Fisheyes in the Field: Using Method Triangulation to Study the Adoption and Use of a Source Code Visualization

Mikkel Rønne Jakobsen, Kasper Hornbæk
Department of Computer Science, University of Copenhagen
Njalsgade 128, Building 24, DK-2300 Copenhagen, Denmark
{mikkelrj,kash}@diku.dk

ABSTRACT
Information visualizations have been shown useful in numerous laboratory studies, but their adoption and use in real-life tasks are curiously under-researched. We present a field study of ten programmers who work with an editor extended with a fisheye view of source code. The study triangulates multiple methods (experience sampling, logging, thinking aloud, and interviews) to describe how the visualization is adopted and used. At the concrete level, our results suggest that the visualization was used as frequently as other tools in the programming environment. We also propose extensions to the interface and discuss features that were not used in practice. At the methodological level, the study identifies contributions distinct to individual methods and to their combination, and discusses the relative benefits of laboratory studies and field studies for the evaluation of information visualizations.

Author Keywords
Information visualization, evaluation methodology, field study, programming, fisheye view, experience sampling, logging, thinking aloud, interviews.

ACM Classification Keywords
H5.2. Information interfaces and presentation (e.g., HCI): User Interfaces (Evaluation/Methodology).

INTRODUCTION

An abundance of techniques and tools have emerged in the field of information visualization. In the past ten years, it has become increasingly common to see proposals for new techniques or tools accompanied by empirical evaluations of the usability and usefulness of the technique or tool. Not only do these evaluations provide useful information, they also testify to the maturation of the field.

The evaluation of information visualizations are mostly done as laboratory experiments [21]. Typically, participants spend an hour or two completing predefined tasks with a limited set of tools and data. Laboratory experiments allow precise measurement of the usability of a technique or tool, and extensive control of the extraneous factors that may influence use of the visualization.

However, laboratory experiments have general limitations [e.g., 3,26] and issues specific to information visualization also restrict their usefulness [e.g., 22,23,31,36]. Let us give just three examples; many others may be found in recent work on evaluation of information visualizations [e.g., 2,4,29]. First, the tasks used in a laboratory experiment greatly influence the results, but are often simpler than real life tasks [8,31]. Second, in real-life use visualizations have to be integrated with other tools and may not fit all activities or work habits equally well [11,17]; laboratory experiments rarely focus on integration with other tools. Third, laboratory studies often do not go beyond initial use of an interface [31]. An often-suggested answer to these issues is long-term studies that employ multiple methods [4,33,36]. While such studies exist, they are rare and advice about their design and benefits lacking.

The present paper studies a fisheye visualization of source code by deploying it among professional programmers for several weeks. While deployed, we collected data using experience sampling and logging; after participants gained proficiency, we interviewed them and analyzed videos of their use of the visualization. These data are used for method triangulation [7,25] so as to understand adoption and use, and are also contrasted to an earlier laboratory evaluation of the visualization [16]. The aim is twofold: (a) to advance our understanding of fisheye interfaces by studying their adoption and use in a real-life setting; to our knowledge this is the first long-term field study of a fisheye interface and (b) to discuss the methodology of evaluating information visualizations based on our use of method triangulation. The results will inform practical work on fisheye and other distortion interfaces, while advancing the discussion of how to evaluate information visualizations.

RELATED WORK

This paper aims to combine and make contributions within two themes: the methodology of information visualization evaluation and fisheye views for supporting programmers. Next we summarize the relevant literature for each theme.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CHI 2009, April 4–9, 2009, Boston, Massachusetts, USA.
Copyright 2009 ACM 978-1-60558-246-7/09/04...$5.00.

Evaluating Information Visualization

During the past ten years, evaluation of proposals for tools and techniques in information visualization has become commonplace [5]. For example, out of 16 papers at CHI 2008 with the keyword visualization or information visualization, 14 contained empirical evaluations (9 of which were laboratory studies). At the same time, however, methodology papers [4,36] and workshops [2] argue that solid evaluation of information visualizations is difficult.

The difficulties of evaluation of information visualizations may be illustrated with reference to laboratory experiments. Laboratory experiments are the most widely applied evaluation method [6,8,21] and perhaps therefore also the method with which the most difficulties have been identified. Difficulties include the use of experimental tasks that are markedly simpler than real life tasks. Also, durations of laboratory studies are often short. Perer and Shneiderman [30] reviewed a collection of information visualization papers and mentioned how only 39 out of 132 papers reported evaluations, and that all evaluations included less than 2 hours of tool use. Because participants need time to adopt novel interaction techniques [1], laboratory studies often do not address gaining proficiency beyond initial use of an interface [31]. In real-life, visualization techniques have to be integrated with other tools and may not fit all activities or work habits equally well [11,17]; such concerns are ignored in laboratory tasks. Other aspects of the setting in a laboratory and in realistic use contexts may impact performance and adoption. Reilly and Inkpen [32], for instance, studied the effectiveness of map morphing. They found differences in for instance recall when running a study in the lab and in a noisy public space. A final difficulty with laboratory experiments is that while the choice of participants are crucial to a laboratory experiment [8], non-professionals are often participants in such experiments. Taken together these difficulties limit the validity and generality of findings from laboratory studies.

One answer to the difficulty of laboratory experiments is new approaches to the evaluation of information visualizations. For instance, long-term studies of the use of information visualizations have been suggested [4,33,35,36]. Shneiderman and Plaisant [36] described multi-dimensional, in-depth long-term case studies, shortened to MILCs. Their proposal was used by Perer and Shneiderman [30], who developed a visualization for analyzing social networks. Perer and Shneiderman had domain experts use the visualization on their own problems, and followed a methodology that included training and changing the software in response to experts' needs. Other researchers have used variants of the MILCs approach [27,39]. While long-term studies may give unique insights, they are resource demanding and are, as an evaluation method, often more formative than summative.

Another answer has been methodologies based on self-reporting, such as diary studies and experience sampling [24]. One prominent example of this is insight-based evaluation [29,33], which aims to quantify the number and types of insights that analysts get using a visualization. Saraiya et al. [33] asked two biologists to use five visual tools to conduct exploratory analysis of microarray data sets, an actual work task for the biologists. For three months, the biologists were asked to keep a diary of their work process, the insights they gained from the data, and how the tools led to those insights. Saraiya et al. concluded that their study "indicates the viability and importance of a longitudinal, motivated, domain embedded, self-reporting approach to evaluating visualizations." A general problem with this methodology, however, is that it is hard to couple insights and the actual use of information visualizations.

Still another approach has been to systematically apply qualitative research methods, including systematic observation [15] and grounded theory [37]. For instance, Faisal et al. [9] used grounded theory to study a tool for visualizing academic literature. They argued that grounded theory helped them characterize users' experience of using visualizations.

Fisheye Interfaces as a Case

The specific focus of this paper is on fisheye interfaces [10]. We focus on this technique for two reasons. First, Lam and Munzner [23] remarked that "even though focus+context visualizations have been around for over 20 years, we do not know when, how, or even if they are useful"; the inconclusiveness of research on focus+context techniques includes fisheye interfaces. Second, while many evaluations have been conducted on fisheye interfaces [e.g., 1,13,14,34], we are unaware of any long-term studies. Also, most studies of fisheye interfaces use laboratory studies only [e.g., 1,16]. Thus, the benefits of the methodologies reviewed above have yet to bear on fisheye research.

We focus on fisheye use in programming. Programming is a challenging activity to support with a fisheye interface, but also to evaluate. It is cognitively complex and any insights from visualizations are likely to be secondary in relation to higher-level programming objectives. Two earlier studies presented relevant empirical insights. Jakobsen and Hornbæk [16] compared a fisheye view with a linear view of source code in a controlled experiment where 16 participants performed tasks involving navigation and understanding of source code. Results from the study suggest that a fisheye view can help programmers to navigate and understand source code. Kersten and Murphy [18] used diaries to investigate the utility of Mylar, an extension for the programming environment Eclipse, that allows the assignment of a degree of interest to interface elements. The diaries identified a range of changes to Mylar. Kersten and Murphy [19] later used logging to investigate if Mylar improved programmers' productivity.

In conclusion, a variety of methods are available to assist evaluation of fisheye interfaces. In particular, it seems that combinations of the evaluation methods proposed have not been tried in relation to fisheye interfaces.

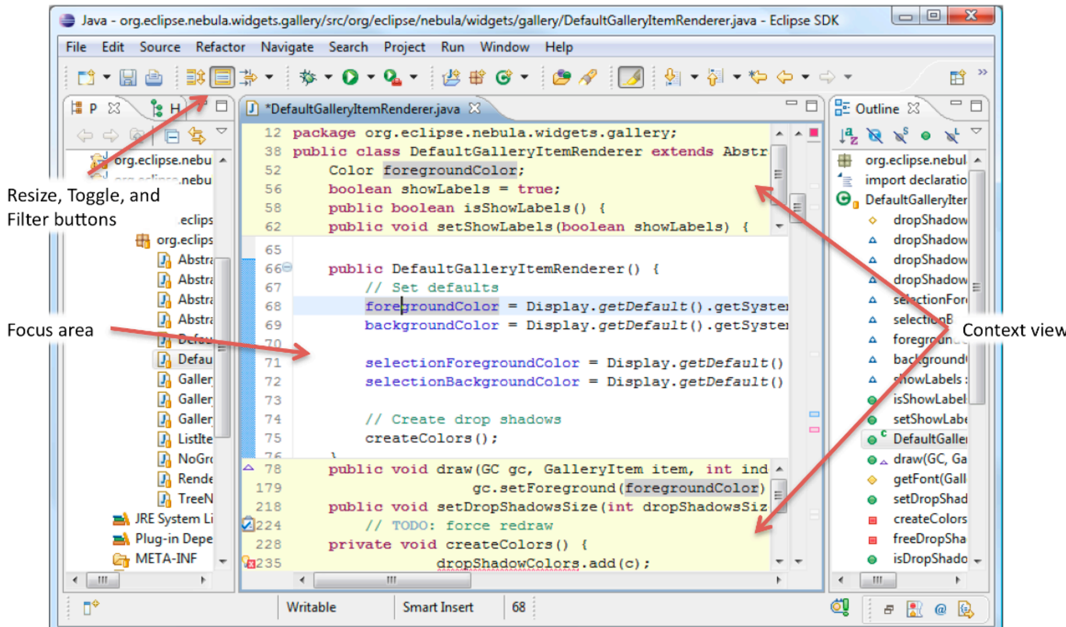

Figure 1: The Fisheye Java editor in Eclipse.

FISHEYE JAVA EDITOR

Navigating and understanding the source code of a program are highly challenging activities. The aim of our work is to support programmers in those activities using information visualization, specifically, fisheye interfaces [10]. Previous work has used laboratory experiments to show that fisheye interfaces may help navigation tasks [16]. As discussed in the related work section, such experiments are not entirely satisfactory. Before we describe our evaluation approach, this section introduces the fisheye editor that we evaluate.

Based on three years of development and experimentation, our current prototype looks like Figure 1. To be easily useful for real programming tasks, we have extended the Java editor provided in Eclipse, a widespread development environment, with a fisheye view. In the Fisheye Java editor[1], the editor window is divided into a focus area and a context view (see Figure 1). The focus area, the editable part of the window, is reduced to make room for the context view. The context view uses a fixed amount of space above and below the focus area. It contains a distorted view of source code in which parts of the source code that are of less relevance given the user's focus in the code, are elided.

The Fisheye Java editor contains all the features of the normal Java editor in Eclipse. For instance, the editor highlights annotations of different types, such as search results and compilation errors in the source code. One type of annotation called occurrences allows programmers to see where a variable, method, or type is referenced. For instance, a variable can be selected by placing the caret in the variable name whereby all references to that variable are highlighted in the source code. In an overview ruler shown to the right of the editor's scrollbar, rectangles indicate lines in the file that contain annotations. The Fisheye Java editor takes these annotations into account when selecting which lines to show in the context view.

Degree of Interest

In the Fisheye Java editor, a degree of interest (DOI) is determined for each program line in the file. Lines in the context view are then elided if their DOI is below a threshold k.

The DOI of a program line x given the focus point p (defined as the lines in the focus area) is calculated as:

$$\begin{aligned} DOI(x \mid p) = \; & enclosing(x, p) \\ & + annotated(x) \\ & + cursor(x) \\ & + sibling_{AST}(x, p) \\ & - d_{line}(x, p) \end{aligned}$$

First, lines are interesting if they contain declarations or statements that enclose the code visible in the focus area. Such lines contain a package, class, interface or method declarations, or one of the keywords for, if, while, switch, etc. If line x is such a line and it defines a block that encloses the code in the focus area p then $enclosing(x, p) = k$. Second, lines containing annotations, such as errors, search results, or occurrences of a selected element, are interesting. To provide context for annotations, lines that contain declarations of methods that enclose annotations are also of interest. Thus, $annotated(x) = k$ adds to the DOI of line x that contains an annotation or declares a method that enclose an annotation. Third, $cursor(x) = k$ adds to the DOI

[1] A Fisheye Java editor plug-in for Eclipse is available at http://mikkelrj.dk/projects/fisheye2009

of line x containing the editor caret, which may for instance be important for returning to the position of the caret. Fourth, lines that contain declarations of methods, fields or types that are close to the focus area may support orientation in the code. Thus, if line x declares a member of a class or interface that can be reached by moving upwards in the abstract syntax tree from a line in the focus area p then $sibling_{AST}(x, p) = k/2$. Fifth, a distance $d_{line}(x, p) \in [0; k/2]$ proportional to the number of program lines from line x to focus area p detracts from that line's DOI.

Source code elision in the context view

Lines are always included in the context view if they have a degree of interest above the threshold k. If there are not enough lines with $DOI > k$ to use all the space available in the context view, lines with $DOI <= k$ are added to the context view in descending order of DOI. This includes first declarations of methods or fields immediately above or below the code that is currently visible in the focus area, and then other lines directly adjacent to the focus area.

Placing the caret in a variable may cause many lines to have $DOI > k$ because they contain highlighted occurrences of the selected variable. All lines cannot be shown simultaneously in the fixed amount of space of the context view. Clipping or magnifying lines in the context view may result in some lines becoming unreadable, yet all lines may be important to the user. Thus, to guarantee users that the context view contains all highlighted occurrences, the windows containing the upper and lower context view can be scrolled. The context view automatically scrolls to show lines closest to the focus area when its contents change.

Interacting with the Fisheye Java editor

The user can interact with the focus area like a normal editor. The caret can be moved within the bounds of the focus area, scrolling the view contents when moving the caret against the upper or lower bound. The context view automatically reduces in size to fit the content; near the top of the document, for example, when the user scrolls by holding an arrow key to move the caret past the upper edge of the focus area, the upper part of the context view retracts. The context view can be switched on and off. When switched off, the context view can be call up temporarily with a keyboard shortcut, and it can be dismissed by hitting Esc or by clicking outside the context view. Clicking on a line in the context view jumps to that line and places the caret at the clicked position. Also, the context view can be resized, either by clicking on a button in the toolbar or by using a keyboard shortcut.

Filtering and customizing the context view

The user can change whether annotations or enclosing statements are included in the context view. Also, the user can select which annotations to show among all the annotation types available in Eclipse including bookmarks, errors, occurrences, search results, and tasks. In the example shown in Figure 1, errors and tasks are enabled, causing one line with an error and one line with a TODO task annotation to be shown in the context view. More customization options are available in a preference dialog page, for instance, whether to include the cursor line when it is scrolled out of view or whether to include all lines that contain method or variable declarations.

FIELD STUDY WITH PROFESSIONAL PROGRAMMERS

We conducted a field study of the Fisheye Java editor with professional Java programmers. Our aim was in part to understand how programmers will adopt a fisheye view of source code over two weeks and use it in their own work, in part to investigate the use of multiple methods in combination in a way not previously tried in evaluations of fisheye interfaces.

Participants

Ten professional Java programmers from three software companies participated in the study. Participants had between 1 and 20 (M = 9) years of programming experience. Eight participants had IT-related education whereas two participants had a business-oriented background. Participants used Mac OS X (2 participants) or Windows (7 participants) or both (1 participant), and they all used Eclipse 3.2 or later. All ten participants were male.

Method

We studied the programming activities of participants at their work place. Our aim was to study each participant using Eclipse for at least ten workdays; the actual period of study varied from two to five weeks. To provide a rich basis for analyzing the use of the Fisheye Java editor in the daily programming activities of participants, multiple data collection methods were used (see Figure 2). We were particularly inspired by Denzin's [7] definition of triangulation as "the combination of methodologies in the study of the same phenomenon" (p. 291) and by the lack of work that integrates the new evaluation approaches mentioned in the section on related work.

Two meetings were arranged to interview participants and observe them while thinking aloud during their daily work. In the period between the two meetings, data were automatically logged to describe participants' interaction with Eclipse. We probed participants during work using an adaptation of the experience sampling method [24]. Interviews, thinking aloud, logging, and probes complement each other to collect quantitative and qualitative, subjective and objective data; in the Discussion we return to how this worked in practice. Next, we describe in turn how each method was used.

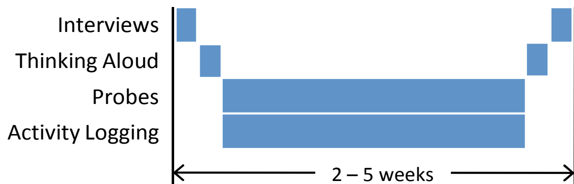

Figure 2: Use of methods to gather data about participants' programming activity and their experience using the Fisheye Java editor.

Thinking Aloud

We observed participants at their work place while they were thinking aloud, working with programming tasks that involved use of a Java editor. Because programming is a cognitively complex task – and because participants were working on real tasks – we only reminded participants to think aloud infrequently. To support a detailed analysis of how participants interacted with the Fisheye Java editor, we used screen recordings to capture participants' interactions with their computers, combined with a web camera that recorded participants' utterances. Screen recordings may be less obtrusive than using physical video equipment in participants' work environment and have been previously been used to record participants without an observer present [38], thus allowing a broad sample of the daily work of participants. In our case, however, we wanted participants to think aloud, so as to provide insights in their intent and experience of use. Thus, we wanted an author to be present and only recorded a couple of hours for each participant.

We analyzed the video recordings of participants thinking using grounded theory [37]. The first author found segments of recordings where participants either interacted with the context view using keyboard or mouse, or made utterances or gestures that indicated they were looking at information in the context view. We coded each segment where participants were (1) looking at the lines in the context view (and possibly scrolling the context view) or (2) clicking on a line in the context view to navigate to that line. In all, we recorded 10:41 hours of participants thinking aloud using Eclipse with the Fisheye Java editor installed. Technical problems with the recording software caused one thinking aloud session to yield no usable data.

Activity Logging

In the period of ten work days between the two thinking aloud sessions, data were automatically collected about (a) how participants used menus, toolbars, keyboard shortcuts and views in Eclipse, as in [28], and (b) how participants interacted with the Fisheye Java editor. We used these data to characterize participants' use of the programming environment, and in particular to describe how they interacted with the context view and how often they did so.

Probes

We collected data obtained using an adaptation of the experience sampling method [24], in which we randomly probed participants with a survey delivered in a dialog window from within the programming environment. Participants were probed during periods where user activity was registered in Eclipse and a Java editor was active. Interruptions were more than 90 minutes apart. Because we were interested in situations where participants used the context view, we delayed probes for up to 15 minutes to be delivered to participants the moment after they had interacted with the context view. The probe dialog window contained five pages asking participants (1) what they were doing when interrupted (using categories from [20]), (2) if they used the context view and if so, what they used it for, (3) how well they knew the source code they were working with, (4) what type of task they were working on (e.g., correcting a bug or restructuring the source code), and (5) how long they had been working on the task (ranging from "less than 10 minutes" to "more than a month").

Interviews

We interviewed participants before the first thinking aloud session to gather information about their background and programming experience, the project they are working on, and the types of task that they spend time on during their workday. After the second thinking aloud session, another interview was conducted to investigate the participants' experience of using the Fisheye Java editor. Also, the interview allowed for discussion of benefits and drawbacks of the editor and possible improvements. Recordings of the second interviews were transcribed and analyzed, using open coding and comparison of the coded interview segments to find common themes in participants' experiences of using the Fisheye Java editor.

Procedure

The experimenter met with participants at their workplace. First, participants were interviewed for about ten minutes. Next, the participant's computer was set up to capture the screen of the monitor showing the Eclipse window and a web camera was set up to record the participant while thinking aloud. Participants were then instructed to think aloud while they were working. Having observed the participant for approximately one hour of programming, the participant was allowed a break. A plug-in with the Fisheye Java editor was installed in Eclipse together with a plug-in for logging participants' interaction with Eclipse. The participant was instructed in the use of the Fisheye Java editor, and then supervised while trying the editor to allow for questions and clarifications. During the first five days of the study period, a window with instructions on how to use the Fisheye Java editor opened twice a day to remind participants about how to use the editor. Also, the first author visited or contacted participants to answer any questions participants might have about the Fisheye Java editor. Participants were not paid as an incentive for using the editor and they could at any time switch it off.

At the second visit about ten workdays after the first visit, participants were observed for an hour using Eclipse with the Fisheye Java editor installed. Participants were instructed to think aloud, and the session was recorded similarly to the first meeting. Finally, participants were interviewed about the work they had been doing after the first visit and about their experience with the editor.

RESULTS

Our results consist of recordings of participants' thinking aloud, logged data describing participants' activity in Eclipse, answers to probes, and interview transcriptions.

Thinking Aloud

Our analysis of participants' thinking aloud identified 55 incidents where the context view was used. We

characterized what was going on in each incident using open coding, and we compared incidents to develop categories for different uses of the context view and the situations where these uses occurred. Table 1 shows the most common situations of use with the number of incidents of each situation.

Most incidents involved the use of highlighted occurrences of a variable, method, or class. Often participants selected a method or variable to highlight its occurrences that would show up in the context view. Typically, participants found an occurrence and navigated there quickly or looked in the context view to investigate its dependencies, possibly clicking on an occurrence to investigate further. For instance, one participant had to move a set of buttons from one part of an application window to another. This task required navigating between at least four files, moving variables from one file to another. The participants used the context view, making sure all the dependencies either were moved along or dealt with in a more appropriate manner.

The second most common use of the context view involved looking for or navigating to the declaration of a method. In one situation, participants searched for the right method to use or investigate further. In another situation, participants navigated to a method they had recently investigated. Also, we found three incidents that resembled the situation of navigating to errors as part of manually refactoring code: after using the "quick-fix" tool in Eclipse to automatically add a required method to a class, participants looked in the context view to find the added method and navigate there.

The third most frequent use of the context view we saw involved navigating to compilation errors. In five incidents, participants made a change that caused errors in related code elsewhere and then immediately navigated to the error to correct it. A participant later explained that it was sometimes faster for him to add a parameter to a method and navigate to errors in calls to the method and fix them, than it was to use the refactoring tool in Eclipse. Also, in three incidents participants inspected an error that they had caused earlier without noticing.

We did not see participants use package declaration or enclosing statements in the context view, which surprised us because such higher-level information has been conjectured to provide important context [10]. A possible explanation is that participants were simply not working in long and complex blocks of code with heavy indentations, but mainly smaller methods or methods with many lines but no deep indentation.

In conclusion, we saw eight participants use the context view during thinking aloud. One participant had disabled the Java Fisheye editor because he experienced problems with it. Use of the context view varied greatly between participants; one participant mainly used the context view to inspect highlighted occurrences of variables, whereas another participant mainly used the context view for navigating to errors. This is not surprising, since the use situations we saw for each participant very likely were influenced by the tasks and the code that participants were working on in the small sample of each participant's work.

Activity Logging

The data that were logged in Eclipse comprise 114 days of Eclipse use. Each participant used Eclipse for at least ten days. However, no usable log file was produced for one of the participants due to technical problems.

From the logged interaction events, we determined periods where participants used Eclipse. A period was determined as at least two interaction events with less than five minutes in between, adding half a minute to the beginning and end of each period. In all, participants used Eclipse for around 370 hours. Using the method of determining periods of use, we determined and summarized the length of periods where participants made changes to the source code. Participants were editing Java source code for around 207 hours (56%), and each participant was editing code between 26% and 72% of the time they were working in Eclipse.

We visualized the participants' interaction with Eclipse by creating for each participant a series of timelines (one per day), indicating when the user was interacting with Eclipse

Use	*Situation*	*N*	*C*	*Example of an incident*
Use of highlighted occurrences	Inspected references to a variable	15	9	Determined which of the methods containing dependencies to a given variable that should be moved to another file.
	Inspected calls to a method	4	4	Located and navigated to a call to a given method and deleted the call.
Use of method declarations	Looked for a method	8	4	Looked for method that might be used to update a given type of object.
	Navigated to method that was recently investigated	3	3	Navigated to a method, which he had recently investigated, to copy code for use in the method he was currently writing.
	Navigated to a method that contained a TODO annotation	3	3	Navigated to a method that was just created automatically using a "quick-fix" tool.
Use of errors	Navigated immediately to correct error in related code	5	5	Manually refactored code in that he added a parameter to a method and then corrected a call to the method.
	Inspects an error caused by recent change	3	3	Noticed after digressing from a change to a method that the change was incomplete and returned to fix the error.

Table 1: Common situations involving use of the context view in the Fisheye Java editor identified in recordings of participants thinking aloud. N refers to the number of incidents of each situation and C refers to the number of those incidents where participants clicked on a line in the context view to navigate to that line.

and with the context view. Figure 3 shows an example of seven days of interaction for one user. The timeline visualizations gave three insights into the adoption and use of the context view. First, the use of the context view is evenly distributed over days. Only in 10% of the days, do participants not interact with the context view and then typically little interaction with Eclipse occurs in the day. Also, interaction with the context view typically happens several times during the day (in about 90% of the days). Second, we do not see a decline of use over time. Across participants, a comparable number of uses of the context view are found on the first and last day of logging. Third, some participants have long durations of activity where they do not use the context view (in Figure 3 this happens at the middle part of day 7 and the beginning of day 8). This typically happens when the participant is not editing. Overall, the time lines show that participants have very different work patterns. For instance, one participant who was filling in for the project leader during the study had many short periods of interacting with Eclipse during his workday and only few long periods of programming.

As a measure of how frequently participants used the context view, we grouped the times where participants scrolled or clicked in the context view into periods so that repeated interaction with the context view within a five-minute window counted as a single period of use. In average, participants interacted with the context view 1.7 times per hour. For comparison, we determined how often common tools in Eclipse for searching and navigating in the current file were used. In average, participants used 'Find' 0.7 times per hour, an outline of the file 2.3 times per hour, and a search for references 1.4 times per hour.

Probes

In all, participants were probed 332 times (out of which 193 were postponed and not analyzed further). We discarded six probes that participants completed more than five minutes after the interruption, because we did not think those answers reliably reflected a participant's experience at the time of interruption. Of the resulting 133 probes, 36 were conditional probes (that were made because participants had just interacted with the context view) and 14 were unconditional probes where participants reported that they had used the context view. Thus, 50 probes were answered after participants had used the context view.

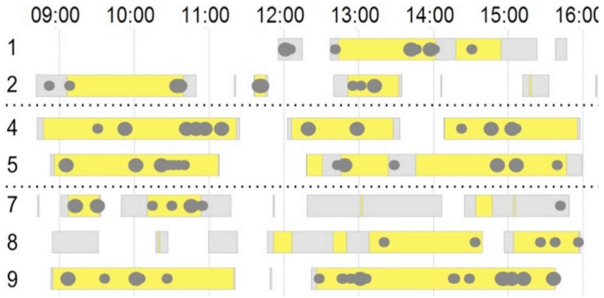

Figure 3: Example of timeline visualization of seven days (y-axis) of interaction with Eclipse and the Fisheye Java editor. Periods of editing are yellow; periods of interaction with Eclipse are gray; gray circles indicate use of the context view.

Table 2 shows the activities that participants reported they were doing when probed. The most frequent activities participants mentioned doing when probed were editing (54%), reading code (20%), or testing (17%). Other activities that participants reported doing when probed mainly included forward porting (8%), just starting or resuming work in Eclipse (6%), or synchronizing (4%). Participants report more often that they navigated dependencies in the code when they had used the context view, than when they had not used the context view, and participants reported navigating when they had used the context view only in conditional probes. This suggests that participants used the context view to navigate, but also that navigating dependencies is a brief activity that only few unconditional probes interrupted.

The tasks that participant most frequently reported working on when probed were extending the program with new functionality (27%), modifying the program's existing functionality (23%), or fixing a bug (23%). When using the context view, participants reported slightly more often that they were fixing bugs (28% vs. 20%) or extending the program (54% vs. 40%) compared to when they were not using the context view. In contrast, they reported less often optimization (0% vs. 10%) or restructuring (4% vs. 16%) when using the context view.

When probed after using the context view, participants had used it to find highlighted occurrences (18), navigated to a particular line (9), see the declaration of the current class and method (8), and see enclosing statements (6).

Interviews

Table 3 summarizes the main findings from analysis of our interviews with participants after they had used Eclipse with the Fisheye Java editor installed. Concerning adoption of the fisheye view, eight participants said they would continue to use the Fisheye Java editor. One participant explicitly said that it was "a better editor with the fisheye view than it is without". We think this is a strong indication that participants found the benefits of the fisheye view to outweigh the drawbacks. Furthermore, six participants felt

	Use of context view		No use
Probe was…	conditional	unconditional	unconditional
	N = 36	N = 14	N = 83
Read code	22%	7%	22%
Edit	53%	79%	51%
Navigate	39%	0%	4%
Search	11%	7%	4%
Test	17%	7%	19%
Read API docs	0%	0%	4%
Other	14%	7%	23%

Table 2: Frequency of activities participants answered they were doing when probed (1) conditionally when context view was used, (2) unconditionally when context view was used, and (3) unconditionally when context view was not used. Multiple activities could be specified, so columns do not sum to 100%.

they had not fully learned and adopted the fisheye view after the few weeks of having it installed. Altogether, we took this to mean that some participants would at least keep the fisheye view installed so as to try to learn using it.

Concerning the overall experience of using the fisheye view, six participants found it was confusing at times, because it was hard to know what was shown in the context view. Three reasons were mentioned: (1) adjacent lines that filled unused space in the context view made it difficult to determine where blocks of code were left out, (2) not all methods declared in the file were shown, and (3) different types of lines were shown at different times. Five participants said they disabled or did not care about the fisheye view when working in tasks where it was not useful. Also, four participants said they had reduced the size of the context area. Some comments suggest that it is not so much the context area that is too large as it is the focus area that is too small to get an overview of the code in focus. Some participants mentioned that they would have liked a taller display, and one participant had in fact pivoted his widescreen display to use the Fisheye Java editor in a tall window. Three participants made comments suggesting that they sometimes would forget that the context view was there but the visually distinct appearance of an error or an occurrence could draw their attention to it.

Seven participants said they liked that errors and occurrences were shown in the context view. One reason mentioned was: "you learn 400 different shortcuts for example to navigate between different compiler errors, so I think it's a good thing that you actually have something visual". In particular, comments of two participants seem to hint that being able to see in the context view the errors – noting that some errors follow from others – helps them determine what code to actually fix to correct the errors. Participants did not agree about the usefulness of class/method declarations or of enclosing statements. While some participants found enclosing statements useful to form a context for the code in focus, others said that they made no use of them. One participant, who liked the enclosing statements, admitted that he once experienced losing overview of a large method anyway. Finally, three participants said they used the context view to see methods that were near the code in focus.

DISCUSSION

Findings on Fisheye Interface in Programming

The main finding is that the fisheye interface was adopted by participants and integrated in their work. The activity logging showed that most participants used the context view regularly throughout the study and that the frequency of use was comparable to core tools in Eclipse. Most participants said they would continue to use the Fisheye Java editor after the study had finished. Compared to some other studies of workplace adoption of information visualization [e.g., 11], this is a strong and encouraging result; in relation to fisheye research [e.g., 1,13,14,34], the adoption suggest that some ideas in fisheye interfaces may be useful in real-life tools for tasks as complex as programming.

While adoption is thus confirmed by several types of data, some programming tasks were not supported by the Fisheye Java editor. In interviews, participants said that the fisheye interface did not support tasks like debugging or composing new code. The activity logging also shows long episodes of non-use of the context view. While our notion of focus point was tied to one editor window, participants' focus could easily change between windows or other parts of the editor. We contend that extending the notion of focus in fisheye interfaces to encompass different parts of the interface (similarly to Mylar [18]) could be interesting for real-life fisheye interfaces. On the other hand, the thinking aloud sessions showed use of the fisheye interface across a range of tasks, including some surprising ad-hoc uses.

The usefulness of the Fisheye Java editor was linked to the highlighted occurrences of variables and methods. Most incidents of use of the context view in thinking aloud sessions involved highlighted occurrences; a third of the probes following use of the context view also mention

Theme	*Main finding*	*N*	*Examples or quotes*
Learning and adopting	Continued use after the study	8	"I actually think it is good. At least, I have decided to keep it." "I'm not ready to disable it yet"
	Not fully learned and adopted fisheye view	6	"Have been busy … so, often I have relied on habits …"
Overall experience	Content sometimes confusing	6	"… difficult to know when it was shown and when it was not."
	Not relevant for some tasks	5	Debugging. Writing new code. Copying large code blocks.
	Large view of code in focus	4	"It's like [the context view] can use a certain percentage [of the display space], and then it starts to become noise."
	Unobtrusive	3	"It is kind of unobtrusive, you can easily forget it is there …" Draws attention when something stands out visually.
Use of context view	Inspected and navigated to occurrences	7	"Often faster to use the fisheye view to jump because it shows it … rather than page down or search"
	Determined where to correct errors	4	"[In the overview ruler] you can just see there is an error, but down here you can see there is an error *and* what the error is."
	Awareness of nearby methods	3	"[Used it] to see nearby methods, click on one and jump in the code."

Table 3: Main findings from analysis of interviews. N refers to the number of interviews in which a finding was made.

highlighted occurrences. While the DOI function underlying the fisheye editor integrates different kinds of interest, it appears that the direct and transparent relatedness of highlighted occurrences in the editor and in the context view matters the most to users. More generally, the a priori determined components of the DOI function may matter relatively less in real-life use. This speculation brings into doubt a defining characteristic of fisheye interfaces, and is an important focus for future work.

The last finding we want to emphasize is a lack of clarity and predictability in the fisheye interface. Six participants mentioned in interviews that they were confused about when methods and lines were shown and when they were not (e.g., "it should be more predictable so that you can guess what you get or understand better what information you get from the fisheye"). These remarks warrant further investigation, because they conflict with another defining characteristic of fisheye interfaces [10], namely that the view changes based on changes in the focus point. We are considering how to make it clearer which lines are shown in the fisheye interface and which lines are elided. A possible improvement is to allow users to control directly in the fisheye interface how different types of information in the context view are shown or elided, perhaps using fold and unfold mechanisms (e.g., ▽ and ▷) used in widespread code editors.

Strengths of Methods in Combination

We found individual methods contributing insights into *adoption*, *use* of specific functions, and participants' *intent* to varying degrees. In combination, the methods provide stronger evidence of participants' adoption and use of the Fisheye Java editor than any method alone, making up for limitations of individual methods. We give three examples.

First, interviews provide subjective data where participants explain their full experience and intent, but explanations are retrospective and hard to connect to concrete situations in their work and specific functions in the Fisheye Java editor. In contrast, thinking aloud provides rich insight into participants' programming activity based on concrete use situations.

Second, participants' assessments in interviews of their adoption of the fisheye interface are retrospective and thus ambiguous. Also, observing each participant a few hours provides only a small sample of their work and it is difficult to tell if participants have adopted and used the Fisheye Java editor in all their work activities based on thinking aloud data. To compensate for these limitations, activity logging provides quantitative, fine-grained data about hundreds of hours of work that show that participants used the fisheye interface regularly. Also, probes provide subjective data about many hours of participants' work that show how participants used the fisheye interface in different types of activity. These data allow us to extrapolate on our observations of participants' use of the fisheye interface in their work across tasks.

Third, determining participants' intent during uses of the fisheye interface solely from activity logging and probes is difficult, if not impossible: activity logging does not give the context of participants' work, nor their intent with the logged activity; interruptions by probes annoy participants and only limited data can be gathered. Thinking aloud thus complements logging and probes by situating use of the fisheye interface in observations of participants.

Laboratory Experiment vs. Field Study

We find four comparisons between the previous laboratory experiment [16] and the present field study of interest. First, our focus on adoption is not possible in a laboratory experiment [12]. The data provided by activity logging is much more convincing than our earlier collected preferences. Second, while realism of tasks is often claimed a hallmark of field studies, we were mostly surprised by the variability and ad hoc use of the fisheye view, as captured in the thinking aloud sessions. Because tasks were fixed and relatively simple in the laboratory study, we did not see such behavior. Third, as mentioned earlier, a common criticism of laboratory studies is that they do not allow participants to gain proficiency [31]. The present field study is not a panacea in that respect. Participants mention time-pressure and being busy as barriers to using the fisheye editor. Perhaps proficiency with tools need other forms of collaboration between researchers and participants, for instance, the long-term collaborations in MILCs [36]. Fourth, the field study required full integration of the editor in participants' programming environment, causing a number of practical problems.

CONCLUSION

Fisheye interfaces for source code promise to support programmers in navigating and understanding code. Such interfaces, however, have only been evaluated in laboratory experiments, leaving it uncertain if they would be adopted and used in real-life programming. This uncertainty reflects a general lack of multi-method longitudinal studies of information visualizations. We have conducted a field study of ten professional programmers solving their normal work tasks using a fisheye editor. Data were collected using experience sampling, activity logging, thinking aloud, and interviews.

The results suggest that participants adopted and used the fisheye interface as extensively as other common tools in their programming environment. However, lack of predictability was an integral part of using the interface, certain activities were not supported well, and core assumptions in the design of fisheye interface (which had not been challenged in a previous laboratory study) did not hold in the field. Methodologically, we have shown how triangulation of data helps reach closure about benefits and limitations of the visualization. Future work could couple more tightly the data collection methods so as to obtain data both on adoption, specific episodes of use, and on users' intent.

REFERENCES

1. Baudisch, P., Lee, B., & Hanna, L. Fishnet, a Fisheye Web Browser With Search Term Popouts: a Comparative Evaluation With Overview and Linear View, *Proc. AVI 2004*, ACM Press (2004), 133-140.
2. Bertini, C., Plaisant, C., & Santucci, G. Rewind: BELIV'06: Beyond Time and Errors; Novel Evaluation Methods for Information Visualization., *Interactions*, *14*, 3 (2007), 59-60.
3. Bracht, G. H. & Glass, G. V. The External Validity of Experiments, *American Educational Research Journal*, *5*, 4 (1968), 437-474.
4. Carpendale, S., Evaluating Information Visualizations, in Kerren, A., Stasko, J. T., Fekete, J.-D., & North, C. (eds.) *Information Visualization: Human-Centered Issues and Perspectives,* Springer, 2008, 19-45.
5. Chen, C. & Czerwinski, M. P. Special Issue on Empirical Evaluation of Information Visualizations, *International Journal of Human-Computer Studies*, *53*, 5 (2000).
6. Chen, C. & Yu, Y. Empirical Studies of Information Visualization: A Meta-Analysis, *International Journal of Human-Computer Studies*, *53*, 5 (2000), 851-866.
7. Denzin, N. *Sociological Methods: A Sourcebook*, McGraw Hill, New York, 1978.
8. Ellis, G. & Dix, A. An Explorative Analysis of User Evaluation Studies, *Proc. BELIV'06 - Beyond Time and Errors: Novel Evaluation Methods for Information Visualization - Workshop of AVI'06*, (2006), 15-20.
9. Faisal, S., Craft, B., Cairns, P., & Blandford, A. Internalization, Qualitative Methods, and Evaluation, *Proc. BELIV'08*, ACM Press (2008), 45-52.
10. Furnas, G. W. The Fisheye View: A New Look at Structured Files.,*Bell Laboratories Technical Memorandum #81-11221-9*, Morgan Kaufmann, (1981). 312-330.
11. González, V. & Kobsa, A. A Workplace Study of the Adoption of Information Visualization Systems, *Proc. I-KNOW 03*, (2003), 96-102.
12. Greenberg, S. & Buxton, B. Usability Evaluation Considered Harmful (Some of the Time), *Proc. CHI'2008, ACM Press* (2008), 111-120.
13. Gutwin, C. Improving Focus Targeting in Interactive Fisheye Views, *Proc. CHI'2002*, ACM Press (2002), 267-274.
14. Hornbæk, K. & Hertzum, M. Untangling the Usability of Fisheye Menus, *ACM Transactions on Computer-Human Interaction*, *14*, 2 (2007).
15. Isenberg, P., Tang, A., & Carpendale, S. An Exploratory Study of Visual Information Analysis, *Proc. CHI 2008*, ACM Press (2008), 1217-1226.
16. Jakobsen, M. R. & Hornbæk, K. Evaluating a Fisheye View of Source Code, *Proc. CHI 2006*, (2006), 377-386.
17. Jakobsen, M. R. & Hornbæk, K. Transient Visualizations, *Proc. OZCHI 2007*, ACM (2007), 69-76.
18. Kersten, M. & Murphy, G. Mylar: a Degree-of-Interest Model for IDEs, *Proc. AOSD*, (2005), 159-168.
19. Kersten, M. & Murphy, G. Using Task Context to Improve Programmer Productivity, *Proc. SIGSOFT 2006*, ACM Press (2006), 1-11.
20. Ko, A. J., Aung, H., & Myers, B. A. Eliciting Design Requirements for Maintenance-Oriented IDEs: a Detailed Study of Corrective and Perfective Maintenance Tasks, *Proc. ICSE'05*, ACM Press (2005), 126-135.
21. Komlodi, A., Sears, A., & Stanziola, E. Information Visualization Evaluation Review, *SRC Tech. Report, Dept.of Information Systems, UMBC. UMBC-ISRC-2004-1* (2004).
22. Kosara, R., Healet, V., Interrante, V., Laidlaw, D. H., & Ware, C. Thoughts on User Studies: Why, How, and When, *Computer Graphics and Applications*, *23*, 4 (2003), 20-25.
23. Lam, H. & Munzer, T. Increasing the Utility of Quantitative Empirical Studies for Meta-Analysis, *Proc. BELIV'08*, ACM Press (2008)
24. Larson, R. & Csikszentmihalyi, M. The Experience Sampling Method, *New Directions for Methodology of Social and Behavioral Science*, *15* (1983), 41-56.
25. Mackay, W. E. Triangulation Within and Across HCI Disciplines, *Human-Computer Interaction*, *13*, 3 (1998), 310-315.
26. McGrath, J. E., Methodology Matters: Doing Research in the Behavioral and Social Sciences, in Baecker, R. M., Grudin, J., & Buxton, W. A. (eds.) *Human-Computer Interaction: Toward the Year 2000,* Morgan Kaufmann, 1995, 152-169.
27. McLachlan, P., Munzer, T., Koutsofios, E., & North, S. LiveRAC - Interactive Visual Exploration of System Management Time-Series Data., *Proc. CHI 2008*, ACM Press (2008), 1483-1492.
28. Murphy, G., Kersten, M., & Findlater, L. How Are Java Software Developers Using the Eclipse IDE?, *IEEE Software*, *23*, 5 (2006), 76-83.
29. North, C. Visualization Viewpoints: Toward Measuring Visualization Insight, *IEEE Computer Graphics and Applications*, *26*, 3 (2006), 6-9.
30. Perer, A. & Shneiderman, B. Integrating Statistics and Visualization: Case Studies of Gaining Clarity During Exploratory Data Analysis, *Proc. CHI 2008*, ACM Press (2008), 265-274.
31. Plaisant, C. The Challenge of Information Visualization Evaluation, *Proc. AVI 2004*, (2004), 109-116.
32. Reilly, D. F. & Inkpen, K. M. White Rooms and Morphing Don't Mix: Setting and the Evaluation of Visualization Techniques, *Proc. CHI 2007*, ACM Press (2007), 111-120.
33. Saraiya, P., North, C., Lam, V., & Duca, K. An Insight-Based Longitudinal Study of Visual Analytics, *IEEE Transactions on Visualization and Computer Graphics*, *12*, 6 (2006), 1511-1522.
34. Schaffer, D., Zuo, Z., Greenberg, S., Bartram, L., Dill, J., Dubs, S., & Roseman, M. Navigating Hierarchically Clustered Networks Through Fisheye and Full-Zoom Methods, *ACM Trans. on Computer-Human Interaction*, *3*, 2 (1996), 162-188.
35. Seo, J. & Shneiderman, B. Knowledge Discovery in High-Dimensional Data: Case Studies and a User Survey for the Rank-by-Feature Framework, *IEEE Transactions on Visualization and Computer Graphics*, *12*, 311 (2006), 322.
36. Shneiderman, B. & Plaisant, C. Strategies for Evaluating Information Visualization Tools: Multi-Dimensional In-Depth Long-Term Case Studies, *Proc. BELIV'06*, (2006), 1-7.
37. Straus, A. & Corbett, J. *Basics of Qualitative Research. Techniques and Procedures for Developing Grounded Theory*, Sage., Thousand Oaks, CA, 1998.
38. Tang, J. C., Liu, S. B., Muller, M., Lin, J., & Drews, C. Unobtrusive but Invasive: Using Screen Recording to Collect Field Data on Computer-Mediated Interaction, *Proc. CSCW'06*, ACM Press (2006), 479-482.
39. Valiati, E. R., Freitas, C. M., & Pimenta, M. S. Using Multi-Dimensional In-Depth Long-Term Case Studies for Information Visualization Evaluation, *Proc. BELIV'08*, ACM Press (2008), 1-7.