

Transient or permanent fisheye views: A comparative evaluation of source code interfaces

Information Visualization
11(2) 151–167
© The Author(s) 2011
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1473871611405643
ivi.sagepub.com



Mikkel Rønne Jakobsen¹ and Kasper Hornbæk¹

Abstract

Transient use of information visualization may support specific tasks without permanently changing the user interface. Transient visualizations provide immediate and transient use of information visualization close to and in the context of the user's focus of attention. Little is known, however, about the benefits and limitations of transient visualizations. We describe an experiment that compares the usability of a fisheye view that participants could call up temporarily, a permanent fisheye view, and a linear view: all interfaces gave access to source code in the editor of a widespread programming environment. Fourteen participants performed varied tasks involving navigation and understanding of source code. Participants used the three interfaces for between four and six hours in all. Time and accuracy measures were inconclusive, but subjective data showed a preference for the permanent fisheye view. We analyse interaction data to compare how participants used the interfaces and to understand why the transient interface was not preferred. We conclude by discussing seamless integration of fisheye views in existing user interfaces and future work on transient visualizations.

Keywords

information visualization, fisheye view, transient visualizations, user study, programming

Introduction

A fundamental challenge in information visualization is to map data to visual structures and to transform those visual structures into views suitable for users' tasks.¹ Seesoft, for example, maps lines in a source code file to line marks on a vertical axis, aimed at helping users understand changes to the code.² Document lens uses a focus + context transformation to allow users to inspect a particular part of a document while being able to see the entire document to stay in context.³

The user's control of the visual structures and view transformations is central to visualization.¹ Often visualizations are designed to support a specific task and make fixed mappings and transformations that are effective in that task. In contrast, applications often support a variety of tasks in complex work settings. Integrating a visualization aimed at supporting a specific task in an existing application results in permanent changes to the user interface. Thus, it seems there is a gap in our understanding of how users can control a visualization to switch between visual structures or view transformations, which makes it

difficult to integrate visualizations in established user interfaces.

One alternative would be to use information visualization without permanently changing the user interface. Transient visualizations aim to do that by providing immediate and transient use of information visualization close to, and in the context of, the user's focus of attention.⁴ By using transient visualizations to support specific tasks and contexts of use, the permanent view can be dedicated to information used across tasks and contexts of use. However, empirical data on the relative benefits of transient and permanent interfaces are lacking.

This article studies fisheye views of source code – a visualization that has been shown to help programmers

¹Department of Computer Science, University of Copenhagen, Njalsgade 128, Building 24, 5th Floor, DK-2300 Copenhagen, Denmark

Corresponding author:

Mikkel Rønne Jakobsen, Department of Computer Science, University of Copenhagen, Njalsgade 128, Building 24, 5th Floor, DK-2300 Copenhagen, Denmark
Email: mikkeltjr@diku.dk

in navigating and understanding source code.⁵ The fish-eye view as originally proposed by Furnas⁶ balances in a single view ‘the need for local detail against the need for global context.’ A fisheye view of source code does so by displaying only those parts of the code with the highest degree of interest (DOI) given the user’s current focus. However, information shown because it has a high DOI may not be equally important in all tasks. In some tasks, like, for instance, reading or editing source code, a fisheye view may even be unfavorable compared to a large ‘local detail’ view of source code. One solution is to allow users to call up a fisheye view on demand. A transient fisheye view of source code that can be temporarily called up may support navigation and understanding while still providing a large view of code for reading and editing.

We describe an experiment designed to gain insight into the benefits and limitations of permanent and transient versions of a fisheye view. Compared to an earlier paper on transient visualization,⁴ we present richer experimental data from a much more complex domain. This article starts by relating this study to previous work on transient use of visualization and lightweight interaction (‘Related work’ section). Next, the interfaces that are investigated in this article are described (‘Transient and permanent fisheye views’ section), as is the experiment designed to investigate their benefits and limitations (‘Experiment’ section). Based on the results from the experiment (‘Results’ section), we discuss how to advance research in information visualization and how to support work in complex domains with fisheye interfaces and other information visualizations (‘Discussion’ section).

Related work

The idea of transient visualizations was introduced in Jakobsen and Hornbæk.⁴ According to Jakobsen and Hornbæk,⁴ transient visualizations are immediate (bring the user into direct and instant involvement with the information representation), transient (information is only displayed temporarily, and is easily dismissed), close to the users’ focus (the information is shown close to the region of focus in the display), and contextual (the information is related to the user’s current focus of attention). Other researchers have supported this idea. For instance, based on the observation that a particular design of a permanent visualization may be suitable only in some scenarios, Baudisch et al.⁷ recommended that users should be allowed to bring up different visualizations on demand depending on their particular needs.

Earlier work has applied related ideas of transient representations of information and lightweight interaction. For instance, Excentric Labeling provides

labels for a neighborhood of objects located around the cursor.⁸ By showing labels temporarily when the cursor stays over an area for more than a second, the technique avoids information clutter and the need for extensive navigation. Side Views uses transient views to provide dynamic previews of multiple commands by visualizing the outcome of commands on the current selection, for instance using bold, italic, or underline on selected text.⁹ Zellweger et al.¹⁰ studied the impact of lightweight, animated glosses for link anchors on hypertext browsing. Altogether, these transient preview techniques help users to probe relevant information without navigation and display switching, and to assess possible actions without resorting to ‘trial-and-undo.’

Context menus that pop up near the mouse cursor or text caret present commands related to the current focus (e.g., for changing the font of selected text). Hotbox extends context menus with multiple menu bars close to the cursor and with access to additional menus via mouse gestures.¹¹ See-through tools are another technique that provides close and contextual access to commands without requiring permanent use of display space.¹²

Many information visualizations use brushing to highlight (or otherwise affect) instances in other views of an object that the user brushes over.¹³ Highlighting techniques have been adopted, for example, in the Eclipse Java source code editor, where the caret can be placed in a variable to highlight all references in the code to that variable. Similar ideas have been demonstrated in spreadsheets.¹⁴ These techniques provide immediate and non-intrusive visualizations through lightweight interaction.

Novel interaction techniques have been generated to temporarily bring objects that are otherwise hard to interact with closer to the user. The interaction technique for large displays called Vacuum helps reach remote objects through proxies that are transiently placed close to the cursor for easy manipulation, reducing the physical demands on the user.¹⁵ Similar challenges in small displays have brought about techniques to visualize and navigate to off-screen targets with halos and proxies.¹⁶

Despite the motivations for transient visualizations mentioned above, the use of transient visualizations has, to our knowledge, only been empirically investigated in an experimental study of overview + detail map interfaces.⁴ That study showed how participants preferred a transient overview, which appeared temporarily close to the mouse cursor, compared to a fixed overview, which was shown permanently in the corner of the display. Thus, we proceed to experimentally compare interfaces in the much more complex domain of programming and in a much longer term

experiment than that reported in Jakobsen and Hornbæk.⁴

Transient and permanent fisheye views

To investigate how transient visualizations can be used in complex work settings such as programming, we implemented a transient fisheye view of source code in Eclipse, a widespread development environment (Figure 1). The fisheye view divides the window of the Java editor into a focus area and a context view. 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 transient fisheye view is compared to a permanent fisheye view (using the same method for

producing the fisheye view as in the transient view) and to a baseline linear view.

Fisheye view of source code

Before we present the experiment, we describe the fisheye views of source code used in the experiment.

Degree of interest. A DOI is determined for each line in the source code 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:

$$\text{DOI}(x|p) = \text{enclosing}(x, p) + \text{occurrences}(x) - d_{\text{line}}(x, p)$$

First, lines are interesting if they contain declarations or statements that structurally enclose the code that is visible in the focus area. Such lines contain a

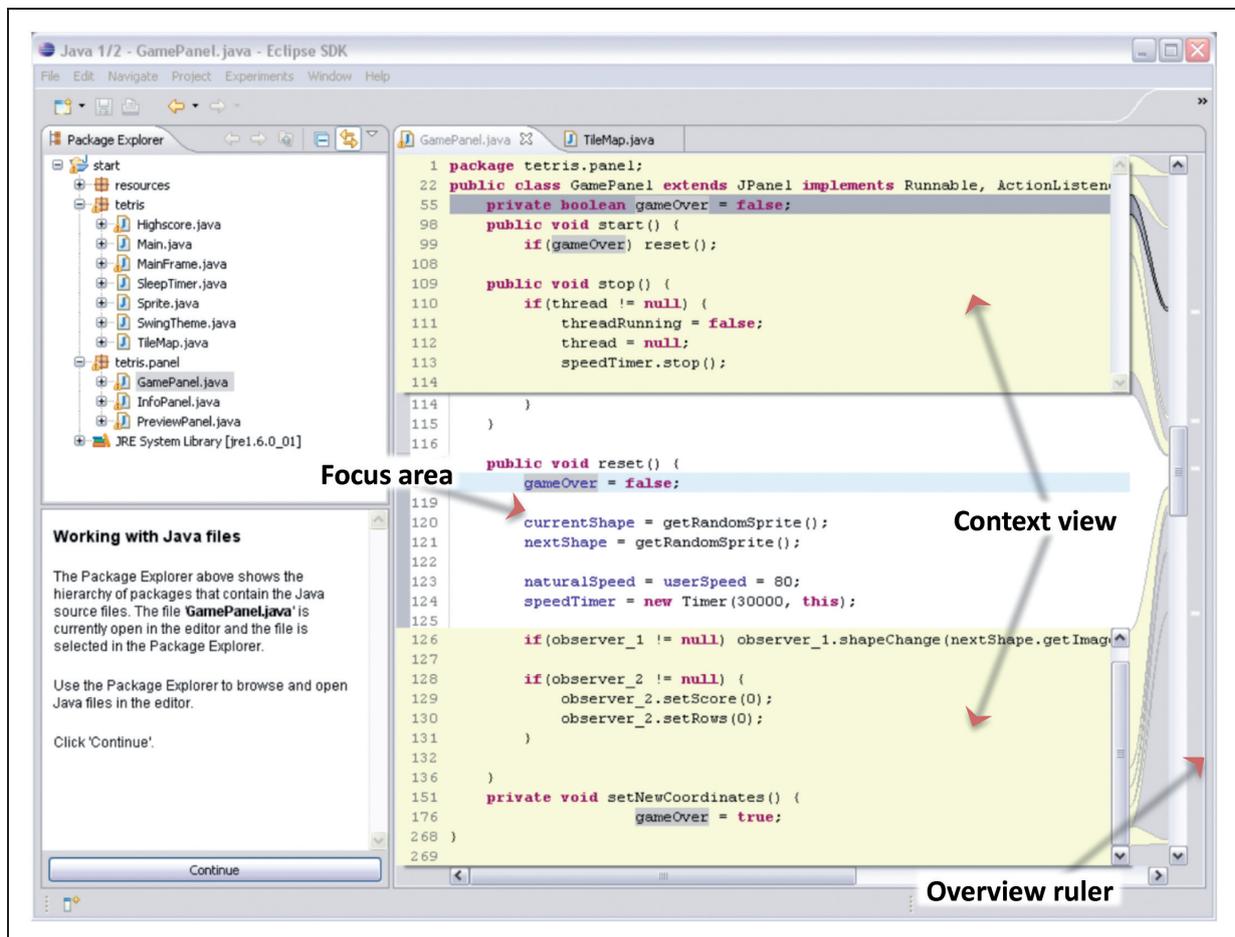


Figure 1. Transient fisheye view called up in Eclipse to divide the Java editor window into a focus area and a context view. Lines containing occurrences of a selected variable are shown in the context view and in the overview ruler to the right of the scrollbar (as white rectangles).

package, class, interface or method declaration, 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 that are semantically related to the code in focus may be interesting to the user. The Java editor in Eclipse allows programmers to highlight occurrences of a variable, method, or type to better see where it 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. Lines containing such highlighted occurrences of a selected element are interesting. Further, lines that contain declarations of methods that enclose these occurrences are also of interest since they provide context for the occurrences. Thus, $occurrences(x) = k$ adds to the DOI of line x that contains an annotation or declares a method that encloses an annotation.

Third, a distance $d_{line}(x, p) \in [0; k]$ proportional to the number of program lines from line x to focus area p detracts from that line's DOI.

Similar to the DOI-function used in Jakobsen and Hornbæk,⁵ this DOI-function is composed of both syntactic distance (the *enclosing* component) and semantic distance (the *occurrences* component). Also, syntactic distance is defined in terms of the abstract syntax tree (AST) of the source code file, like in Jakobsen and Hornbæk,⁵ but in a simpler way: $enclosing(x, p)$ is a binary function that returns k only if the node x is on the path from the focus node p to the root of the AST, otherwise $enclosing(x, p)$ returns 0. The rationale for this simplification of the DOI-function is to make it easier to understand what lines are shown in the context view.

Source code elision in the context view. Lines are included in the context view if they have a DOI 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 $DOI \leq k$ are added to the context view in descending order of DOI. This includes lines that are 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. Similarly, in code that is heavily indented, many lines may have a high DOI because they contain declarations or statements that structurally enclose the code in the focus area. However, 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 lines that are important, the windows containing the upper and lower context views can be scrolled. The context view automatically scrolls to show lines closest to the focus area (i.e., those lines with the highest DOI) when its contents change.

To sum up, we compare the fisheye interface to that of Jakobsen and Hornbæk.⁵ The present fisheye interface shows program lines at a fixed readable font size and guarantees that a line x is included in the context view if it structurally encloses the code that is visible in the focus area ($enclosing(x, p) = k$) or if it is semantically related to the focus by containing an occurrence of a selected element ($occurrences(x, p) = k$). In contrast, the fisheye interface studied in Jakobsen and Hornbæk⁵ reduces the size of the least interesting lines, whereby some lines that structurally enclose the code in the focus area or are semantically related to the focus can become unreadable.

Interfaces

Three interfaces to a Java editor were used in the experiment (Figure 2). The three interfaces all contain syntax highlighting, line numbers, and the highlighting of occurrences, which was described above. The interfaces also include an overview ruler next to the editor's scrollbar, in which highlighted occurrences are shown as white rectangles. Clicking on a white rectangle jumps to the line containing the occurrence and places the caret at that line. The part of the document shown in the editor window is visually connected with its position in the overview ruler by curved lines. All features except those described above are disabled in the Java editor. Below we describe each of the three interfaces in turn.

The *Permanent interface* contains a fisheye view of source code. The editor window is permanently divided into a focus area and a context view – permanently transforming the view of the visual structure of information is the typical implementation of fisheye and focus + context interfaces. 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 area uses one-third of the display space in the editor window. However, the context view automatically reduces in size near the top and bottom of the document. 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. Clicking on a line in the context view jumps to that line and places the caret at the line.

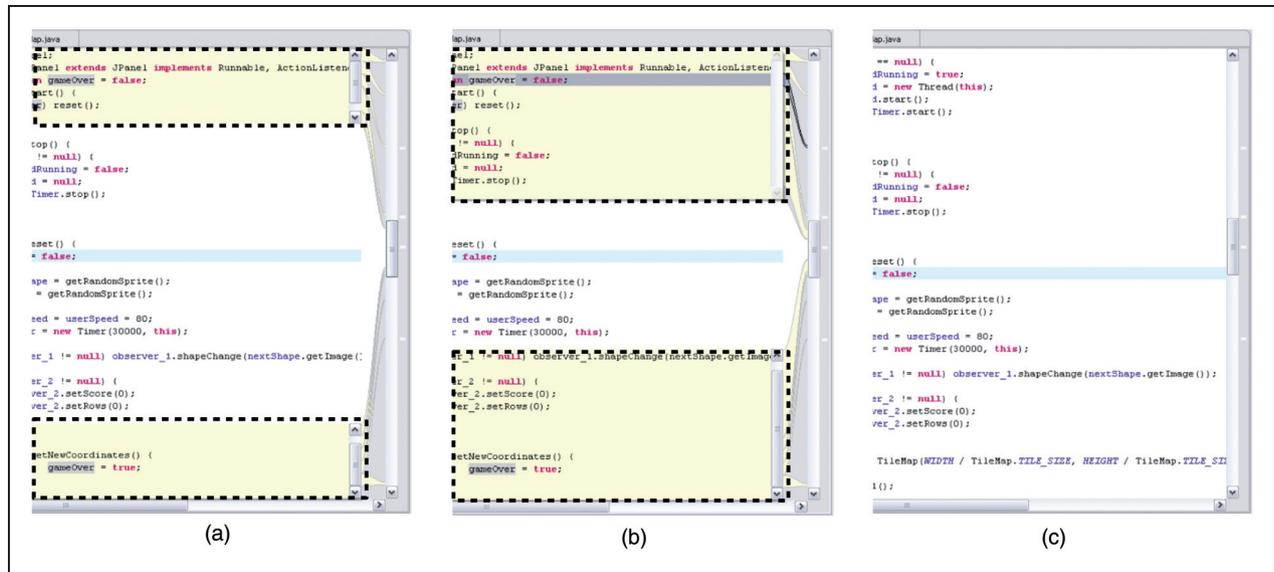


Figure 2. The three Java editor interfaces used in the experiment: (a) Permanent interface in which the context view is permanently shown, (b) Transient interface in which the context view has been temporarily called up – otherwise it looks like the Baseline interface – and (c) Baseline interface that shows a linear representation of code. Dashed rectangles are added to emphasize the difference between the context views used in the interfaces.

The *Transient interface* contains a linear view of source code, but allows the user to call up a transient fisheye view. The user calls up the context view with a keyboard shortcut. The context view remains visible until the user either hits Esc, clicks outside the context view, or clicks on a line in the context view. Clicking on a line in the context view jumps to that line and places the caret at the line. Alternatively, the user can use the arrow keys to select a line in the context view and jump to that line by hitting return.

One general characteristic of transient visualizations is that they involve no permanent use of display space, because information is only shown temporarily and is easily dismissed.⁴ When the user calls up the context view in the Transient interface, we hypothesize that information in the focus area is less important because the user shifts their attention to the context view. We therefore think that it is useful to show a larger context view in the Transient interface that uses more display space than in the Permanent interface, so as to allow more lines to be visible simultaneously in the context view of the Transient interface than the Permanent interface. The ratio of focus area size to context view size is therefore 2 to 3 in the Transient interface, whereas the ratio is 3 to 2 in the Permanent interface (compare Figure 2(a) and (b)). While this confounds context view size with transience, we think it is the best implementation of the Transient interface.

The *Baseline interface* contains a linear view of source code similar to the normal Java editor in Eclipse.

Experiment

The main aim of the experiment is to compare a transient visualization to a permanent visualization in a more complex domain than has previously been researched.⁴ A secondary aim of the experiment is to extend on findings from an earlier evaluation of a fisheye view of source code.⁵ The present experiment replicates the study by Jakobsen and Hornbæk⁵ in that a fisheye interface is compared with a baseline interface using similar types of task. However, compared to the earlier study of Jakobsen and Hornbæk,⁵ this study investigates longer term use of the interfaces. Also, the fisheye interface used in this study implemented changes addressing some of the issues discussed in the earlier study.

Participants

The 14 participants (one female) were graduate students in computer science enrolled at the authors' department. They were between 24 and 44 years of age ($M = 30.1$). Participants were given course credit as an incentive for participating in the experiment.

Participants were asked how long they had spent programming in general or in an object-oriented language (less than 1 year, 1–3 years, 3–5 years, or more than 5 years). The programming experience distribution of the participants is shown in Figure 3(a). All had at least 1 year of experience with programming in an object-oriented language and all but two participants

had experience with Java. Participants were also asked whether they had used Eclipse or programmed in Java before, and if so, how long ago they had last done so, see Figure 3(b). Half of the participants had used Eclipse before, but only one had used Eclipse within the last month.

Tasks

Two sets of tasks were used in the experiment, both involving navigation and understanding of source code. *Program investigation tasks* involve varied navigation and understanding activity as part of investigating the source code of a program. Program investigation tasks are of relatively high complexity in that they vary in the degree of structure, concreteness of the answer, number of paths to the answer, and the amount of information needed to answer the task. *Controlled tasks* included five specific types of navigation and understanding task. Controlled tasks are of relatively low complexity in that they are well structured, have a single path to a single precise answer, and limited information is needed to answer the task.

Program investigation tasks were included to see how participants used the interfaces during varied program investigation activity that includes reading code and switching between different files. Because these tasks are ambiguous, containing several paths to an answer, they aim for variation in participants'

approaches to seeking the information they need to answer the tasks. Therefore, they allowed us to compare how participants performed tasks with the interfaces only at a high level. In contrast, controlled tasks focus specifically on navigation and understanding, that is, programming activity for which fisheye interfaces may be particularly useful. Because these tasks focus on specific aspects of source code navigation and understanding in obtaining a single answer, they allowed us to compare in detail how participants interacted with the interfaces to provide the answer. Below we describe each set of tasks in detail.

Program investigation tasks. The program investigation tasks required participants to investigate the source code of an open source graphics program. Participants could browse all files comprising the source code of the program, but since we focus on the interaction with the editor, we provided names of particular source files in the tasks as a starting point. Participants were not able to run the programs in the experiment, but were handed a screenshot of the main window of the program to provide context for the tasks.

These tasks used source code from three open source programs: 11 tasks used TinyUML (tinyuml-0.13_02-src.zip downloaded from <http://sourceforge.net/projects/tinyuml/> contained 18 K-LOC), 11 tasks used JDraw (jdraw_v11.5.src.zip downloaded

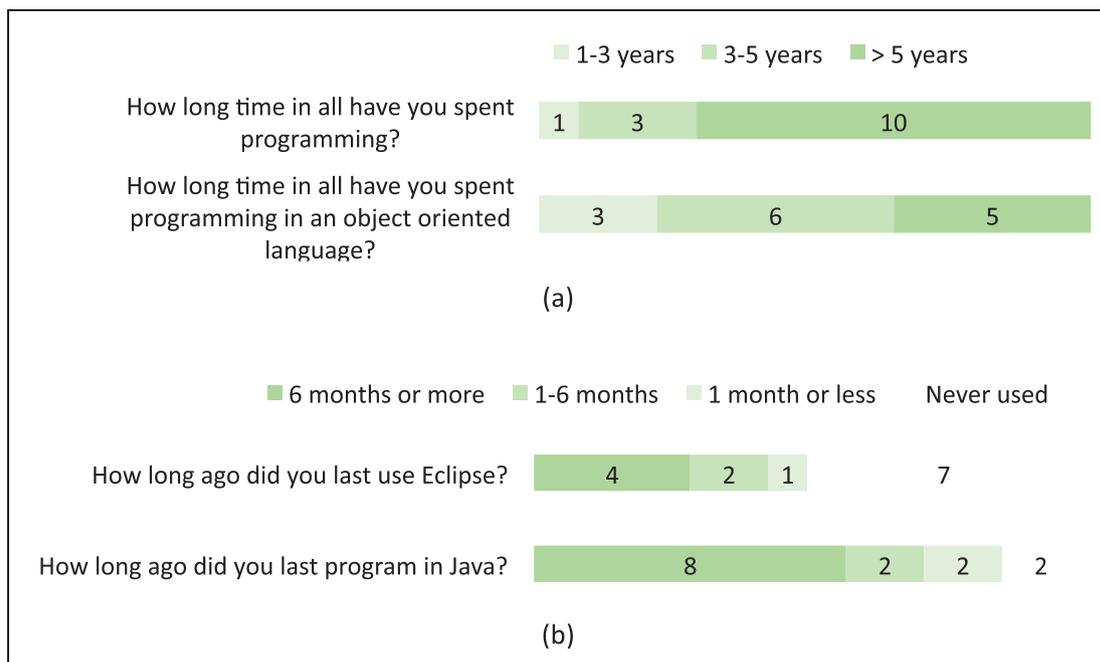


Figure 3. Programming experience of the participants as described by their answers to questions about (a) how long they had spent programming and (b) how long ago they had last used Eclipse or programmed in Java.

from <http://jdraw.sourceforge.net/> contained 23 K-LOC) and 10 tasks used Magelan (magelan-1-3.zip downloaded from <http://sourceforge.net/projects/magelan/> contained 39 K-LOC).

Table 1 gives examples of program investigation tasks for TinyUML (T1, T2, T3), JDraw (J2, J5, J7), and Magelan (M4, M6, M7). Some tasks called for a relatively precise answer (e.g., T3, M6), other tasks called for higher level explanations (e.g., T1, T2, J7, M7), and some called for both precise and high-level answers (e.g., J2, M4). Also, some tasks involved more than one file (e.g., T1 and M7), whereas others focused on code in only one file.

The difficulty of program investigation tasks was aimed at making participants spend about an hour to complete as many tasks as possible; we did not intend for all participants to complete all the tasks. We expected that participants would complete more of the tasks, coming up with equally good or better answers using either of the fisheye interfaces than using the baseline interface.

Controlled tasks. Five types of controlled tasks were used, each of which involved a particular aspect of navigating or understanding source code. In contrast to

program investigation tasks where participants opened source files themselves, a source code file was automatically opened in the editor window for each controlled task. The order in which these types of tasks were used in the experiment was systematically varied. Tasks were taken from previous studies of programming activity.^{5,17} The five types of task were:

Navigate-method tasks, for instance: ‘In the method ‘hasGreen,’ find the return type of the method that is called last.’ Only the method name in the task text was varied between tasks of this type. Participants performed one task of this type with each interface.

Determine-control-structure tasks involved the control structure within a single method. For instance, 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?’ Another task of this type involved finding the closing brace of a block. Participants performed two tasks of this type with each interface.

Determine-dependencies tasks that required determining references to a particular variable or calls to a particular method, for instance: ‘How many methods in this file contain calls to ‘computeProposals’ declared on line 470?’ Participants performed two tasks of this type

Table 1. Examples of program investigation tasks for the three open source programs used in the experiment: TinyUML (tasks labeled T1, T2, T3), JDraw (J2, J5, J7), and Magelan (M4, M6, M7)

T1	Classes AbstractNode and AbstractConnection (in org.tinyuml.draw) are diagram elements. What is the field parent used for in the two classes?
T2	AbstractConnection (in org.tinyuml.draw) contains a field isValid. From inspecting this class, what makes a connection valid or invalid?
T3	DrawingContextImpl.java (package org.tinyuml.draw) uses shapeFactory to provide shapes. Type the name of the methods in the class that use a dashed stroke for drawing. Apart from strokes, what classes of object are obtained from shapeFactory?
J2	Of the methods in FolderPanel.java (package jdomain.jdraw.gui) that call one or more methods on frameFolder, name those that also call frameChanged(). Why do not all methods call frameChanged() after calling methods on frameFolder?
J5	PalettePanel (package jdomain.jdraw.gui) contains rows of eight-colored squares. Clicking on a square in the panel selects a foreground or a background color. Name the class that draws colors as squares in the PalettePanel and explain where a color selection is stored when clicking on a square.
J7	ColourEntry.java (jdomain.jdraw.data) has an invalidate() method. What can make a ColourEntry valid, once it is invalid?
M4	In DefaultDrawingEditor.java (package magelan.core.editor), the field mode describes the current mouse action state. When is SELECT mode initiated and in what program line does this happen? What happens when clicking on an entity modifier when in SELECT mode?
M6	The Hatch class has a style field that affects how the entity is painted. What styles are defined in Hatch and which of them supports line styles?
M7	For each of the entity classes Circle and ImageEntity: How many EntityModifiers do they each have and how do they modify the entity?

with each interface, one task concerning variable references and one task concerning method calls.

Determine-field-encapsulation tasks involved determining whether or not two variables in a class have corresponding get- and set-methods defined, for instance: ‘How many of the fields ‘fText’ and ‘fFont’ have both a get-method and a set-method implemented?’ Only the names of the fields in the task text were varied in these tasks. Participants performed one task of this type with each interface.

Determine-delocalization tasks involved determining delocalization in the source code, for example: ‘The method ‘update’ (line 207–214) contains six method calls. How many of the methods called are declared in this file (that is, excluding inherited methods)?’ Participants performed two tasks of this type with each interface, one task involving variable references and one task involving method calls.

Overall, we expected participants to complete controlled tasks faster using the Permanent interface or the Transient interface than using the Baseline interface.

Materials

The experiment was conducted in a laboratory with six identical computers with 19” CRT monitors at a resolution of 1280 × 1024. On each computer, Eclipse was set up with its window using all available screen space. Tasks were presented in a task view to the left of the editor in Eclipse (Figure 1). Participants typed their answer to the tasks in the task view and clicked a button to continue. In the set of program investigation tasks, the Eclipse window was configured to contain a Package Explorer view above the task view to the left of the editor. In the set of controlled tasks, the Eclipse-window was configured to contain only the editor window and the task view. In all interfaces, the editor window contained 50 lines of text and was 100 characters wide. Before each set of tasks, Eclipse was automatically configured with a workspace containing only

the source code files used for those tasks so that participants could not inadvertently view files that were used in subsequent tasks.

Design. A within-subjects design was used with interface (Permanent, Transient, Baseline) as an independent factor. We wanted each participant to use all three interfaces for at least one hour each. To avoid tiring out participants, the experiment was divided into three blocks to take place on separate days (Figure 4). In each block, participants used one of the three interfaces. The order of interface was systematically varied across participants so as to reduce the influence of learning effects. Also, we systematically varied the order of controlled task types; we wanted to be able to compare the results with those of Jakobsen and Hornbæk.⁵ In contrast, participants performed program investigation tasks before controlled tasks in the experiment. We were not comparing program investigation tasks with controlled tasks as an independent variable, and thus control for learning effects was not required. In fact, the fixed order gave participants time to learn to use the interfaces before performing the controlled tasks. Jakobsen and Hornbæk⁵ found that the fisheye interface might require more time to use effectively, and we thus expected more time to learn to use the interfaces before performing the controlled tasks would increase the reliability of the results in those tasks.

Procedure. In each block of the experiment, participants were first given an introduction to the interface they were about to use. The introduction included a written explanation of the interface and exercises to try the interface. Then, participants performed a set of program investigation tasks. If participants had not finished in 55 minutes, a message dialog informed participants they had five minutes to complete the current task. After the first set of tasks, participants were allowed a break and then continued to perform controlled tasks. For these tasks, participants were

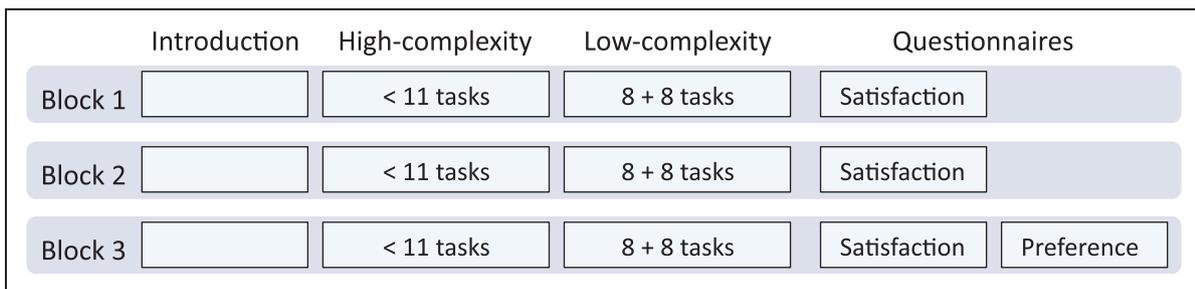


Figure 4. The experimental design in which interface was varied between the three blocks.

instructed to give correct answers as quickly as possible. Participants completed eight training tasks and eight test tasks, and were then administered a questionnaire about the interface just used. The questionnaire included six questions from QUIS¹⁸ and two questions asking about strengths and drawbacks of the interface. After completing the third block of the experiment, participants received a questionnaire asking them to compare the three interfaces and rank them in the order of their preference. The questionnaire also asked the participants for their age, gender, and programming experience.

The experiment was conducted over a period of one week and lasted between four and six hours per participant. Participants met in the laboratory on three different weekdays for each of the three blocks of the experiment, except one participant completed two blocks in one day. Because participants had busy schedules, only few participants met in the laboratory on three consecutive days: we did not control for variation in when participants completed each block, but there were three days at the most between two blocks. Up to six participants were present in the laboratory at a time. Participants were seated so far apart that they could not read the displays of other participants. The experimenter was present in the laboratory to answer questions during the introduction, but otherwise participants completed the experiment unsupervised.

Participants' interactions with the interfaces and answers to tasks were logged. Time used to complete the tasks is derived from the logged data; answers to the tasks were also logged and from the logs accuracy could thus be inferred.

Results

Results from the experiment include objective data (task completion times and accuracy) and subjective data (preference, satisfaction scores, and comments from participants). We also describe data on participants' interaction with the interfaces.

Accuracy and task completion times

We analysed participants' answers to tasks and completion times for each of the two sets of task: program investigation tasks and controlled tasks.

Program investigation tasks. In the first task set comprising program investigation tasks, participants provided 384 answers. Every answer was assigned a score based on an assessment of how correct and complete the answer was. Judged by the first author, 100 answers were accurate wherein participants provided a correct answer that covered all aspects of the task, 151 answers were correct, but missed at least one aspect of the task, and 35 answers were incorrect in at least one aspect but were otherwise correct. Scores 3, 2, and 1 were given to these answers. All other tasks were given a score of 0, including 39 tasks answered incorrectly, 59 tasks that participants abstained from answering (e.g., they did not understand the task), and 64 tasks that participants failed to complete within the 55 minutes. Table 2 summarizes the answers given by participants using the three interfaces. In average, participants spent 49 minutes solving program investigation tasks with each interface. Because of the 55 minutes limit for solving the program investigation tasks in a block, participants only completed all tasks in 23 blocks (55%).

There was no difference in the total score of participants' answers with the interfaces, $F_{1,13} = 0.243$, ns. If anything, participants appeared to complete fewer tasks using the Transient interface than Permanent or Baseline.

Table 3 presents average completion times for program investigation tasks where participants completed all tasks within the time limit. For tasks where participants completed all tasks within the time limit, completion times with the interfaces differed significantly, $F_{2,243} = 4.34$, $p < 0.05$. Although participants appeared to complete fewer tasks using the Transient interface, participants who completed all tasks spent less time with Transient compared with Permanent ($p < 0.05$

Table 2. Summary of answers given to program investigation tasks using the three interfaces

	Score	Baseline	Permanent	Transient	Total
Accurate	3	34	32	34	100
Correct, but incomplete	2	53	55	43	151
Partly incorrect	1	13	11	11	35
Incorrect	0	11	12	16	39
Abstained	0	21	22	16	59
Tasks not completed (no time)	0	18	17	29	64
Participants completing all tasks (number of tasks)		10 (106)	6 (64)	7 (74)	23 (244)

Table 3. Average task completion times in seconds for program investigation tasks using the three interfaces, where participants completed all the tasks. Columns for each interface show number of tasks completed (N), mean task completion time (M), and standard deviation (SD)

Baseline			Permanent			Transient		
N	M	SD	N	M	SD	N	M	SD
74	248	118	106	267	132	64	210	111

Table 4. Average accuracy with the three interfaces for the different types of controlled task. Columns for each interface show number of tasks completed (N), and mean accuracy (M) and standard deviation (SD) in percentage of tasks completed correctly

Task type	Baseline			Permanent			Transient		
	N	M (%)	SD (%)	N	M (%)	SD (%)	N	M (%)	SD (%)
Navigate-method	13	92	28	13	100	0	12	100	0
Determine-control-structure	24	96	20	24	100	0	24	96	20
Determine-dependencies	28	71	46	27	81	40	27	81	40
Determine-field-encapsulation	14	86	36	14	71	47	14	83	38
Determine-delocalization	26	85	37	26	73	45	25	74	44
Average	105	85	36	104	85	36	102	85	36

in Bonferroni adjusted post hoc tests). Completion times did not differ significantly between Transient and Baseline.

Controlled tasks. In the second task set, participants completed 336 controlled tasks. Data from 25 tasks were discarded from our analysis because participants did not appear to understand the question (7), wrote ambiguous answers (5), or wrote verbose answers (12). We could not correct any misunderstandings that participants might have during the training tasks because participants completed the tasks unsupervised. Finally, an outlier that was more than three times above the inter-quartile range for completion time was discarded. The analysis below comprises the remaining 311 tasks.

Overall, 85% of the controlled tasks were answered correctly. The accuracy for different types of task is summarized in Table 4. There was no difference in accuracy with the three interfaces, $F_{1,13} = 0.089$, ns.

Task completion times were not different between interfaces, $F_{1,13} = 0.310$, ns. Average task completion times are summarized in Table 5. While interface was found to interact with task type, $F_{1,13} = 2.19$, $p < 0.05$, there were no significant differences in completion times with the interfaces for any type of task. Below, we compare task-specific completion times with those of Jakobsen and Hornbæk.⁵

For Navigate-method tasks, participants tended to perform slower using Permanent ($M = 43.4$ s) or Transient ($M = 44.1$ s) than Baseline ($M = 38.3$ s), whereas Jakobsen and Hornbæk⁵ found the fisheye interface to be significantly faster than a baseline linear interface in those tasks. A possible explanation is that the Baseline interface in this study showed highlighted occurrences in the overview ruler.

For Determine-control-structure tasks, participants spent about the same time with Permanent ($M = 38.1$ s) and Baseline ($M = 37.9$ s). In the study of Jakobsen and Hornbæk,⁵ participants tended to perform slower for similar tasks that involved finding the closing brace of an enclosing statement. Closing braces were not visible in the context area in that study, whereas braces were visible in the context view in this study. Participants tended to perform slower using Transient ($M = 44.1$ s) compared with Permanent.

For Determine-dependencies tasks, Transient ($M = 49.1$ s) seemed faster than Baseline ($M = 55.1$ s), which in turn seemed faster than Permanent ($M = 55.1$ s). This type of task was not included in the study of Jakobsen and Hornbæk.⁵

For Determine-field-encapsulation tasks, participants seemed to spend more time using Transient ($M = 43.1$ s) compared with Permanent ($M = 36.5$ s) and Baseline ($M = 37.6$ s). One possible explanation that Transient might have been slower is that the

Table 5. Average task completion times with the three interfaces for the different types of controlled task. Columns for each interface show number of tasks completed (N), and mean task completion times (M) and standard deviation (SD) in seconds

Task type	Baseline			Permanent			Transient		
	N	M	SD	N	M	SD	N	M	SD
Navigate-method	13	38.3	16.8	13	43.4	21.5	12	44.1	20.7
Determine-control-structure	24	37.9	20.4	24	38.1	13.5	24	45.3	21.7
Determine-dependencies	28	55.1	22.6	27	58.6	22.8	27	49.4	26.2
Determine-field-encapsulation	14	37.6	20.0	14	36.5	17.7	14	43.1	26.6
Determine-delocalization	26	64.5	27.7	26	51.0	16.4	25	59.6	23.3
Average	105	49.1	24.9	104	47.1	20.1	102	49.4	24.3

context view had to be called up for each variable that participants had to inspect. Whereas this study found no difference between Permanent and Baseline, participants seemed to perform tasks slower using the fisheye interface compared with the baseline interface in the study by Jakobsen and Hornbæk.⁵ Comments by participants indicated that the mechanism for showing semantic relationships, which resulted in frequently changing relationships being shown, complicated the use of the fisheye interface in their study. Although the differences are not statistically significant, we take this as an indication that highlighted occurrences is a more stable mechanism for showing semantic relationships in the fisheye view.

For Determine-delocalization tasks, Permanent ($M = 51.0$ s) seemed faster than Transient ($M = 59.6$ s), which in turn seemed faster than Baseline ($M = 64.5$ s). This result confirms those of Jakobsen and Hornbæk⁵ that found participants to perform these tasks significantly faster (about 51%) with the fisheye interface compared with the baseline interface; however, the difference in this study is smaller and not statistically significant. Similar to Determine-field-encapsulation tasks, Transient might have been slower because the context view had to be called up several times, requiring additional user efforts.

Satisfaction

After having used all three interfaces, participants completed a questionnaire to rank the interfaces. Participants' ranking of the interfaces differed significantly, $F_{1,13} = 0.035$, $p < 0.05$. Figure 5 shows participants' preferences. All but two participants preferred Permanent or Transient, which is a strong indication that they found the fisheye view useful. Also, two-thirds of the participants ranked the Permanent interface first.

Participants rated their satisfaction with the interfaces on six questions. Overall, participants' ratings varied for the three interfaces, though not significantly at the 0.05 level as found by a multivariate analysis of variance, Wilk's Lambda = 0.027, $F_{1,13} = 1.78$, $p = 0.069$. The main reason for this trend was that participants rated the interfaces differently only on a scale from boring to fun ($F_{1,13} = 4.63$, $p < 0.05$), finding both Permanent and Transient more fun to use than Baseline ($p < 0.05$ in Bonferroni adjusted post hoc tests).

Five participants commented that they liked the Transient interface because the fisheye view could be called up temporarily. In contrast, three participants said about Permanent that it was good that the fisheye view was there all the time. However, some participants commented that the fisheye view in Transient 'disappears too easily – has to call it up several times to get all the needed information' and that it was 'confusing when it disappears.' One participant who ranked Baseline as first choice noted in his preference questionnaire that 'if the fisheye view [in Transient] would not disappear all the time, then [Transient] would be ranked 1.' Together, these comments suggest that users may find it useful to be able to switch the fisheye view on and off on demand, so they can use it for longer periods of time than is possible with the short-lived fisheye view in the Transient interface.

Interaction with the interfaces

We analysed the data logged during the experiment to understand how participants used the interfaces. We summarized interaction data from program investigation tasks to measure how participants adopted and used the context view in the fisheye interfaces. We visualized the interaction data from controlled tasks in progression maps (similar to Jakobsen and Hornbæk⁵) and analysed these maps to understand how participants used each interface to solve the

	1st	2nd	3rd
Permanent	9	2	3
Transient	3	7	4
Baseline	2	5	7

Figure 5. Number of participants ranking each interface as first, second, or third choice (from left to right). For example, nine participants ranked Permanent as their first choice.

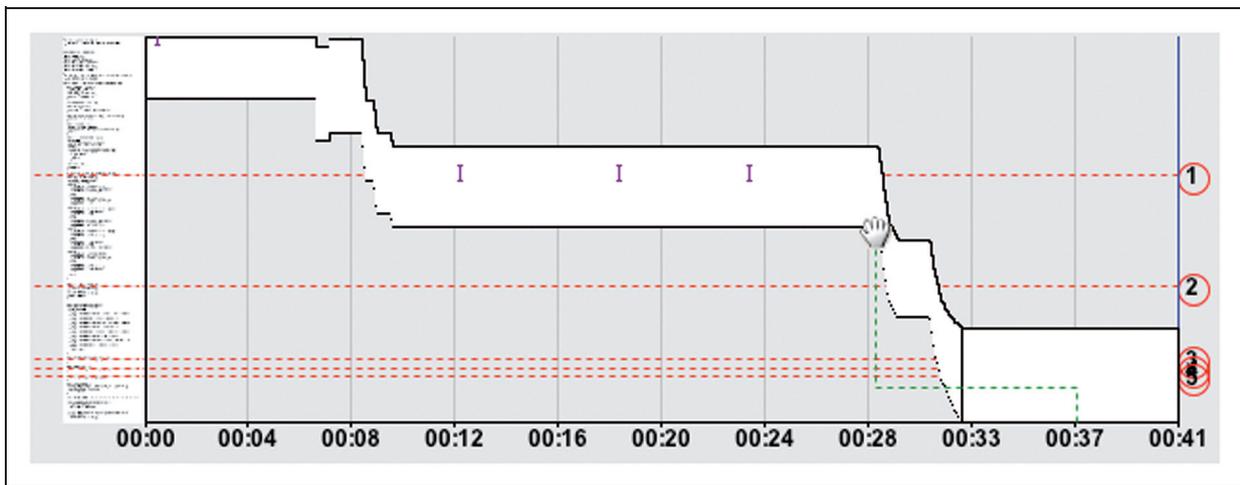


Figure 6. Progression map for a Determine-dependencies task using Permanent interface.

tasks. The progression maps show which part of the file was visible in the focus area at a given time during the task (Figure 6). Program lines are mapped to the y-axis with the first line at the top. Dashed horizontal lines indicate program lines that hold part of the answer to the task. Time is mapped to the x-axis. A solid vertical line indicates that the participant has completed the task. Symbols in the progression maps annotate certain types of interaction (e.g., a hand symbol indicates when the user dragged the scrollbar thumb; a text caret indicates when the user placed the caret in the focus area, for instance to highlight a method; an arrow-in-document symbol indicates when the user clicked in the context view). Other interaction forms are directly discernable from the map, such as scrolling by page up/down keys. Interpreting the progression maps is not always straightforward. For instance, the task shown in Figure 6 involves finding calls to a particular method. The user places the text caret after 12 seconds and then two more times, presumably in the method, before scrolling to see the highlighted occurrences. It is not clear in this task, however, why the user places the caret three times. Because program investigation tasks varied in

structure, and in some cases involved multiple files, we did not use progression maps to analyse those tasks.

Program investigation tasks. In average, participants interacted with the context view in 64% of the program investigation tasks they completed using the Permanent interface and 70% of the tasks using the Transient interface. Participants used the context view to navigate in the code an average of 11 times across all tasks in a task set, equally often with the Permanent interface and the Transient interface. While the context view was always shown in the Permanent interface, participants had to explicitly call up the context view to use it in the Transient interface – they did so 27 times in average across all tasks in a task set.

Using the Permanent interface, 10 participants interacted with the context view in more than half the tasks. Participants may also have looked at information in the context view without interacting with it, but we were unable to determine such use from the data logged in program investigation tasks. Using the Transient interface, two participants did not once

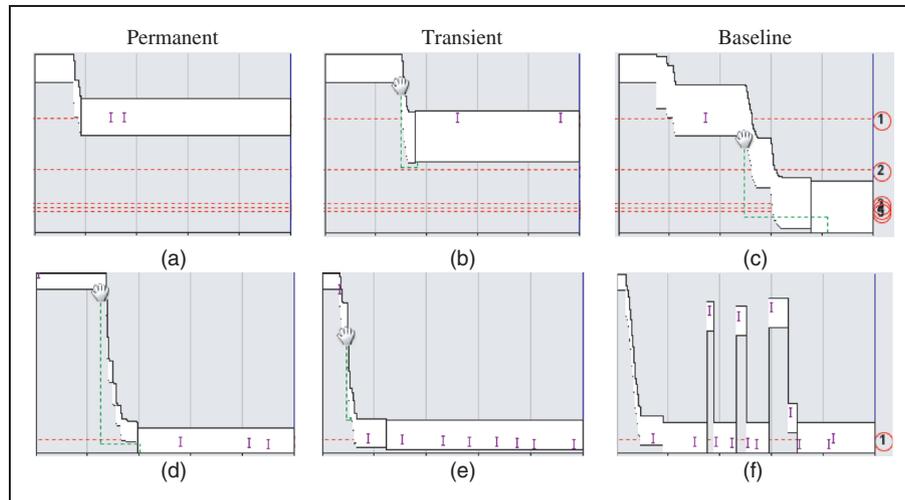


Figure 7. Progression maps representative of participants' navigation when using the three interfaces in: (a–c) a Determine-dependencies task involving method calls and (d–f) a Determine-delocalization task.

use the context view, whereas the other 12 participants used the context view in more than half of the tasks.

Controlled tasks. Analysis of progression maps for controlled tasks revealed patterns in the participants' interaction with the interfaces. In all controlled tasks except for Determine-control-structure tasks, participants most often selected a method or variable and used its highlighted occurrences to either navigate more quickly or to avoid navigating. Figure 7 shows progression maps that are representative of this type of interaction using each of the interfaces for completing a Determine-dependencies task (Figure 7(a)–(c)) and a Determine-delocalization task (Figure 7(d)–(f)). Using the Permanent interface or the Transient interface, participants could often find the lines in the context view that contained the answer to the task without navigating further. This can be seen in the progression map in Figure 7(a) for a task where the participant must find calls to a particular method: the view is moved only once, at the beginning of the task, to bring the line containing the particular method into view. Next, the participant selects the method (indicated by the text caret symbol) and can then find lines containing calls to the method in the context view. Using the Baseline interface, participants often seemed to navigate to lines with highlighted occurrences, which might contain the answer to the task. This can be seen in the progression map in Figure 7(c), which shows that the participant, after having selected the method, scrolls the view using the mouse to bring each call to the method into view. Below we describe the different interactions used to solve the tasks and how frequently they were used by participants.

Using the Permanent interface, participants were able to find the answer to 54 of 84 tasks directly in the context view with minimal navigation (as described above; Figure 7(a) and (d)). Participants navigated to occurrences in the context view to find the answer in six tasks. In contrast, in 24 tasks participants navigated to occurrences by scrolling or by clicking in the overview, or they manually searched the file. Using the Transient interface, participants called up the context view in 55 of 84 tasks and found the answer there with minimal navigation (Figure 7(b) and (e)). In 28 tasks, participants navigated to occurrences by scrolling or by clicking in the overview, or they scrolled to manually search through the file. Using the Baseline interface, participants completed 68 of 84 tasks by finding occurrences in the overview ruler instead of manually searching through the file. Often participants then navigated to occurrences either by scrolling like in Figure 7(c) (39 tasks), or by clicking in the overview ruler like in Figure 7(f) (27 tasks). Participants solved two Determine-delocalization tasks without scrolling or navigating to occurrences, but seemingly by examining the white rectangles showing occurrences in the overview ruler.

In all interfaces, participants made effective use of highlighted occurrences for navigating. However, in Determine-dependencies tasks where participants should determine which methods contained value assignments to a particular variable (shown in Figure 8), all participants using the Baseline interface ended up scrolling to search manually through the entire file. Similarly, six participants using Permanent and four using Transient scrolled through the entire file to solve the task. This was surprising because all participants navigated

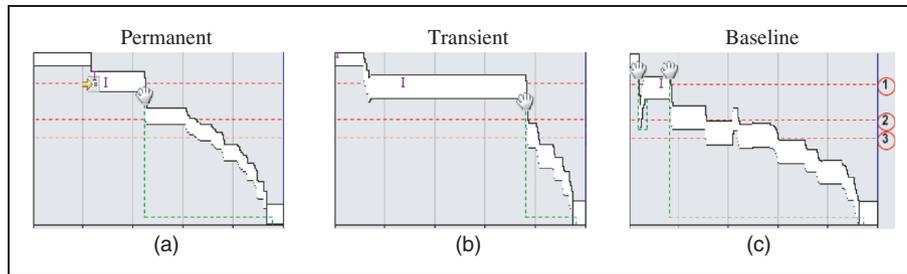


Figure 8. Example progression maps where participants scrolled through the entire file to solve a Determine-dependencies task involving variable assignments.

effectively using occurrences to solve Determine-dependencies tasks where they should determine which methods contained calls to a particular method (Figure 7(a)–(c)).

Determine-control-structure tasks asked participants to find the ‘}’-brace that closes a given block, or asked participants to count the for-, if- and while-statements that enclose a given line. Using Baseline, participants had to scroll to find the closing brace or enclosing statements. Using Permanent, six participants found the line number of the closing brace in the context view whereas two navigated to the closing brace; seven participants read enclosing statements in the context view. Using Transient, five participants called up the context view, and three of these read the line number whereas two navigated to the closing brace; nine participants called up the context view and read the enclosing statements.

Two participants did not once call up the fisheye view using the Transient interface, and using the Permanent interface, they seemed to use the fisheye view only in program investigation tasks. Those two participants were the only ones with no Java experience. The three participants who preferred the Transient interface used the fisheye view in all controlled tasks. In program investigation tasks, two of these participants used it frequently, whereas one used it only occasionally.

Discussion

We now relate the findings from our study to previous research in focus + context interfaces of source code. We then discuss issues with the transient use of fisheye interfaces in programming based on our results.

Focus + context interfaces for source code

Results from the study confirm earlier empirical findings in support of fisheye views of source code⁵ and code elision¹⁷. All but two participants preferred either the Permanent or Transient interface, which contained a fisheye view of code, compared with

the Baseline interface, which contained a linear view. In contrast to Jakobsen and Hornbæk,⁵ however, time and accuracy measures were inconclusive. Data logged during the experiment show that participants often used semantic highlighting of related code. Using either Permanent or Transient interface, participants could often find the answer directly in the context view with minimal navigation, whereas using the Baseline interface, participants had to navigate to find the answer in many of the tasks. Highlighting might have helped participants navigate faster also in the Baseline interface, especially by use of the overview ruler. In interpreting our results, it is therefore important to note that highlighting was not included in previous studies of focus + context views of source code. However, highlighting is a common feature in code editors and therefore perhaps well known to participants. In contrast, the fisheye view is not well known. Participants in our study may not have had time in the study to learn to use it effectively. Even longer term studies may uncover how fisheye views are used when fully learned and adopted by users.

More work is needed to better utilize fisheye interfaces in real-life programming. An advantage of the fisheye interfaces that automatically change the view is that users can see parts of the file that are related to their focus, even if those parts are located far apart in the file. In practice, programming tasks involve code in multiple files. A clear limitation of the fisheye interfaces studied here is that they provide context for the user’s focus only within a single source code file. A challenge for future work is extending the fisheye view to provide context across the entire code base, by integrating code fragments located in multiple files. Moreover, we have studied fisheye interfaces for programming only on displays of moderate size. The widespread of larger displays begs the question how display size influences the usefulness of these fisheye interfaces. Programmers can view multiple files on a large display, yet there is a cost to manually arranging the views (e.g., to view different parts of a

source code file). Last, modern programming environments provide tools for navigating in source code, but we restricted the tools available in the experiment. In practice, programmers may choose between multiple approaches to, for instance, navigate dependencies in the code. Further work is needed to understand the benefits that programmers gain from fisheye interfaces compared with these tools. Issues with existing tools for navigating in source code, which fisheye interfaces may alleviate, are ‘loss of context’¹⁹ and ‘navigational overhead’.²⁰

Transient use of a fisheye view

We compared the usability of a transient fisheye view, which participants could call up temporarily, to a permanent fisheye view. The transient fisheye view aimed to support navigation and understanding while still providing a large view of source code for other tasks such as reading and editing. Analysis of participants’ interaction with the interfaces showed effective use of the fisheye view in both the Permanent interface and the Transient interface. Also, some participants’ comments confirm the idea of a fisheye view that can be called up temporarily. However, only two participants preferred the Transient interface. From participants’ feedback, we learned about issues that might have detracted from the usability of the transient fisheye interface. Below we discuss these issues and other factors that might have influenced participants’ use of the transient fisheye view.

First, results from this study may be contrasted to the empirical findings of Jakobsen and Hornbæk.⁴ That study showed preference for a transient overview, which appeared temporarily close to the mouse cursor, compared to a fixed overview, which was shown permanently in the display. In contrast to the tasks used by Jakobsen and Hornbæk⁴, which focused narrowly on navigation, participants in this study performed more varied tasks in a more complex domain. For instance, participants used the fisheye view for navigating, but also for understanding relationships in the code.

Some participants mentioned that the context view in the Transient interface disappeared too easily. We suspect this may have been a problem in tasks that involved determining enclosing statements. These types of task involved aligning indentation of lines in the context view to lines in the focus area. In contrast, we think it is appropriate that the context view disappears after having called up the context view to navigate in the code. However, an interesting alternative, which was hinted at by some participants’ comments, is to allow users to switch the fisheye view on and off on demand.

We hypothesized that the Transient interface would benefit from a large context view that allowed more important lines to be visible simultaneously. We had expected that users would call up the context view to use the information contained therein, and therefore not pay much attention to the focus area. However, several participants commented that the context view used too much space in the transient fisheye view. In the experiment, participants may have needed to relate information in the context view to information located in a part of the editor window that became hidden by the context view. One way to minimize the risk of covering code in the editor with the context view is to place the context view outside the bounds of the editor window as far there is display space above and below the editor window.

We suggest that a transient visualization may support a specific task more effectively by allowing users to call up a representation of only the types of information useful to that task. The fisheye view in the Transient interface was based on the same DOI function as in the Permanent interface and thus the fisheye views in the two interfaces included the same types of information. In practice, a transient fisheye view of source code could prove more effective if aimed at helping programmers to understand only certain relationships in the code, and include only lines that show those relationships in the context view. However, more work is needed to determine how users can more directly control the focus used in the DOI function underlying the fisheye view.

Limitations

The experiment has limitations that need to be taken into account when interpreting the results. We discuss the tasks, participants, materials, and the procedure used. First, although we included varied programming tasks, the tasks involved only reading and navigating in source code, and are thus not representative of real-life programming activity. Participants did not write code or have all the tools available in modern programming environments at their disposal. Consequently, our study may have emphasized tasks for which the fisheye view is particularly useful and therefore favored the Permanent interface. Second, only students participated in the experiment. Although most participants had several years of programming experience, the results might not generalize to experienced professional programmers. Third, open source programs were used for the experimental tasks. Although none of the participants said they had seen the source code for these programs before, we did not explicitly screen for this potentially confounding circumstance. Also,

several participants were in the laboratory at the same time and participants knew each other as students. This might have introduced a level of competition among participants, which could influence the results. Finally, there is potential bias in participants' subjective ratings in favor of the fisheye interfaces because participants might have considered the experimenter as stakeholder in the fisheye interfaces. However, such a bias in participants' ranking of Permanent and Transient interfaces is unlikely.

Conclusion

Transient visualizations promise to support specific contexts of use without making permanent changes to the user interface. To further understand how transient visualization can be used to support complex work, we have designed and evaluated an interface with a transient fisheye view of source code that users can call up temporarily. In a user study, we compared the transient fisheye interface with a permanent fisheye interface and a baseline interface. Fourteen participants performed tasks of both high and low complexity.

Results from the user study showed that all but two participants preferred either of the interfaces containing a fisheye view compared to the baseline interface. This supports results from earlier studies of source code views.^{5,17} The transient fisheye view aimed at supporting navigation and understanding tasks while still providing a large view of source code for reading and editing. However, participants preferred a permanent fisheye view to the transient fisheye view. No clear differences in task completion times and accuracy were found, and analysis of participants' interaction showed that the fisheye view was used equally often in permanent and transient conditions. Participants' comments indicate subtle issues with the transient fisheye interface that might have detracted from its usability.

We have concluded by proposing ideas to improve transient use of fisheye views in existing user interfaces. For instance, when temporarily called up, the context view may be extended to use display space adjacent to the existing view so as to avoid hiding information in the user's focus of attention. Also, we propose using a DOI function that focuses narrowly on information important in a specific task; a transient fisheye view tailored for a specific task may increase its effectiveness.

References

1. Card SK, Mackinlay JD and Shneiderman B. *Readings in Information Visualization: Using Vision to Think*. San Diego, CA: Academic Press, 1999.
2. Eick SG, Steffen JL and Sumner EE. SeeSoft: a tool for visualizing line oriented statistics software. *IEEE Trans Software Eng* 1992; 18: 957–968.
3. Robertson GG and Mackinlay JD. The document lens. *UIST '93: Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology* (Atlanta, Georgia), ACM: New York, 1993; 101–108.
4. Jakobsen MR and Hornbæk K. Transient visualizations. *OZCHI '07: Proceedings of the 19th Australasian Conference on Computer-Human Interaction* (Adelaide, Australia), ACM: New York, 2007; 69–76.
5. Jakobsen MR and Hornbæk K. Evaluating a fisheye view of source code. *CHI '06: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada), ACM Press: New York, 2006; 377–386.
6. Furnas GW. The fisheye view: a new look at structured files. Bell Laboratories Technical Memorandum #81-11221-9, 1981.
7. Baudisch P, Lee B and Hanna L. Fishnet, a Esheye web browser with search term popouts: a comparative evaluation with overview and linear view. *AVI '04: Proceedings of the Working Conference on Advanced Visual Interfaces* (Gallipoli, Italy), ACM Press: New York, 2004; 133–140.
8. Fekete J and Plaisant C. Excentric labeling: dynamic neighborhood labeling for data visualization. *CHI '99: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Pittsburgh, PA), ACM Press: New York, 1999; 512–519.
9. Terry M and Mynatt ED. Side views: persistent, on-demand previews for open-ended tasks. *UIST '02: Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology* (Paris, France), ACM Press: New York, 2002; 71–80.
10. Zellweger PT, Regli SH, Mackinlay JD and Chang B. The impact of fluid documents on reading and browsing: an observational study. *CHI '00: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Hague, Netherlands), ACM Press: New York, 2000; 249–256.
11. Kurtenbach G, Fitzmaurice GW, Owen RN and Baudel T. The Hotbox: efficient access to a large number of menu-items. *CHI '99: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Pittsburgh, PA), ACM Press: New York, 1999; 231–237.
12. Bier EA, Stone MC, Pier K, Buxton W and DeRose TD. Toolglass and magic lenses: the see-through interface. *SIGGRAPH '93: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques* (Anaheim, CA), ACM Press: New York, 1993; 73–80.
13. Becker RA and Cleveland WS. Brushing Scatterplots. In: Meeker Jr WQ (ed.) *Technometrics*. Vol. 29, Alexandria, VA: American Society for Quality Control and American Statistical Association, 1987, pp.127–142.
14. Igarashi T, Mackinlay JD, Chang B and Zellweger PT. Fluid Visualization of Spreadsheet Structures. *VL '98: Proceedings of the IEEE Symposium on Visual Languages* (Halifax NS, Canada), IEEE Computer Society: Los Alamitos, 1998; 118–125.
15. Bezerianos A and Balakrishnan R. The vacuum: facilitating the manipulation of distant objects. *CHI '05: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Portland, Oregon), ACM Press: New York, 2005; 361–370.
16. Irani P, Gutwin C and Yang XD. Improving selection of off-screen targets with hopping. *CHI '06: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada), ACM: New York, 2006; 299–308.
17. Cockburn A and Smith M. Hidden messages: evaluating the efficiency of code elision in program navigation. *Interact Comput* 2003; 15(3): 387–407.

18. Chin JP, Virginia A and Norman KL. Development of an instrument measuring user satisfaction of the human-computer interface. *CHI '88: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Washington, DC), ACM Press: New York, 1988; 213–218.
19. Alwis B and Murphy GC. Using visual momentum to explain disorientation in the Eclipse IDE. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (Brighton, UK), IEEE: Los Alamitos, 2006; 51–54.
20. Ko AJ, Coblenz MJ and Aung HH. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans Software Eng* 2006; 32(12): 971–987.