with control structure diagrams [13]; and Storey and colleagues examined comprehension strategies programmers used with three different tools [24]. The primary focus of these studies was on how developers use particular diagramming tools rather than how they think about diagramming tools and what they need in those tools.

Another approach that has been taken is to survey programmers. Bassil and colleagues surveyed 107 participants with 38 questions aimed at rating the characteristics of existing tools [2]. Koschke surveyed 82 researchers with 19 questions that investigated the challenges of and visions for future diagramming tools [16]. Our approach differed in gathering in-depth qualitative feedback in an industrial setting.

Kienle and colleagues [14] and Storey and colleagues [22] identified potential requirements and issues of diagramming tools through literature surveys. Though those studies compiled comprehensive requirements for diagramming tools, they made inferences about the needs of programmers from studies focused on particular issues; with our study format. Our focus was on gathering similar information directly from practicing programmers.

Tools with a diagramming component have typically been evaluated in some way. For instance, Relo, a tool that supported exploring source code from a graphical element, was evaluated by nine programmers in a lab [21]. These type of studies provided some feedback on what programmers need from a diagramming tool, but the format of the study typically focused on the characteristics of the particular tool rather than on a programmer's needs across a range of activities. Our study aimed to capture comments across more than one usage context.

3. Study Format

Our study was interview-based. We conducted an initial interview with each participant to gather information about his or her experience using diagrams to aid programming activities. We then installed, and asked each participant to use, a simple tool intended to elicit more detailed responses about programming-oriented diagramming tools. We modified the tool as we gathered comments from participants. After each participant had time to use the tool, we conducted a follow-up interview.

3.1. Participants

We recruited nineteen industrial programmers from three companies to be participants in the study. To be eligible, a participant needed to program in Java using the Eclipse JDT[1] as a primary activity in their work.

[1] The Eclipse JDT (Java Development Tools) provides a state-of-the-practice integrated development environment for Java. It is available from www.eclipse.org, verified 01/03/08.

These constraints were necessary for a participant to be able to use our tool.

Each of the company sites we visited had over fifteen programmers. At the first site, twelve programmers participated: three from one team, four from another and five from a third team. At the second site, the two participants were from different teams. At the third site, all five participants were from one team.

Figure 1 presents the years of industry experience for each participant. Each participant is identified by a two-character combination: the first representing the company and the second representing the individual. The participants' experience ranged from 0.2 to 20 years, with an average working experience of 7 years.

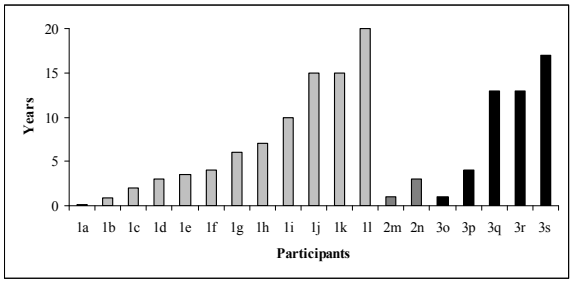


Figure 1. Participant experience.

3.2. Method

We visited each participant twice. During the first visit, we interviewed participants about their general experiences using diagrams and diagramming tools, installed a diagramming tool intended to help a programmer keep oriented during a code modification task and explained how to use the tool. This visit took less than 30 minutes per participant. During the second visit, we interviewed each participant for up to one hour. One investigator (the third author on this paper) interviewed 12 participants at the first site. Two investigators (the first and fourth authors on this paper) worked as a team to interview seven participants across the second and third sites. These visits occurred over three months.

3.2.1. Interview on diagramming experience. The questions asked at the first interview focused on a participant's work environment and prior use of diagrams. Table 1 provides a sample of the questions.

From these interviews, we learned that eleven participants occasionally sketched diagrams on whiteboards (9 of 19: 1c,1g,1h,1i,1j,1l,3p,3q,3s) or on paper (6 of 19: 1g,1h,1i,3o,3p,3r) to communicate with other team members or to understand an abstraction of the source code. These diagrams included personal variations on class diagrams and sequence diagrams.

Most participants had experience using diagramming tools that supported the drawing of diagrams. Only three participants (1d,1l,3o) had never used a diagramming tool.

Table 1. First interview questions.

Q1. Do you use diagrams in your work?
Q1.1 If so, what kind of diagrams do you use?
Q1.2 How do you create these diagrams?
Q1.3 What is the purpose of these diagrams?
Q1.4 How often do you use diagrams?
Q2. Have you ever used a diagramming tool?
Q2.1 If so, do you have a diagramming tool installed on your computer?

3.2.2. Using the prototype. To help elicit more detailed feedback about a diagramming tool for programming activities, we asked each participant to use a simple prototype tool that we had developed. As a previous study had shown that programmers can easily get lost when switching tabs in editors [1], this tool is displayed a class-diagram-like view about code currently opened in editors. The tool can present the diagram-like view in either a separate view within the development environment or on a separate screen.

The bottom right view in Figure 2 shows our tool after a programmer has opened editors on two files. As the programmer opens new editors, the diagram changes to include the classes declared in those editors.

The tool has three modes. In the first mode, methods and fields are added to the diagram when selected in the editor by the programmer. The second mode automatically presents all of the fields and methods in the classes for which there are structural relationships. The third mode automatically presents all of the fields and methods of the displayed classes.

Not surprisingly, given its simple nature, each participant had comments about how to improve the tool. We chose to modify the tool over the course of the study to accommodate some of these suggestions in an attempt to elicit a wide range of comments. In particular, between sessions at the second and third sites, the tool was changed to use multiple threads to improve responsiveness. Before sessions conducted at the third site, we changed the tool's layout from a spring layout to a tree layout and improved the layout to keep displayed classes at the same positions in the view. We also made the selection of a method in the view reveal the method in the editor.

We asked participants to use the tool in their daily work for up to a week and to record their experiences with the tool daily in a diary. We also equipped the tool with a monitoring facility to log participants' interaction with source elements and the prototype. We used these two data sources to help place participants'

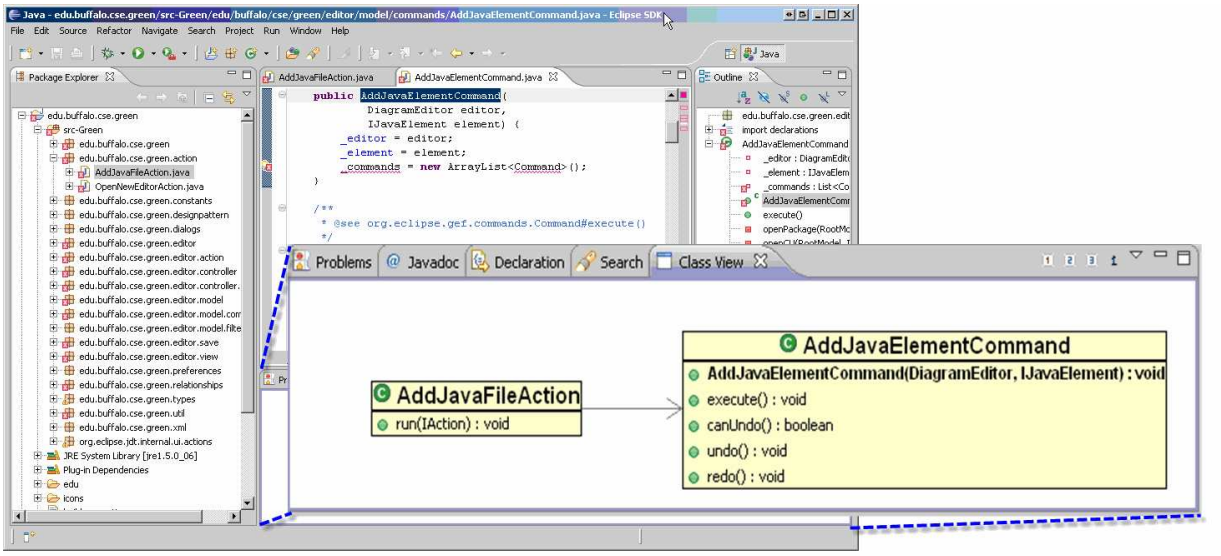


Figure 2. Prototype tool.

qualitative feedback in context. As there is no discrepancy between participants' responses and the data, we focus on summarizing participants' comments.

3.2.3. Post-tool-use Interview. Table 2 outlines the general questions we asked in the second post-tool-use interview. Due to space constraints, we show the specific hierarchy of questions asked only for the first question. Our approach to each interview was to ask the general questions, asking the specific questions only when a participant did not express detailed opinions. We did not stick rigorously to this interview plan, instead engaging the participant in conversation as it seemed appropriate.

Table 2. Post-tool-use interview questions.

Q1. What do you think about the tool?
Q1.1. How would you evaluate the tool?
Q1.1.1. Which mode do you like the most? Why?
Q1.1.2. Which mode do you like the least? Why?
Q1.2. Did you find the tool useful? Why or why not?
Q1.2.1. How often did you use the tool?
Q1.2.2. For what tasks did you use the tool?
Q1.2.3. How was the tool most useful to you?
Q1.2.4. Would you continue to use this tool or a similar tool? Why or why not?
Q2. In what situations would you use the tool?
Q3. How was the amount of information shown in the class diagram? too much? too little? …
Q4. How was the information presented in the class diagram?
Q5. What was the biggest usability problem with the tool?
Q6. Did you ever re-organize the diagrams? If so, why?
Q7. Can you think of any particular situations where using the diagrams reduced the amount of time or effort you spent understanding some source code?
Q8. Did you have any errors in the prototype?
Q9. Do you have any more comments or questions?

4. Study Results

What do programmers want to see in a diagram? When do they want to use a diagramming tool? How do they want to interact with a diagram? Does the diagram need any particular visual characteristics? We use these overall questions to describe the interview data gathered from the participants.

4.1. What should be in a diagram?

The participants in our study described many kinds of information that they desired to see in a diagramming tool. We have attempted many different categorizations of the information that they described; the breadth of the information described means that no categorization we have tried has been completely satisfying. The categorization we have found to provide the best structure for interpreting the information is the level of detail desired in the information presented, such as whether the main nodes in the diagram are packages, classes, methods or objects. We use this categorization in Table 3 to organize the information about what participants thought should appear in a diagram.

Many of the rows within Table 3 request information graphically that is already available through existing textual facilities in a development environment, such as which methods call a particular method or which methods reference a particular field. Other rows describe information not easily available, such as how objects interact with each other; one

participant described, *"I draw how objects connect to each other at runtime when I want to understand code that is unknown; an object diagram is more interesting than a class diagram, as it expresses more how it functions"* (1j). [2] As another example, another participant described desiring a diagram about which classes are listener classes: *"If you have [many] system listeners, where people register methods or classes to callback […an] interesting visualization would be […] to explore the actual instances of classes at run-time; it would be better than the list of listeners"* (3q).

Many participants (10 of 19) suggested a diagram should show *interesting* information. Several uses of this term referred to what parts of a system should be included (or excluded) in a diagram, as in "*it would be nice to see the context of the element I am currently looking at: classes around and how they relate"* (1e), and *"I picked out something and looked at its context; always only sections of it"* (1j). The term was also used to describe that the diagram content should be determined automatically based on an analysis of characteristics of the system; for instance, one participant stated, *"what should actually be refactored is most interesting to me"* (1h). Four participants described that the information presented should change according to a programmer's interest, shifting over time (1c,1e,1f,1h): *"First, I wanted to see what was going on in the code, something new […] It would be interesting for the next morning to remember what you have done the day before."* (1f). These participants did note the challenge of determining and focusing on interesting information: *"most existing tools generate those huge diagrams with loads of uninteresting information […] it is difficult to find out what is interesting"* (1c).

4.2. When is a diagram useful?

We asked participants to describe general situations in which a diagramming tool might be useful. These uses can be roughly categorized into exploration, navigation and externalization.

Many participants expressed interest in a diagramming tool to help when exploring a part of the code base, particularly in situations where they are new to the code or they receive code from someone else to review (17 of 19: 1a,1b,1c,1d,1f,1g,1h,1i,1j,1k, 1l,2m, 3o,3p,3q,3r,3s). A participant described *"I would definitely use a [diagramming] tool at certain times; particularly, if I was exploring the area where I was not familiar with a class hierarchy; I will probably open it up and just quickly look at how things are going"* (3s). Another participant, speaking to the situation of reviewing code from someone else, said *"looking at a patch would be very interesting: comparing to how things are usually done and trying to find things that are missing"* (1l).

Some participants also stated that a diagramming tool could provide a quick overview to help them quickly navigate back to code visited previously (8 of 19: 1a,1c,1d,1i,1j,1l,3q,3r): *"it would be useful to develop a cognitive map of the software, and it would help to navigate relationships"* (3q). One participant

Table 3. Desired diagram information.

Level	Desired Information	IDs
Method	Who calls a method / call hierarchy	1b,1c,1d 1e,1f,1i, zq
	Who uses/references who	1b,1d,1f, 1h,1j,
	Which class overwrites which method	1b,1c,1j,
	Who implements a method	1e, 1j
	What is used most	1h
	Who creates who	1c
	The paths navigated through methods	1d,zq,3r
	Which classes are related to a method	1d
	The methods implementing those APIs	1j, 3q
	Trigger paths when callbacks happen	1f, 1j
	Deadlocks	1j
	A stack trace	1d
Variable	The changes of variables of a class (i.e. when the variables are initialized and used)	1k
Object	Objects that interact with each other	1f, 1j, 3q
	Scenarios / Sequence diagrams	1i, 1l
Class	Which are the listener classes	1j
	Who owns who	1k
	Who interacts with who	1l
	Type hierarchy	1b,1c,1d, 1f,3q,3s
	The relationships among those classes created by himself	1h
	How two classes are related	3s
	Which classes are related to a class	3p
	Who is exactly listening for a system of listeners	3q
	Which classes are parts of a public API	3s
Package	Which package depends on which other packages	1h,3p
	Which packages have circular references	1h
	What external references packages refer to	1f

[2] Some of the participants were interviewed in German. As a result, some quotes are translations. We have used quotation marks for all quotes to make clear where comments are from participants.

also believed that a diagram oriented at providing an overview of code navigation might help prevent a bug, *"The diagram could give a better idea of what a developer was working on to prevent him from introducing a bug (e.g. this inherits from X so I need to be careful when overriding method Y)"* (3r).

There were a number of different uses mentioned by participants for externalized diagrams that could be saved and restored to facilitate communication with others and used as documentation for future use (8 of 19: 1a,1b,1c,1d,1k,1l,2m,3p). As an example, one participant described, *"It would be cool if I could step through something, create a diagram and save it so that I can look at it again in a year"* (1c). It was not clear how the participant would use the diagram a year later. They also imagined using it as a support for design: *"Diagrams would be useful for designing and discussion."* (1d).

4.3. How should a diagram be used?

Our participants noted several actions that they would like to perform in a diagramming tool to manage and work with information presented.

*Adaptation***.** Most participants wanted interactive support for adapting a displayed diagram to their needs, such as being able to add and remove elements from the diagram through various means, such as dragging and dropping (1a,1b,1c,1d,1f,1h,1i,1j,1k,1l, 3p,3q,3s) and filtering by name patterns (1g,1h,1k) or by the frequency of use by other software elements (1h). Others expressed wanting to add to diagrams in a bulk fashion, such as dropping in an entire stack trace (1b), a package (3s) or patch file (1l).

Grouping, Zooming, Marking and Annotating. Similar to features provided in many existing diagramming tools, participants mentioned a desire to group elements to reduce the presented information (1b,1k) along the type hierarchy or by name patterns. Participants also wanted the ability to zoom in and out in levels of detail of information presented in the diagram (1i,1k,2m,3p) and to mark and annotate the diagram (1b,1c,1f,1i,1l).

Property Checking. Participants desired support that checks for the violation of particular properties (1a,1h,3q), such as circular dependencies of packages (1h) and dead code (3q).

Switching Diagram Content. Five participants requested the ability to have a single diagram per task and to switch the diagram according to the task (1b, 1i,2n,3q,3s).

4.4. Presentation Challenges

"All diagramming tools take a lot of screen real-estate and it is hard to generate a diagram that contains exactly what a developer wants." (2m)

Many participants commented on the limited screen space available to show a diagram (10 of 19: 1b,1e,1i, 1j,1k,1l,2m,3p,3q,3s) and commented on the challenges of managing the amount of information in a diagram (12 of 19: 1b,1c,1d,1e,1f,1g,1h,1i,1j,1k,1l,3p).

To keep a programmer oriented, nine participants wanted a diagramming tool to support an automatic layout close to human perception (9 of 19: 1b,1c,1f,1g,1i,1j,1l, 2m,3q). It was also desired that a displayed diagram have a stable layout because participants described remembering where nodes were positioned and losing their orientation if the layout changed (8 of 19: 1b,1e,1f,1g, 1h,1j,1l,2m).

5. Discussion

The results of our study present several challenges for tool developers. The results also affirm uses of diagrams described in an earlier study conducted by researchers at Microsoft. We discuss each of these issues and describe threats to the validity of the results of our study.

5.1 A Challenge for Tool Developers

From a tool developer's perspective, it would have been great if all programmers wanted a similar kind of diagram to be supported by a diagramming tool. A narrow set of requirements could likely be built in a single tool and experimented with to see if productivity or quality increased as a result of using the new tool.

Our study did not produce such a narrow set of results. Although the wide range of results that did come from the participants' comments indicates many opportunities for tool developers, it also introduces substantial challenges. How can the many needs of the participants be met when there is a limited amount of screen real estate to devote to a diagram?[3]

Introducing a separate tool for each kind of diagram and situation may be unworkable due to the large number of tools that would be needed; finding the right tool in such an environment would likely be impossible for a programmer, particularly since some uses appear to be for a short amount of time.

[3] Twelve of the participants in our study use multiple screens with a total available screen space of 34~38 square inches. The majority of programmers are unlikely to have substantially more screen real estate available in the next five years.

Supporting more than one need in a tool might lead to complicated tools that may be too difficult for a programmer to use cost-effectively. Tool developers will likely need to find a balance between these two extremes. Tools may need to adapt to their context of use, reconfiguring with the appropriate kind of information, possibly based on analyzing a developer's interaction with the development environment.

5.2 Uses of Diagrams

In the interviews we conducted, we asked participants to describe programming situations in which diagramming tools might be useful. Table 4 compares the uses reported by participants in our study with uses reported by participants in a study conducted by Cherubini and colleagues at Microsoft [4]. For the most part, the uses described in each study are similar. One minor discrepancy between the two studies is a use for navigation: eight participants in our study wanted to use a diagram when they return to a place in the code that they had visited previously (1a,1c,1d,1i,1j,1l, 3q,3r). This use may map to the documentation scenario in the study of Microsoft developers; however, the navigation case considers a potentially different time period and target for use than general documentation.

The results of our study affirm the results presented in the study by Cherubini and colleagues in two other ways. First, participants in both studies reported a prevalent use of whiteboards. Second, participants in both studies reported a *"need to understand both the microscopic details of the code and macroscopic conceptual structure" [1, p. 565]*.

This consistency of results between the two studies across multiple corporations improves the generalizability of the results.

5.3 Threats to Validity

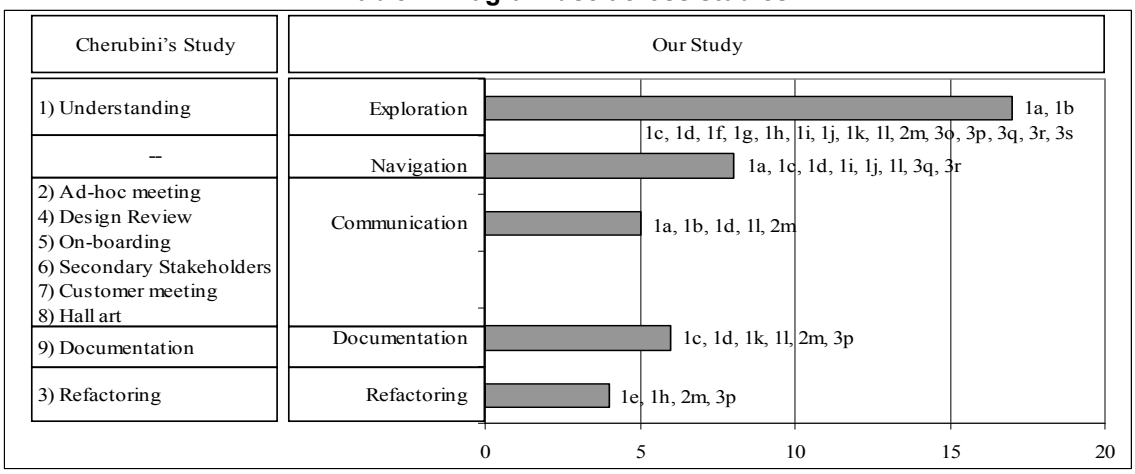
The results of our study are threatened by the limited number of participants. By including participants from different companies, we have attempted to mitigate this limitation by considering a wider range of work environments. Our results are also threatened by relying on a largely interview-based approach. Participants may be unable to easily reflect upon the potential importance of different diagramming support for different situations. Although we have gathered a breadth of different diagram support desired, we do not know the relative importance of different support for improving software development. As our intent was to perform an exploration study, we were more interested in generating hypotheses for future testing rather than attempting to rank the qualitative comments from the participants.

6. Conclusion

Programmers continue to work with largely textual representations of source code despite the fact these textual representations cannot easily convey the many graph-based relationships between pieces of source code. For example, programmers continue to work with lists of references between methods rather than interacting directly with a graph of method calls. This reliance on textual representations continues despite years of research into diagrammatic tools.

In this paper, we have reported on an exploratory study we conducted to understand how programmers use diagrams today and what diagramming support is desired in the future. We found substantial breadth in the support that the programmers desired in terms of the information content of the diagrams, the expected

Table 4. Diagram use across studies.



uses of the diagrams, and the actions that should be supported. The results of our study suggest that tool developers face substantial challenges in meeting the needs of programmers. Meeting these challenges will either require flexible, adaptive tool support or a more precise understanding of when and where diagrams are most useful for improving software productivity. The results of our study also affirm some of the findings of an earlier study of Microsoft developers in the ways that programmers desire to use diagrams.

Acknowledgments

The authors would like to acknowledge the thoughtful comments and time provided to our study by the participants. This work was funded in part by NSERC.

7. References

[1] B. de Alwis and G. C. Murphy, "Using Visual Momentum to Explain Disorientation in the Eclipse IDE," *IEEE Symp. on Visual Languages and Human-Centric Computing (VLHCC),* 2006, pp. 51-54.

[2] S. Bassil and R.K. Keller, "Software Visualization Tools: Survey and Analysis," *9th Int'l Workshop on Program Comprehension (IWPC),* 2001, pp.7-17

[3] F.P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, 20(4):10-19, 1987.

[4] M. Cherubini, G. Venolia, R. DeLine, and A.J. Ko, "Let's Go to the Whiteboard: How and Why Software Developers Draw Code," *ACM Conf. on Human Factors in Computing Systems (CHI)*, 2007, pp. 557-566.

[5] M. Cherubini, G. Venolia, and R. DeLine, "Building an Ecologically Valid, Large-scale Diagram to Help Developers Stay Oriented in Their Code," *IEEE Symp. on Visual Languages and Human-Centric Computing (VLHCC),* 2007, pp. 157-162.

[6] E.J. Chikofsky and J.H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, 7(1):13-17, 1990.

[7] A. Cox, M. Fisher and J. Muzzerall, "User Perspectives on a Visual Aid to Program Comprehension," *Int'l Workshop on Visualizing Software for Understanding and Analysis,* 2005, pp.70-75.

[8] A. Cox, M. Fisher and P. O'Brien, "Theoretical Considerations on Navigating Codespace with Spatial Cognition," *17th Workshop of the Psychology of Programming Interest Group*, 2005, pp. 92-105.

[9] M.T. Dishaw and D.M. Strong, "Supporting Software Maintenance with Software Engineering Tools: a Computed Task-Technology Fit Analysis," *Journal of Systems and Software*, 44(2): 107-120, 1998.

[10] M. Eichberg, M. Haupt, M. Mezini and T. Schafer, "Comprehensive Software Understanding with SEXTANT," *21st IEEE Int'l Conf. on Software Maintenance (ICSM)*, 2005, pp. 315-324.

[11] J. Grundy and J. Hosking, "Supporting Generic Sketching-Based Input of Diagrams in a Domain-Specific Visual Language Meta-Tool," *29th Int'l Conf. on Software Engineering (ICSE),* 2007, pp. 282-291.

[12] I. Hadar and O. Hazzan, "On the Contribution of UML Diagrams to Software System Comprehension," *Journal of Object Technology*, 3(1):143-156, 2004.

[13] T. Hendrix, J. Cross II, S. Maghsoodloo and M. McKinney, "Do Visualizations Improve Program Comprehensibility? Experiments with Control Structure Diagrams for Java," *31st SIGCSE Technical Symp. on Computer Science Education*, 2000, pp. 382-386.

[14] H.M. Kienle, and H.A. Müller, "Requirements of Software Visualization Tools: A Literature Survey," *4th IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis (VISSOFT),* 2007, pp.2-9.

[15] A.J. Ko, R. DeLine and G. Venolia, "Information Needs in Collocated Software Development Teams," *Int'l Conf. on Software Engineering (ICSE)*, 2007, pp. 344-353.

[16] R. Koschke, "Software Visualization in Software Maintenance, Reverse Engineering, and Reengineering: A Research Survey," *Journal on Software Maintenance and Evolution*, 15(2):87-109, 2003.

[17] T.D. LaToza, G. Venolia and R. DeLine, "Maintaining Mental Models: a Study of Developer Work Habits," *28th Int'l Conf. on Software Engineering (ICSE),* 2006, pp. 492-501.

[18] A. Von Mayrhauser, A.M. Vans, "Program Comprehension during Software Maintenance and Evolution," *Computer*, 28(8):44-55, 1995.

[19] M. Petre, A. Blackwell and T. Green, "Cognitive Questions in Software Visualization," *Software Visualization: Programming as a Multi-Media Experience,* MIT Press, 1997, pp. 453–480.

[20] S. Reiss, "The Paradox of Software Visualization," *3rd IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis (VISSOFT),* 2005, pp. 59-63.

[21] V. Sinha, D. Karger and R. Miller, "Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases," *IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC'06)*, 2006, pp. 187-194.

[22] M.-A.D. Storey, "Theories, Methods and Tools in Program Comprehension: Past, Present and Future," *13th International Workshop on Program Comprehension (IWPC)*, 2005, pp.181-191.

[23] M.-A.D. Storey, H.A. Müller, "Manipulating and Documenting Software Structures Using SHriMP Views," *11th Int'l Conf. on Software Maintenance (ICSM),* 1995, p.275.

[24] M.-A.D. Storey, F. Fracchia and H. Müllecr, "Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization," *5th Int'l Workshop on Program Comprehension (IWPC),* 1997, pp. 17–28.

[25] S. Tilley and S. Huang, "A Qualitative Assessment of the Efficacy of UML Diagrams as a Form of Graphical Documentation in Aiding Program Understanding," *21st Annual Int'l Conf. on Documentation*, 2003, pp.184-191.

[26] I. Zayour and T.C. Lethbridge, "A Cognitive and User Centric based Approach for Reverse Engineering Tool Design," 2000 conference of the Centre for Advanced Studies on Collaborative Research, 2000, pp.16-30.