Evaluating a Fisheye View of Source Code

Mikkel Rønne Jakobsen & Kasper Hornbæk
Department of Computing
University of Copenhagen
Copenhagen East, Denmark
mikkelj@acm.org, kash@diku.dk

ABSTRACT
Navigating and understanding the source code of a program are highly challenging activities. This paper introduces a fisheye view of source code to a Java programming environment. The fisheye view aims to support a programmer's navigation and understanding by displaying those parts of the source code that have the highest degree of interest given the current focus. An experiment was conducted which compared the usability of the fisheye view with a common, linear presentation of source code. Sixteen participants performed tasks significantly faster with the fisheye view, although results varied dependent on the task type. The participants generally preferred the interface with the fisheye view. We analyse participants' interaction with the fisheye view and suggest how to improve its performance. In the calculation of the degree of interest, we suggest to emphasize those parts of the source code that are semantically related to the programmer's current focus.

Author Keywords
Fisheye view, information visualization, programming, Eclipse, user study

ACM Classification Keywords
H.5.2 [Information Interfaces and Presentation (e.g., HCI)]: User Interfaces

INTRODUCTION
Programming is a complex human activity. The programmer is typically required to develop correct source code from a general description of how a program should work. As the source code grows in size and complexity, the navigation between and within the files comprising the source code becomes mentally demanding. In addition, the programmer must continually switch between writing new code and understanding existing code, possibly constructed by other persons. Extensive research has aimed to find ways of supporting the programmer in these activities [11, 12, 17].

CHI 2006, April 22-27, 2006, Montréal, Québec, Canada.

One approach to supporting navigation and understanding of source code is information visualization [10, 14]. The first instance of such an approach was probably Furnas's fisheye views [5]. In fisheye views, all source code lines are assigned a degree of interest calculated from their a priori importance and their relation to the line of source code in focus. Lines with a degree of interest below some threshold can thus be removed or rendered at smaller sizes for a view that contains both details and context. Fisheye views promise to integrate pertinent information in just one view; information that in state-of-the-art programming environments like Eclipse, NetBeans and Visual Studio are presented in separate windows or require explicit action on part of the user.

The benefits of applying fisheye views to programming have not been examined empirically. Empirical studies of fisheye views in other domains have shown positive results, for example [15], but have also shown high task completion times [6], interference with users' ability to remember the location of objects [16], and low incidental learning [8].

This paper presents an extension of a widely used open-source development environment with a fisheye view of source code. The design of the fisheye view is described, as are the underlying decisions. We present an empirical evaluation of the fisheye view that emphasizes both measures of usability and analysis of interaction patterns. Based on the evaluation, we suggest potential improvements to the algorithms and user interface design underlying the fisheye view. We discuss in particular the algorithm used to calculate the degree of interest; this is relevant not only for fisheye views of source code, but also for the general notion of fisheye views and for fisheye interfaces in other domains.

RELATED WORK
Fisheye views have been used to visualize source code and programs at different levels of detail. The SHriMP system, for example, uses fisheye views on graph representations of the program structure [18]. Turetken et al. [19] described how to use fisheye views of models used in systems analysis and design. Below, however, we discuss only the use of fisheye views, and distortion techniques more generally, on a single file of source code. In addition we discuss empirical evaluations of applying fisheye views to other types of mainly textual data, such as electronic documents and web pages.

Furnas [5] defined a general case fisheye view and suggested that it could be applied to source code, so as to display con-

text information in addition to the lines of source code that the programmer focuses on. To create such a view, lines of source code are assigned a degree of interest based on (1) their level of detail, or a priori importance, and (2) their distance from the user's focus (e.g., the currently selected line of source code). The level of detail is determined from the hierarchical structure of the program, as given by the indentation of source code lines. Thus, enclosing conditional- or loop-statements are considered of greater general interest than highly indented lines. Likewise, local details are considered more interesting than remote details: lines indented on the same level and in the same block as the line in focus are considered of high interest to the user, while lines in other blocks are considered less interesting. Furnas's fisheye view hides program lines with a degree of interest below a certain threshold. The display space gained from hiding parts of the source code provides for contextual information (i.e., lines of source code with a high degree of interest not visible in a traditional view). Furnas argued that the fisheye view, in virtue of its combination of program lines close to the focus and higher-level information, would show the lines of greatest interest to the programmer, thereby facilitating programming.

Furnas's paper left unanswered several questions about the implementation of fisheye views for source code. Below we discuss these questions to outline related work; the remainder of the paper may be seen as an attempt to answer them. One question concerns the use of display space in the fisheye view, in particular how to handle a large amount of lines with the same high degree of interest. Koike [9] proposed to keep the total amount of information displayed (i.e., the number of source code lines) constant, and presented an algorithm that usually, but not always, fills the available space. No general answer to this question is therefore available.

Another question concerns how to establish the user's focus in the source code, needed to calculate the distance component of the degree of interest function. In Furnas's paper the focus is given by the currently selected line. It is not obvious, however, that the focus need be only one line, nor clear how to determine the focus in situations where the user interacts with the source view using a mouse. An alternative to the fisheye view, source code elision, requires the user to manually fold and unfold blocks of program lines, thus avoiding the issue of defining the focal point. In Jaba [3], for example, methods in Java classes were elided, diminishing the bodies of methods while displaying the method signature lines in normal size. An empirical study by Cockburn and Smith [3] showed that such elision may improve navigation tasks in programming. However, the cost of the user's direct manipulation of the view may in practice prove to outweigh the benefits of elision. The experimental tasks used in Cockburn and Smith's study were simple and required little use of the folding mechanism, leaving this question unanswered.

A third question concerns whether we can utilize richer information about the program structure than Furnas did, that is, enhancing the degree of interest function beyond using just indentation level. One technique to distort the source code that does this is program slicing. Program slicing was first described as a method used by programmers for reducing the amount of code to look at when debugging or trying to understand programs [20]. Program slicing limits the view of the source code to those program lines which affect the value of a specific variable. Tools for performing slicing automatically have been found useful in debugging [21]. However, program slicing most often uses only variables to slice the source code, not the structure of the source code, as Furnas did. The choice of which variables to slice is usually left to the user. In contrast to the intention of fisheye views, this requires explicit and deliberate action on part of the user.

Yet another question concerns how to embed a fisheye view in a source code editor, using the tools available in modern integrated development environments. As pointed out by Koike [9], the focus may change continually when a user edits source code. The effect of such changes on a straightforward implementation of fisheye views would probably be visually very complex. This begs the question how the user's interaction with the editor affects the view, and how often the view should be updated when the user's focus changes. In addition, the effects of editing (e.g., pasting or typing) source code on the visual presentation is not treated in discussions of fisheye views familiar to us.

We are aware of no evaluations of how fisheye views affect programming at the level of interacting with individual files of source code. However, the use of fisheye views on electronic documents and web pages has been investigated empirically. Paez et al. [13] conducted an empirical study of electronic documents where the font size was bigger for the title, headings, and key sentences compared to other parts of the document. Initially, the entire document was fitted on the screen, and the user could zoom in on interesting sections. The empirical study did not find this interface to perform better than hypertext on measures of time, but did find some positive user reactions towards the zooming interface. Hornbæk and Frøkjær [8] compared a fisheye interface for electronic documents to overview+detail and linear interfaces. In a task that required participants to read documents as a basis for writing essays, the fisheye enabled subjects to quickly get an overview of and read the documents. Afterwards, however, participants were able to answer fewer questions about the content. Fishnet [1] extended a web browser with a fisheye view by using a bifocal display in which the context area was compressed, while search terms were kept readable and highlighted. In an empirical study, Fishnet was found to improve certain web search tasks, depending on the organization of the web page. However, only 3 out of 13 participants preferred the fisheye interface.

In summary, Furnas's original paper and related work have only partly addressed the questions regarding how to implement fisheye views for source code. Additionally, we find no studies that have investigated empirically how fisheye interfaces for source code work; studies of fisheye views for electronic documents and web pages show mixed results. The remainder of the paper therefore explores answers to

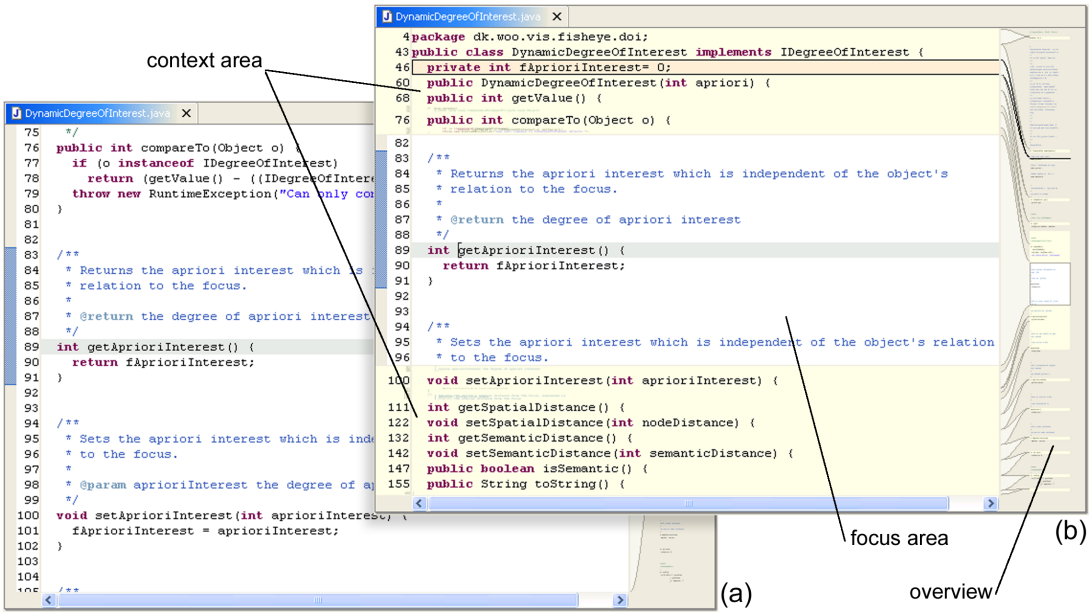


Figure 1. Screenshots of (a) the Linear and (b) the Fisheye interface showing the same source file of 161 lines.

the questions raised, and provides an empirical evaluation of our implementation of a fisheye view.

A FISHEYE VIEW OF SOURCE CODE

To investigate the questions above, we explored a number of alternative designs for fisheye views of source code. Figure 1 (b) shows our preferred design, which we refer to as the Fisheye interface. Below we explain the design, and compare it with a baseline linear interface shown in Figure 1 (a).

Both interfaces include an editor, implemented as a plugin in Eclipse, an extensible development environment[1]. The plugin extends the Java editor included in Eclipse's Java Development Tools. All features except line numbers and syntax highlighting of the source code are disabled in the editor. Both interfaces use an overview+detail approach in which an overview of the entire document is shown to the right of the detail view window; previous research [8] suggests such an interface superior to using just a detail view. The detail view shows a part of the document that the user has selected. The overview shows the source code reduced in size to fit the entire document within the space of the overview area; the standard source code highlighting is preserved. The text is unreadable, but it is possible to discern structural features such as method boundaries and blocks of javadoc comments. The part of the document shown in the detail view is visually connected with its position in the overview by lines. The overview supports the mouse interaction normally expected from a scrollbar; the thumb can be dragged to scroll the detail view and clicking above or below the thumb scrolls the detail view one page up or down.

The above features are common to the interfaces; the next four sections describe the design of the Fisheye interface. The plugin can be downloaded from the authors' web sites.

[1]http://www.eclipse.org

Focus and Context Area

In the Fisheye interface, the detail view of the source code is divided into two areas: the focus area and the context area. The total available space is evenly divided between the two areas. The editable part of the view, the focus area, is reduced in size to accommodate a context area. The context area uses a fixed amount of space above and below the focus area. It contains a distorted view in which certain parts of the source code, being of less relevance given the focus point, are diminished or elided. The focus point is defined as all lines visible in the focus area. Thus, the context area is updated when the user scrolls the view, and remains unchanged when the user moves the caret within the bounds of the focus area. Our design hereby circumvents the issues raised earlier concerning how often the focus changes and the potential problems of frequently updating the view.

Degree of Interest Function

A degree of interest (DOI) function determines if and how much the lines are diminished in the context area. The degree of interest for a program line x given the focus point p is calculated as:

$$DOI(x|p) = API(x) - D_{syntactic}(p,x) - D_{semantic}(p,x)$$

First, the DOI function is based on an a priori interest (API) component defined by (a) the type of program line for which the degree of interest is currently being calculated and (b) that line's indentation level. The type of a program line is determined by deducing the most general abstract syntax tree (AST) node from the line. A priori interest for a node n in the AST of the source file with root node r is defined as:

$$API(n) = BI(n) - \sqrt{w_{LOD} d(r,n)}$$

A priori interest is the base interest of the node, $BI(n)$, diminished with the factor w_{LOD} by the node's distance to the root, $d(r,n)$. Program lines containing one of the keywords

Base interest for an indenting program statement	30
Base interest for a package declaration	20
Base interest for a type declaration	20
Base interest for a method declaration	20
Base interest for a field declaration	10
Base interest for a variable declaration	10
Base interest for block closing "}"	2
Level of detail weight w_{LOD} of node in $API(n)$	2

Table 1. DOI function constants in the Fisheye interface.

package, class, interface, or method, are assigned a higher a priori interest than other lines. Enclosing statements—that is, those lines containing one of the Java keywords catch, if, finally, for, switch, try, or while—are also assigned a higher a priori interest. This is similar to Furnas's proposal. Table 1 lists the values of $BI(n)$ for different types of lines that are used to balance how lines are diminished in the context area. The constants were found through iterations of the design and evaluation with programmers. For efficiency, we process consecutive program lines as a block whenever possible. AST nodes that span multiple lines, and lines of other types than those mentioned above, for example comment lines, are processed as blocks.

A second component of the DOI function is based on the line's distance from the focus point. The distance is calculated as the sum of the syntactic distance and the semantic distance. The syntactic distance is calculated similar to Furnas's proposal; lines in the same indented block as the focus point are closer to the focus point than lines on other indentation levels and in different blocks, thus contributing to a higher degree of interest. In addition to syntactic distance, the Fisheye interface also calculates semantic distance from the focus point. Lines containing declarations of classes, methods and variables that are referenced in the focus point are deemed more relevant than other lines, including syntactically close lines, and are therefore assigned an even higher degree of interest. This type of line is highlighted with an alternate background color to express their semantic relation to the visible lines in the focus point. Thus, our design move beyond the ideas of Furnas by using semantic information in the second component of the DOI function.

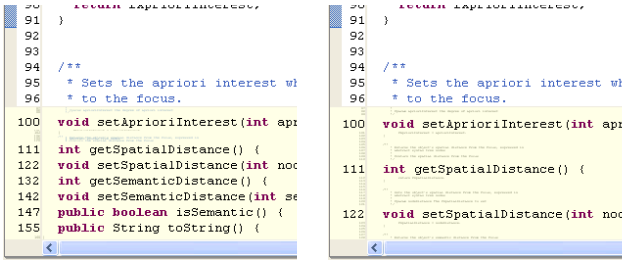
Magnification Function

A magnification function prioritizes each program line according to its degree of interest in order to reduce the size of the least interesting lines. A line's magnification is thus determined by its relevance relative to the amount of lines yet to be allocated space in the context area. Lines with similar degrees of interest are prioritized according to their distance in lines from the focus area, so that lines closest to the focus area are allocated space first. Figure 2 lists a simplified version of the algorithm used in the Fisheye interface.

We chose this strategy in the design of the Fisheye interface to solve the problem of deciding how to use the available display space, an issue that we discussed in the section on related work. An alternative implementation of Furnas's fisheye view is to use a magnification function that does not take

```
linesLeft = countLines(blocks);
foreach (block in prioritized blocks) {
    ratio = availableSpace / linesLeft;
    zoom = block.getDOI() * SQRT(ratio);
    block.setZoomLevel(ZOOM_FACTOR * zoom);
    linesLeft -= block.getLines();
    availableSpace -= block.getHeight();
}
```

Figure 2. Pseudo-code for calculating the magnification of lines in the context area.



(a) (b)

Figure 3. Fisheye view of 161 lines of source code; (a) the Fisheye interface and (b) with alternative magnification function that clips the source code to fit the view.

the amount of available space into consideration, and simply clips the view to the available display space. Figure 3 illustrates the difference between the two strategies. The fixed degree of magnification for the source lines in Figure 3(b) causes lines with a high degree of interest, that are far from the focus area, to be suppressed or clipped from the view. Similar approaches were used by Cockburn and Smith [3] and Hornbæk and Frøkjær [8]. In contrast, our prioritization strategy in Figure 3(a) first allocates space to the lines with high DOI to assure that they are included in the view.

User Interaction

The focus area offers the same facilities for interaction as 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 area 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 area retracts. By moving, or brushing, the mouse over lines in the context area, those lines are highlighted in the overview. Clicking on a line in the context area centers the focus area around that line and places the caret in the line.

EXPERIMENT

To gain a better understanding of the usability of fisheye views of source code, a controlled experiment was conducted in which the Fisheye interface was compared to the Linear interface. One goal of the experiment was to measure the usability of the interfaces for programming tasks, especially to seek evidence regarding the expectations about fisheye views raised by Furnas. Another goal was to describe how users interact with the two interfaces, so as to gain an understanding of how the design presented in the previous section affect user's navigation and understanding.

Participants

The 16 participants (2 female) were students at the authors' department (7) or professional programmers (9). Participants were screened to have at least one year of programming experience in an object-oriented language. Half of them had over five years of general programming experience. The participants were between 24 and 34 years old.

Tasks

Tasks addressed both navigation and program understanding. Navigation tasks from a study of source code elision [3] were used to evaluate the hypothesis that the fisheye view enables the programmer to navigate faster in the source code. We expected that it would be easier to find the information required to solve the task with the Fisheye interface, because there would be no need for scrolling the view. In cases where the information was not directly accessible without scrolling, we expected the user to navigate more quickly to the required information once it had been located.

To study whether the fisheye view affects program understanding, we also used composite task types that require more complex user interaction than the navigation tasks. These composite task types were based on issues in object-oriented programming, including delocalization, which have been discussed in the empirically based literature on programming (e.g., [4]). Finally, we used a type of task concerning the understanding of control structures in the source code, similar to tasks used in a study of control structure diagrams [7]. Below we describe the instances of these task types, which make up the 18 tasks used in the experiment.

One-step-navigation tasks

The first of two types of navigation task was of a form similar to: "In the method 'update', find the program line with the first call to the method 'Math.min'." The tasks of this type varied only with respect to the names of the methods and used source files from [3]. The tasks were repeated with source files of 186–187 lines and 368–376 lines.

Two-step-navigation tasks

The following is an example of the second type of navigation task used in the experiment: "In the method 'hasGreen', find the return type of the method that is called last." Only the method name in the task text were varied between tasks of this type. Like the one-step-navigation tasks, this type of task used source files from an earlier study [3], repeated with source files of 162–176 lines and 365–366 lines.

Determine-field-encapsulation tasks

One of the composite tasks involved determining whether or not two fields are encapsulated, that is, whether the variables are protected from external reference and corresponding get- and set-methods exist. The tasks were of the following form, varying only by the names of the fields: "How many of the fields 'fText' and fFont' are encapsulated correctly?" The source used in these tasks contained 340–361 lines and 34–38 methods—too many methods to be visible simultaneously in the Fisheye interface.

Task type	Linear	Fisheye
One-step-navigation	2	2
Two-step-navigation	2	2
Determine-field-encapsulation	1	1
Determine-delocalization	2	2
Determine-control-structure	2	2
Total	9	9

Table 2. Number of tasks performed by each participant.

Determine-delocalization tasks

Another challenging type of task involved determining delocalization in the source code, for example: "The method 'update' (line 207–214) contains a total of 6 method calls. How many of the methods called are declared in this file?" These tasks used source code files from the JHotDraw 5.2 program (http://www.jhotdraw.org/) with a number of method calls between five and nine, of which several were calls to methods declared in other files (delocalized code).

Determine-control-structure tasks

The last type of task concerned the control structure within a single method. An example of a task concerned with counting enclosing statements read: "In the method 'mergeTermInfos' (line 201–238), how many for, while and if/else statements enclose line 233?" An example of a task concerned with finding the closing brace of a block read: "In the method 'renameFile' (line 225–281), find the line containing the '}' that ends the if-block which starts on line 241." These tasks used source code files from two Apache Jakarta projects selected to contain methods with a body of more program lines than visible simultaneously in the Fisheye interface.

Materials

Participants used a laptop computer for the experiment with the screen set to a 1024 x 768 resolution with 16-bit color. The Eclipse window used all available screen space. For input, participants used the laptop's keyboard and an optical, wireless mouse. Tasks were presented in a task view in Eclipse next to the editor view. Participants typed their answer to the tasks in the task view and clicked a button to continue, enabling us to accurately register completion times.

Design

A within-subjects experimental design was used, the independent variables being interface type (Fisheye, Linear) and task type (One-step-navigation, Two-step-navigation, Determine-field-encapsulation, Determine-delocalization, Determine-control-structure). Participants performed a set of nine tasks with each interface, see Table 2. The order of tasks and interfaces were systematically varied and counter-balanced across participants.

Procedure

Prior to solving the 18 experimental tasks, participants were given an introduction lasting about 30 minutes. In the introduction, participants were explained how to operate the two interfaces, and were given a few minutes to try them. As part

of the introduction, participants also performed a set of nine warm-up tasks; five tasks using the Linear interface and four tasks using the Fisheye interface. Participants were allowed to ask questions during the warm-up tasks. Details of the tasks were explained and, if participants were hesitant, they were reminded how to operate the interfaces.

After the introduction, a set of nine experimental tasks were performed with each of the two interfaces. The participants were urged to give correct answers as quickly as possible, without asking questions during the experiment. A questionnaire about the interface just used was administered to the participants following each set of tasks. This questionnaire contained five questions from QUIS [2], and eight additional questions specific to the experiment (see Table 4). A third and final questionnaire was administered after all tasks had been completed, asking the participants for their age, gender and programming experience. The questionnaire also asked participants to compare the Fisheye interface with the Linear interface on a comparative scale. Additionally, participants were asked to write advantages and disadvantages of the Fisheye interface compared to the Linear interface. Finally, they were given the opportunity to verbally express their experiences with the two interfaces. The entire experiment lasted between 60 and 90 minutes for each participant.

RESULTS

The data collected comprised task completion times, accuracy, preference, and participants' satisfaction with the interfaces. Data were analyzed with repeated measures analysis of variance. Because the distribution of task completion times was positively skewed, the completion times were subjected to logarithmic transformation prior to analysis.

Accuracy

We find no significant difference between interface type in the accuracy of participants' answers to the tasks, $F(1,14) = .147, p = .707$. In total, 288 tasks were completed by the participants, of which 129 tasks were completed correctly with the Linear interface (89%) and 131 tasks completed correctly with the Fisheye interface (91%).

Task Completion Times

The task completion times are summarized in Table 3. The average task completion time is lower with the Fisheye interface compared to the Linear interface, $F(1,14) = 4.76, p = .047$. However, tasks and interfaces interact, $F(8,7) = 9.57, p = .004$, and we thus analyzed data per task to describe those task related differences.

Completion times show no significant difference between the interfaces in one-step-navigation tasks, $F(1,14) = 0.57, p = .463$. In two-step-navigation tasks, participants used significantly less time with the Fisheye interface compared with the Linear interface, $F(1,14) = 9.49, p = .008$, a difference of 18% in average completion time. We expected the fisheye view to generally improve navigation. However, the results suggest improvements only when navigating to methods that are visible and highlighted because they are being referenced in the focus area, which occurred in the second step of the two-step-navigation tasks.

Task type	Linear		Fisheye	
	M	*SD*	*M*	*SD*
One-step-navigation [a]	32.3	16.2	30.5	13.5
Two-step-navigation [a]	39.9	13.9	**33.8**	13.9
Determine-field-encapsulation [b]	80.7	24.7	96.8	37.6
Determine-delocalization [a]	92.1	46.9	**61.1**	34.1
Determine-control-structure [a]	43.9	17.1	50.5	20.7
Average	55.2	35.8	**49.8**	31.5

Table 3. Task completion times in seconds. Significantly lower times are shown in bold. ***(a) N=32, (b) N=16.***

Participants tended to complete determine-field-encapsulation tasks slower using the Fisheye interface compared with the Linear interface, but the difference in average completion time was not significant, $F(1,14) = 2.24, p = .157$. Though not significant, we did not expect to find inferior performance of Fisheye compared to the Linear interface.

In determine-delocalization tasks, participants counted how many of the methods or fields used in the body of a given method that were declared in the source file. On average, participants completed those tasks significantly faster (about 51%) using the Fisheye interface compared with the Linear interface, $F(1,14) = 13.9, p = .002$.

The determining-control-structure tasks involved counting the conditional and loop statements that enclosed a given program line, or finding the closing brace of a given loop control structure. Overall, we found no difference in completion time for these task types, $F(1,14) = 3.85, p = .070$. However, participants used more time to find the closing brace of a given loop control structure with the Fisheye interface compared to the Linear interface, $F(1,14) = 7.73, p = .015$. When implementing the Fisheye interface, we assigned a relatively low base interest to closing braces, Table 1. As a result, the closing braces to be found in these tasks were not visible in the context area. This may explain why participants used more time with the Fisheye interface, because they had to scroll the view to find the closing brace.

Satisfaction

Overall, participants preferred the Fisheye interface compared with the Linear interface ($t = -5.229, df = 14, p < .001$). Only one participant slightly preferred the Linear interface and one participant did not answer the question.

Average satisfaction scores for the two interfaces are summarized in Table 4 for the 14 questions that the participants answered. All questions were answered on a scale from one to seven. Across all questions, the participants rated the Fisheye interface better than the Linear interface, multivariate analysis of variance $F(1,15) = 10.0, p = .005$. Below we analyse each of the questions; all tests are made with individual analyses of variance tested against $F(1,15)$.

In general, participants liked the Fisheye interface better

than the Linear interface ($p < .006$). The Fisheye interface also scored better on the scale from terrible to wonderful ($p < .004$). There was no significant difference in how the participants found the two interfaces on the scale from hard to easy ($p > .9$). However, three participants mentioned as a disadvantage of the Fisheye interface that it required more training to use effectively. Participants found the Fisheye interface both more pleasant ($p < .03$) and more fun ($p < .001$) to use than the Linear interface.

On the scale from confusing to clear, the participants found the Fisheye interface to be significantly less clear than the Linear interface ($p < .04$); the only question where the Fisheye interface scores lower than the Linear interface. Five participants commented as a disadvantage of the Fisheye interface that it could be confusing to use, in particular with scrolling. Also, some participants did not clearly understand that program lines were shown and highlighted because they were related to one or more lines in the focus area. We found no significant difference between the two interfaces in the participants' answers of whether they often lost their orientation in the source code ($p > .05$), nor was there a difference in the answer to whether it was clear to them where in the source code they were looking ($p > .25$). These results suggest that the Fisheye interface was not confusing in general, but rather that it was confusing when searching by scrolling in the source code.

Satisfaction question	Linear	Fisheye
1. How did you find the interface in general?		
Very poor - Very good	4.13 (.34)	**5.44** (.20)
2.-6. How was the interface to use?		
Terrible - Wonderful	4.00 (.29)	**5.13** (.15)
Hard - Easy	5.19 (.37)	5.13 (.31)
Frustrating - Pleasant	3.81 (.41)	**5.00** (.29)
Boring - Fun	3.56 (.29)	**5.25** (.35)
Confusing - Clear	**5.81** (.31)	4.50 (.37)
7. It was clear most of the time where I was in the source code.		
I disagree - I agree	5.88 (.31)	5.25 (.36)
8. I often lost my orientation in the source code.		
I disagree - I agree	2.88 (.43)	2.56 (.26)
9. How do you perceive the tasks?		
Very challenging - Very easy	5.31 (.27)	5.56 (.24)
10. How were your answers to the tasks?		
Very poor - Very good	5.56 (.26)	5.75 (.27)
11.-12. Was the source code...		
Hard to understand - Easy to understand	4.81 (.31)	5.19 (.23)
Hard to overview - Easy to overview	4.44 (.38)	4.94 (.28)
13. Were methods you were trying to locate in the source code...		
Hard to locate - Easy to locate	3.50 (.39)	**5.31** (.35)
14. Were other information in the source code...		
Hard to locate - Easy to locate	3.50 (.35)	**5.60** (.22)

Table 4. Average satisfaction scores (and standard error of the mean) for the 14 satisfaction questions for the two interfaces. The anchor points on a semantic differential scale is shown below each question. Significantly better scores are shown in bold.

Participants found it easier to find methods ($p < .004$) and other information ($p < .001$) in the source code with the Fisheye interface than with the Linear interface. Also, most participants commented in the questionnaire that they felt the Fisheye interface gave a better overview of the source code and helped to locate methods and variables. About half of the participants commented as an advantage that they could see enclosing statements in the Fisheye interface.

The fisheye view's poor performance in determine-field-encapsulation tasks may be explained by comments made by some participants. They found it difficult to search for variables and methods in the context area while scrolling, because the context area was displaying lines which are semantically related to the lines in the focus area. As lines scroll in and out of focus, different semantic relationships in the source code take effect, resulting in irregular changes to the context area.

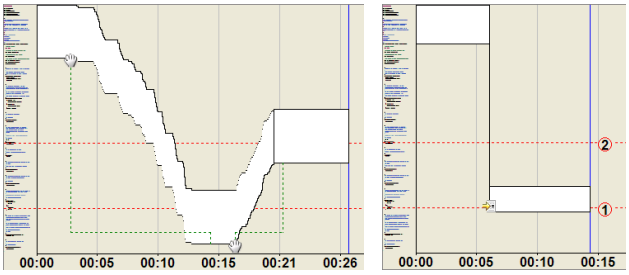
The focus area in the Fisheye interface was too small according to comments made by 12 out of the 16 participants. A few participants added that they would find this a problem when writing or editing the code.

Interaction with the Interfaces

Data describing the participants' interaction with the interfaces were automatically collected during the experiment. We visualized this interaction with progression maps, which have previously been used to analyze reading of electronic documents [8]. Analysis of the progression maps revealed patterns in the participants' interaction which, in many of the tasks, are clearly distinguishable between the two interfaces. The patterns evident in the progression maps support the conclusions based on the task completion times, but also indicate some problems with the Fisheye interface. We show representative patterns and provide counts of participants who interact in a similar way.

The progression maps are used to show which part of the source file was visible in the focus area at a given time during the task (see Figure 4 to Figure 8). Dashed horizontal lines ending in a circled number to the right of the map indicate program lines that hold the answer to the task. In progression maps for tasks where more than one program line is used to answer the task, the numbers indicate the order in which the lines are to be used. Certain forms of interaction are annotated with symbols in the progression maps: a hand symbol when user scrolled the view by dragging the scrollbar thumb and an arrow-in-document symbol when user clicked in the context area. Other interaction forms are directly discernable from the map, such as scrolling by arrow keys and page up/down keys respectively.

Typical patterns found in progression maps for two of the two-step-navigation tasks, see Figure 4, show that with the Linear interface, participants had to search through the file for both methods. With the Fisheye interface, 11 out of



(a) Linear interface (b) Fisheye interface

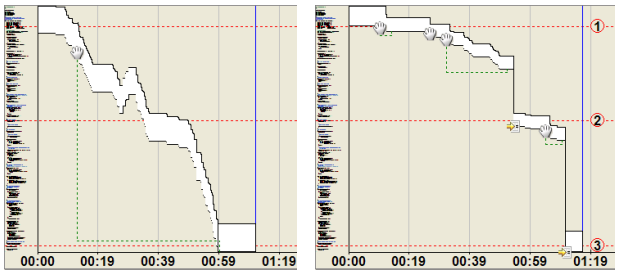
Figure 4. Progression maps, two-step-navigation tasks.

16 participants were able to find the return type in the second method directly in the context area. Similar differences are evident in progression maps for the one-step-navigation tasks.

Progression maps representative for determine-field-encapsulation tasks are shown in Figure 5. The patterns indicate that while participants found places of interest and jumped by clicking in the context area in the Fisheye interface, they also needed to scroll to search the 34–38 methods. Analysis of the progression maps does not yield any explanation why participants solved this type of task slower with the Fisheye interface, as the task completion time results suggest. One possible cause is that participants searched more slowly by scrolling in the Fisheye interface than in the Linear interface.

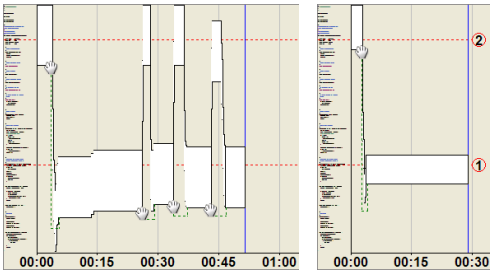
Typical interaction patterns can be seen in the representative progression maps for determine-delocalization tasks in Figure 6 (involving variables) and Figure 7 (involving methods). The progression maps confirm that participants made several searches and jumps in the source code with the Linear interface. Being asked to determine how many of the called methods were defined in the source file, they had to search for the definition of each method, returning each time to find the name of the method called next, start searching again, and so forth. The progression maps for the Fisheye interface show that once participants had navigated to the method, they were able to use the fisheye view's context area to find the information necessary to complete the tasks. In the Fisheye interface, 12 out of the 16 participants completed the tasks with minimal interaction.

Figure 8 shows the progression maps for the two determine-control-structure tasks that involved counting program state-



(a) Linear interface (b) Fisheye interface

Figure 5. Progression maps, determine-field-encapsulation tasks.



(a) Linear (b) Fisheye

Figure 6. Progression maps for determine-delocalization tasks involving variables.

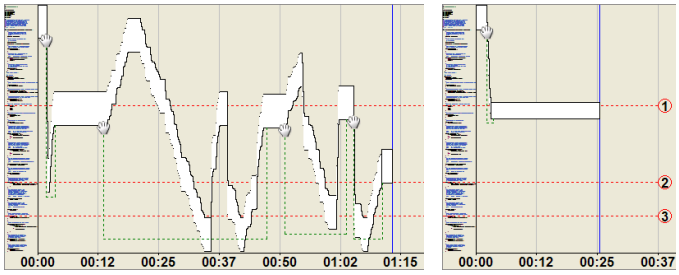
ments enclosing a given line. In the first task, all participants using the Linear interface scrolled down to the specified line, and were then able to answer the task without scrolling further, Figure 8(a). Six out of eight participants using the Fisheye interface continued to scroll the focus area to determine the control structure and answer the task, Figure 8(b). The Fisheye interface thus makes the task of finding the enclosing statements harder for participants. The second task, Figure 8(c) and 8(d), shows a different result. All participants using the Linear interface, once they had found the specified line, had to scroll back at least once to determine the control structure. Seven of eight participants using the Fisheye interface, however, could determining the control structure using the context area without scrolling any further. These interaction patterns confirm our hypothesis that the Fisheye interface helps to determine large control structures faster.

For determine-control-structure tasks where participants had to find the closing brace of a loop-structure, the progression maps did not show any apparent differences in how participants interacted with the Fisheye interface compared to the Linear interface. The inferior performance with the Fisheye interface in these tasks, with respect to task completion times, could be caused by the smaller focus area.

DISCUSSION

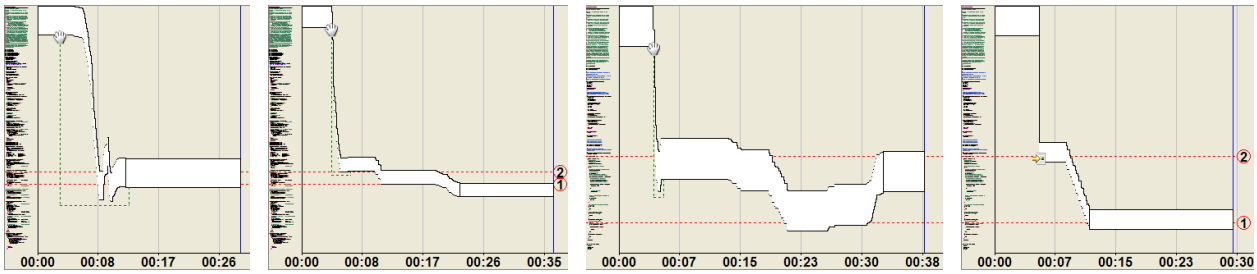
The results from our experiment show an overall improvement in task completion times with the fisheye interface for representative program navigation and understanding tasks. Yet, strong differences in task completion times were found among tasks. Participants were equally accurate in answering the tasks. They much preferred the Fisheye interface and scored it significantly higher on 6 of 14 satisfaction questions, for example concerning whether the interface was easier or more pleasant to use. By analyzing progression maps, we identified great variation in how the participants interacted with the Fisheye interface. In spite of the short time the participants used the interface, several of them displayed very effective use of the fisheye view. The context area was frequently used for searching and navigating in the source code. Many tasks were completed with sparse interaction resulting in reduced physical effort compared to the interaction with the Linear interface.

To discuss our design and empirical results, we return to the questions raised in the section on related work. The first question concerned how to use the display space in a fisheye



(a) Linear (b) Fisheye

Figure 7. Progression maps, determine-delocalization tasks involving methods.



(a) Linear, task 1 (b) Fisheye, task 1 (c) Linear, task 2 (d) Fisheye, task 2

Figure 8. Progression maps for determine-control-structure tasks concerned with counting enclosing statements. The line numbered 1 indicates the program line given in the task, line 2 the farthest line needed to answer the task.

view. Many fisheye views of text [1, 3, 8] show mostly diminished text in the context area. We propose to have mainly readable text displayed in the context area. This allows direct use of the information in the context view, which is evident in the tasks where users directly read source lines displayed in the context area or when they click in the context area to jump to a certain line. Further experimental work is needed to understand the difference between these two approaches to displaying content in the context views. Alternative approaches should be considered; for example, dispensing with a static context area and displaying context information in proximity to the focus point would make it clear to the user why that context information is displayed.

A second question concerned how to establish the user's focus point in the source code. Our solution to use a focus area spanning many lines as the focus point gives the interface stability, because the context view rarely needs updating. Some participants, however, were confused about what semantic relation that caused program lines to be shown and highlighted in the context area. How to make transparent to users why lines are shown in the context view is not easy. One solution could be to allow the user to control the focus point more accurately, for example by the position of the caret. This would allow for an easily understandable relation between focus point and context information, but would also make the interface visually busy.

We succeeded in using richer information in establishing the degree-of-interest, a challenge also raised in the section on related work. Our data show that the Fisheye interface helped participants to find and navigate to a method, if the method is semantically related to the focus area. Participants also spent less time using the Fisheye interface to determine which methods are called in the focus area. The significant effect of showing lines in the context area that are related to the focus area may have been influenced by those lines being highlighted. Nevertheless, we argue that fisheye views in source code editors should include program lines which are referenced by the lines in the programmer's focus. In contrast, the results of our experiment leads us to believe that the Fisheye interface is less useful for displaying lines containing declarations of methods and variables, which are not directly related to the programmer's point of focus. In common programming environments, such lines are typically displayed in outlines of the edited source file. Considering the tradeoff between showing those lines in the context area compared to having a larger focus area for editing source code, we think that the base interest assigned to such lines as method headers in the Fisheye interface seems too high (see Table 1). Future work could examine the relative utility of the various kinds of information that could be shown in the context area, but also alternative ways of creating the degree of interest function; automatically, for example, by using eye tracking or logging of participants' navigation.

The fourth question raised in the section on related work concerned how to integrate fisheye views in a modern development environment. Our plug-in works with Eclipse, but some issues remain. In particular, our implementation of how the fisheye view changes when scrolling is still unsatisfactory: the information needed when scanning during scrolling seems much different from that needed while reading and editing source code.

At least three problems and limitations of the experiment should be considered when interpreting the results. First,

participants were given relatively short time to practice with the interfaces before the experiment. Informal observations made during the experiment suggest that participants sometimes hesitated or expressed doubts, leading us to suspect that they were given insufficient time to become confident in using the Fisheye interface. Second, the realism of the programming environment was reduced because we limited the tools available to the participants during the experiment. Modern source code editors often offer advanced features, such as hyperlinking and advanced highlighting. Also, many tools are usually available in addition to the editor, such as the outline view mentioned earlier, which may affect how programmers use the editor. Our results do therefore not necessarily reflect the effect of the Fisheye interface in practice. Third, simple programming activity was investigated in this paper. In particular, we investigated only navigation and program understanding of static programs, not of programs that are created or modified by the user. We still face the challenge of uncovering what long term effects fisheye views in source code editors may have on programming.

CONCLUSION

We have presented a design and empirical evaluation of a fisheye view applied to source code. The aim has been to support programmers in navigating and understanding source code by displaying those parts of the source code that have the highest degree of interest given the programmer's current focus. In designing the interface, we have prioritized to retain a static division between the focus and the context areas of the fisheye view, and to saturate the context area with readable information. Further, we have introduced semantic relations between parts of the source code in the calculation of the degree of interest. The interface is fully integrated with the Eclipse development environment.

In an experiment, we compared the usability of an interface using the fisheye view with an interface using a linear view of the source code. Sixteen participants performed nine tasks with each of the two interfaces. Overall, the participants performed the tasks significantly faster with the fisheye view, although an effect of task type was present. The participants generally preferred the interface with the fisheye view. The experiment illustrates how participants interacted with the fisheye view, thereby identifying information in the context area that was useful to participants. Semantically related information seems important, while source code displayed because of a high a priori degree of interest was less useful.

In summary, fisheye views seem promising for displaying source code. Our study suggests, however, that further work should attempt to improve performance across all tasks, and that the degree of interest function may be further refined.

ACKNOWLEDGEMENTS

We thank Tue Haste Andersen and Morten Hertzum for helpful comments on a draft of this paper. For providing us with task material used in his study, we thank Andy Cockburn. We would like to thank the persons that helped by participating in the experiment. Finally, we thank the CHI reviewers for constructive comments.

REFERENCES

1. P. Baudisch, B. Lee, and L. Hanna. Fishnet, a fisheye web browser with search term popouts: a comparative evaluation with overview and linear view. In *Proc. AVI 2004*, 133–140. ACM Press, 2004.
2. J. P. Chin, A. Virginia, and K. L. Norman. Development of an instrument measuring user satisfaction of the human-computer interface. In *Proc. CHI '88*, 213–218. ACM Press, 1988.
3. A. Cockburn and M. Smith. Hidden messages: evaluating the efficiency of code elision in program navigation. *Interacting with Comp.*, 15:387–407, 2003.
4. A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *Proc. 22nd Int. Conf. on Software Eng.*, 467–476. ACM Press, 2000.
5. G. W. Furnas. The fisheye view: A new look at structured files. In S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors, *Readings In Information Visualization: Using Vision To Think*, 312–330. Morgan-Kaufmann, 1999. Originally published as Bell Laboratories Technical Memorandum #81-11221-9, October 12, 1981.
6. C. Gutwin. Improving focus targeting in interactive fisheye views. In *Proc. CHI 2002*, 267–274. ACM Press, 2002.
7. D. Hendrix, J. H. Cross II, and S. Maghsoodloo. The effectiveness of control structure diagrams in source code comprehension activities. *IEEE Trans. Software Eng.*, 28:463–477, 2002.
8. K. Hornbæk and E. Frøkjær. Reading patterns and usability in visualizations of electronic documents. *ACM Trans. Computer-Human Interaction*, 10(2):119–149, 2003.
9. H. Koike. Fractal views: a fractal-based method for controlling information display. *ACM Trans. Information Systems*, 13(3):305–323, 1995.
10. J. I. Maletic, A. Marcus, and M. Collard. A task oriented view of software visualization. In *Proc. VISSOFT '02*, 32–42, 2002.
11. A. von Mayrhauser and A. M. Vans. From program comprehension to tool requirements for an industrial environment. In *Proc. 2nd Workshop on Program Comprehension*, 78–86, 1993.
12. A. von Mayrhauser and A. M. Vans. Program understanding behavior during debugging of large scale software. In *Proc. 7th Workshop on Empirical Studies of Programmers*, 157–179. ACM Press, 1997.
13. L. B. Páez, J. B. da Silva-Fh., and G. Marchionini. Disorientation in electronic environments: A study of hypertext and continuous zooming interfaces. In *Proc. 59th Annual Meeting of the American Society for Information Science*, 58–66, 1996.
14. B. A. Price, I. S. Small, and R. M. Baecker. A taxonomy of software visualization. In *Proc. 25th Hawaii Int. Conf. on System Sciences*, 597–606, 1992.
15. D. Schaffer, Z. Zuo, S. Greenberg, L. Bartram, J. Dill, S. Dubs, M. Roseman. Navigating hierarchically clustered networks through fisheye and full-zoom methods. *ACM Trans. CHI*, 3(2):162–188, 1996.
16. A. Skopik, C. Gutwin. Improving revisitation in fisheye views with visit wear. *Proc. CHI 2005*, 771–780, 2005.
17. M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *J. Software Systems*, 44:171–185, 1999.
18. M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Sci. Comput. Program.*, 36(2-3):183–207, 2000.
19. O. Turetken, D. Schuff, R. Sharda, and T. T. Ow. Supporting systems analysis and design through fisheye views. *Commun. ACM*, 47(9):72–77, 2004.
20. M. Weiser. Programmers use slices when debugging. In *Commun. ACM*, 446–452, 1982.
21. M. Weiser and J. Lyle. Experiments on slicing-based debugging aids. *Proc. 1st Workshop on Empirical Studies of Programmers*, 187–197, 1986.