INFORMATION VISUALIZATION IN PROGRAMMING ENVIRONMENTS

MIKKEL RØNNE JAKOBSEN

PHD THESIS

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF COPENHAGEN

AUGUST 2009

The Graduate School of Science, Faculty of Science, University of Copenhagen, Denmark

ABSTRACT

Programming is challenging work. Programmers must navigate and understand large and complex source code, and the careful coordination of many peoples' efforts is often required. Information visualization promises to help programmers cope with these challenges. On this basis, the use of visualization in programming is investigated in empirical studies.

Three studies show evidence of the usefulness of fisheye interfaces for navigating and understanding source code. Overall, participants in two experiments prefer using fisheye interfaces. Also, participants in a long-term field study adopt and use a fisheye interface in their own work. However, the fisheye interfaces that were studied do not seem useful in all tasks and problems detract from the usability of the interfaces. Issues that seem fundamental to the design and use of fisheye views are highlighted for further research. Analysis of participants' interaction with the fisheye interfaces shows that participants perform some tasks with less physical effort. It seems useful to show readable information in the fisheye interfaces and participants find semantically related information the most useful.

The results suggest that a particular fisheye interface may not support all tasks in programming equally well. We argue that transient use of visualization may support specific infrequent tasks without permanently changing the interface. Two comparative evaluations of transient and permanent interfaces show no significant differences in task performance. Participants prefer a transient interface in one study and a permanent interface in the other. Results suggest benefits of a transient interface but also suggest problematic issues in designing transient visualizations for complex work.

A visualization to support coordination in programmer teams is investigated. Results suggest two uses of such visualization: for use during meetings and for occasional use to maintain awareness of the team's work. Issues with the visualization's design distinct to these two uses are found in a field study of the visualization deployed with two teams.

Multiple research methods are used to investigate fisheye interfaces in programming. Limitations of individual methods are thus offset and richer data are collected for analyzing and understanding how fisheye interfaces are used in programming. Regardless of the method used, however, participants seem to lack proficiency in using the fisheye interfaces. Finally, the field study triangulates multiple methods: experience sampling, logging, thinking aloud, and interviews. Benefits of combining multiple methods are identified and opportunities for improving the triangulation approach are suggested for further research.

DANSK RESUMÉ

Programmering er udfordrende arbejde. Programmører skal navigere og forstå omfattende og kompliceret kildetekst, og nøje koordinering af mange personers indsats er ofte nødvendig. Informationsvisualisering kan potentielt hjælpe programmører til at overkomme disse udfordringer. På denne baggrund undersøges anvendelsen af visualisering i programmering i empiriske studier.

Tre undersøgelser viser tegn på fiskeøjegrænsefladers brugbarhed til at navigere og forstå kildetekst. Samlet foretrækkes fiskeøjegrænseflader af forsøgspersonerne i to eksperimenter. Endvidere tager forsøgspersonerne i et langtidsfeltstudie en fiskeøjegrænseflade i brug og anvender den i deres eget arbejde. Fiskeøjegrænsefladerne som undersøges synes dog ikke nyttige i alle opgaver og der er problemer som forringer deres brugsvenlighed. Spørgsmål som synes grundlæggende for fiskeøjevisningers design og brug fremhæves til videre forskning. Analyse af forsøgsdeltagernes interaktion med fiskeøjegrænsefladerne viser at nogle opgaver udføres med mindre fysisk anstrengelse. Det synes fordelagtigt at vise læsbar information i fiskeøjegrænsefladen og forsøgsdeltagerne finder størst nytte af semantisk relateret information.

Resultaterne antyder at visse fiskeøjegrænseflader ikke understøtter alle opgaver i programmering lige godt. Vi argumenterer for at flygtig brug af visualisering kan understøtte specifikke, ikke-hyppige opgaver uden at grænsefladen ændres permanent. To sammenlignende evalueringer af flygtige og faste grænseflader viser ingen signifikante forskelle i mål for opgaveløsning. Samlet foretrækker forsøgspersonerne en flygtig grænseflade i den ene undersøgelse og en fast grænseflade i den anden. Resultater antyder fordele ved en flygtig grænseflade, men også problemer i design af flygtige visualiseringer til komplekst arbejde.

En visualisering til støtte af koordinering i udviklingsteam undersøges. Resultater antyder to anvendelser af en sådan visualisering: til brug i forbindelse med møder og til lejlighedsvis at vedligeholde opmærksomhed om teamets arbejde. Problemer med visualiseringens design vedrørende disse to anvendelser afdækkes i et feltstudie af visualiseringen med to team.

Flere forskningsmetoder bruges i undersøgelse af fiskeøjegrænseflader i programmering. Begrænsninger ved individuelle metoder opvejes således og omfattende data indsamles til at analysere og forstå hvordan fiskeøjegrænseflader bliver brugt i programmering. Uanset hvilken metode der anvendes, synes forsøgspersonernes kunnen i fiskeøjegrænsefladernes brug dog at være begrænset. Endelig bruger feltstudiet triangulering af flere metoder: experience sampling, logning, tænke-højt, og interview. Fordele ved at kombinere flere metoder identificeres og muligheder for at forbedre trianguleringsmetoden foreslås til yderligere forskning.

PREFACE

This thesis is submitted to obtain the PhD degree at the Department of Computer Science, Faculty of Science, University of Copenhagen (DIKU). The work described in the thesis was carried out between June 2006 and July 2009.

The thesis consists of two parts. First, the introduction summarizes the contributions in the PhD. Second, five papers comprise the main part of the thesis. The papers are referred to with the numbers 1 to 5 and are listed on page 6. The full papers are included from page 18 and on.

Thanks are due first of all to my supervisor Kasper Hornbæk. He sparked my interest in research and it has been a pleasure to learn from his experience ever since. Kasper is always inspiring, trusting, and helpful. I thank him for that. I also thank my other colleagues at DIKU. In particular, I have had many rewarding discussions with Erik Frøkjær and Tobias Uldall-Espersen. I am grateful to Erik for his guidance and generous help, and to Tobias for being a role model PhD student, which has made life as a PhD student at DIKU a little easier for me. I visited Microsoft Research in Redmond in the summer of 2008, and foremost I thank the people there for a worthwhile collaboration. In particular, I thank Mary Czerwinski for making my visit possible, and George Robertson for welcoming me and for introducing me to the people in the VIBE and HIP groups from whom I learned much. Many employees and visiting interns at Microsoft Research contributed to making my stay in Washington a great experience—thanks. Also thanks to friends and former colleagues who contributed insight into their programmer experiences, helped in ongoing evaluation of my interface designs, or helped me gain access to busy programmers for my study. Last, I thank my wife and family for their great support.

Mikkel Rønne Jakobsen
Copenhagen, August 2009

CONTENTS

Abstract **1**

Dansk resumé **2**

Preface **4**

Contents **5**

List of papers **6**

Introduction **7**

Background 7

Contributions 8

Abstracts of papers 8

Fisheye interfaces 9

Transient visualizations 10

Visualizations to support team coordination 11

Evaluation of visualization techniques in complex work 12

Conclusion **14**

References **15**

Paper 1 – Evaluating a Fisheye View of Source Code **18**

Paper 2 – Transient Visualizations **29**

Paper 3 – Transient or Permanent Fisheye Views: A Comparative Evaluation of Source Code Interfaces **38**

Paper 4 – Fisheyes in the Field: Using Method Triangulation to Study the Adoption and Use of a Source Code Visualization **47**

Paper 5 – WIPDash: Work Item and People Dashboard for Software Development Teams **58**

LIST OF PAPERS

Paper 1: Jakobsen, M. R. and Hornbæk, K. (2006). Evaluating a fisheye view of source code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada, April 22 - 27, 2006). CHI ’06. ACM, New York, NY, 377-386.

Paper 2: Jakobsen, M. R. and Hornbæk, K. (2007). Transient visualizations. In *Proceedings of the 19th Australasian Conference on Computer-Human interaction: Entertaining User interfaces* (Adelaide, Australia, November 28 - 30, 2007). OZCHI '07, vol. 251. ACM, New York, NY, 69-76.

Paper 3: Jakobsen, M. R. and Hornbæk, K. (2009). Transient or Permanent Fisheye Views: A Comparative Evaluation of Source Code Interfaces, 8 pages, under review.

Paper 4: Jakobsen, M. R. and Hornbæk, K. (2009). Fisheyes in the field: using method triangulation to study the adoption and use of a source code visualization. In *Proceedings of the 27th international Conference on Human Factors in Computing Systems* (Boston, MA, USA, April 04 - 09, 2009). CHI '09. ACM, New York, NY, 1579-1588.

Paper 5: Jakobsen, M. R., Fernandez, R., Czerwinski, M., Inkpen, K., Kulyk, O., and Robertson, G. (2009). WIPDash: Work Item and People Dashboard for Software Development Teams. 14 pages, to appear in *Proceedings of INTERACT 2009 - 12th IFIP TC13 Conference in Human-Computer Interaction* (Uppsala, Sweden, August 24-28, 2009).

INTRODUCTION

This thesis is about the use of information visualization and interaction techniques to support programming. Before summarizing the contributions made, the background for the thesis is outlined.

BACKGROUND

Developing a computer-based system is a challenging undertaking in which programmers play a key role. Programmers are faced with increasingly complex systems and demands for high quality and reliability. Large and complex systems are difficult to understand and they necessitate the coordination of many programmers' efforts. Thus programming has long been considered challenging both as an individual activity and as a team activity (Weinberg, 1971).

Programming is challenging as an individual activity because it is mentally demanding. For instance, the programmer often must understand a program by looking at its source code (Latoza et al., 2006; Ko et al., 2007). For many programs, the source code is large and complex, and has often been developed over several years, by many people. For instance, the Debian 4.0 system contains program packages of a mean size of 28,544 lines of code, 288 million lines in all, developed over more than ten years, by more than 1,000 programmers (Gonzales-Barahona et al., 2009; Debian.org, 2009). Consequently, navigating the source code to understand a program can be mentally very demanding.

Programming is challenging as a team activity because it typically requires the coordinated efforts of many people. The programmers in a team must share a common view of how the system should work, and they must coordinate their work so that it fits together without redundancy and on time (Kraut and Streeter, 1995). Coordination in teams therefore adds to the difficulties of programming.

Programming environments, languages, methods, and tools are continuously being developed to lessen the difficulties of programming, but many difficulties persist. As an example, object-oriented programming in widespread use today may ease the design of complex systems. However, object-oriented programming is likely to cause delocalization, which makes it hard to understand source code because conceptually related pieces of code are located in non-contiguous parts of a program (Soloway et al., 1988; Dunsmore et al., 2000). Faced with such difficulties, a hope is for innovations in programming environments that amplify programmers' abilities.

One way to amplify the abilities of programmers is through information visualization—use of interactive visual representations of abstract data to amplify human cognition (Card et al., 1999). An early example of information visualization in programming is the SeeSoft system (Eick et al., 1992). SeeSoft uses color to represent statistics associated with the lines of text, and fits 50.000 lines in a 1280 x 1024 pixel display. When SeeSoft is used on source code, the programmer can see for example which parts of the program that have been frequently modified. If adopted in programming environments, information visualization thus promises to amplify the cognitive abilities of programmers.

Many innovative visualization techniques have been developed, but few have been adopted and used in real-life work. For wider adoption of visualization techniques, their usefulness must be convincingly demonstrated, key factors in their design must be understood, and guidelines must be provided to ease their design. Fortunately, evaluations are helping us understand how different techniques influence users' abilities and behavior in particular tasks (e.g., Cockburn and McKenzie, 2003; Hornbæk and Frøkjær, 2003), or demonstrate the usefulness of specific visualization systems to professionals in specific fields (e.g., Saraiya et al., 2005; Seo and Shneiderman, 2006). Although information visualizations are increasingly being evaluated, the potential uses and limitations of many visualization techniques are not clearly understood.

CONTRIBUTIONS

This thesis provides empirically founded insight into the use of information visualization to support programmers. Two uses of visualization have been investigated; the first aimed at supporting navigation and understanding of source code, the second at supporting coordination in collocated programmer teams. Contributions fall in four areas: (1) fisheye interfaces that aim to support navigation and understanding of source code, (2) transient use of visualization to support specific tasks in complex work, (3) use of visualization to support coordination in programmer teams, and (4) evaluation of visualization techniques in complex work.

ABSTRACTS OF PAPERS

To give an overview of the five papers comprising this thesis, the abstracts of the papers are included below.

Paper 1: Evaluating a Fisheye View of Source Code
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.

Paper 2: Transient Visualizations
Information visualizations often make permanent changes to the user interface with the aim of supporting specific tasks. However, a permanent visualization cannot support the variety of tasks found in realistic work settings equally well. We explore interaction techniques that transiently visualize information near the user's focus of attention. Transient visualizations support specific contexts of use without permanently changing the user interface, and aim to seamlessly integrate with existing tools and to decrease distraction. Examples of transient visualizations for document search, map zoom-outs, fisheye views of source code, and thesaurus access are presented. We provide an initial validation of transient visualizations by comparing a transient overview for maps to a permanent visualization. Among 20 users of these visualizations, all but four preferred the transient visualization. However, differences in time and error rates were insignificant. On this background, we discuss the potential of transient visualizations and future directions.

Paper 3: Transient or Permanent Fisheye Views: A Comparative Evaluation of Source Code Interfaces
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 tasks of both high and low complexity so as to investigate varied programming activity. All participants used each of 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 analyze 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.

Paper 4: Fisheyes in the Field: Using Method Triangulation to Study the Adoption and Use of a Source Code Visualization
Information visualizations have been shown useful in numerous laboratory studies, but their adoption and use in real-life tasks are curiously under-researched. We present a field study of ten programmers who work with an editor extended with a fisheye view of source code. The study triangulates multiple methods (experience sampling, logging, thinking aloud, and interviews) to describe how the visualization is adopted and used. At the concrete level, our results suggest that the visualization was used as frequently as other tools in the programming environment. We also propose extensions to the interface and discuss features that were not used in practice. At the methodological level, the study identifies contributions distinct to individual methods and to their combination, and discusses the relative benefits of laboratory studies and field studies for the evaluation of information visualizations.

Paper 5: WIPDash: Work Item and People Dashboard for Software Development Teams
We present WIPDash, a visualization for software development teams designed to increase group awareness of work items and code base activity. WIPDash was iteratively designed by working with two development teams, using interviews, observations, and focus groups, as well as sketches of the prototype. Based on those observations and feedback, we prototyped WIPDash and deployed it with two software teams for a one week field study. We summarize the lessons learned, and include suggestions for a future version.

FISHEYE INTERFACES

A fisheye interface combines local detail and global context in a single view of an information space (Furnas, 1981). One type of fisheye interface in programming aims to help navigating and understanding source code by displaying those parts of the code that have the highest degree of interest given the programmer's current focus. We implemented such a fisheye view in the source code editor of a widespread programming environment. Based on that implementation, we compared fisheye views to linear views of source code (papers 1 and 3), and we studied the long-term use of a fisheye interface in programming (paper 4).

Overall, we find evidence in support of fisheye interfaces' usefulness to programmers. Participants in two controlled experiments (paper 1 and 3) preferred a fisheye interface to a linear source code interface. Participants in a field study (paper 4) adopted and used the fisheye interface regularly and across different activities in their own work for several weeks. The fisheye interface does not seem useful in all tasks and activities, however. Participants in one experiment (paper 1) completed tasks significantly faster using the fisheye interface, a difference of 18% in average completion time, but differences were only found for some task types. Although the results indicate usability issues, they also suggest that some tasks were less well supported by the fisheye interface. In addition, data from the field study (paper 4) showed periods where programmers did not use the fisheye interface, and debugging and writing new code were mentioned as activities for which the fisheye interface was not useful.

Specifically, the fisheye view enables programmers to perform some tasks with less physical effort compared with a normal linear view of source code. This was found in controlled experiments (paper 1 and 3) by analyzing in detail how participants' performed tasks with a fisheye view compared with a linear view. Using a fisheye view, participants directly used information in the context area or navigated with sparse interaction; they read program lines in the context area or clicked in the context area to jump to a particular line. We saw this behavior also in observations of programmers using a fisheye interface in real-life work (paper 4).

Consistently across our studies (paper 1, 3, and 4), we found that lines semantically related to the user's focus were the most important—such lines were the most frequently used and were the most often mentioned by participants as a benefit of the fisheye interface. An interesting proposition is to expand on the use of semantically related lines in the fisheye interface with program slicing. Tools for program slicing have been found useful for debugging in that they help programmers to localize code that contain program faults (Weiser and Lyle, 1986; Francel and Rugaber, 2001). Integration of slicing in the fisheye interface could thus potentially increase the utility of the fisheye interface in programming.

Our results support earlier findings that visualization techniques combine with highlighting of occurrences to different effect (Baudisch et al., 2004). In one experiment (paper 1), the fisheye interface resulted in faster overall task performance compared with a baseline interface using a linear view of source code, whereas no differences in task performance were found in a later experiment (paper 3). All three interfaces used in the latter experiment featured semantic highlighting of code and an overview ruler that showed highlighted occurrences. Results from analysis of participants' interaction suggest that the highlighting helped participants navigate to occurrences in all interfaces. However, a notable benefit of the fisheye interface is that some tasks can be performed with less navigation compared with an overview, because code containing the highlighted occurrences and their surrounding context can be directly read in the fisheye interface.

The fisheye interfaces studied here automatically change the view to include context information related to the user's focus. Earlier research has studied another type of fisheye interface, called elision interface in the following, that requires users to manually expand or collapse parts of the document (Cockburn and Smith, 2003; Hornbæk and Frøkjær, 2003). The two types of interface share the difficulty of determining which document parts are important a priori so that the resulting view is useful across tasks (paper 1; Hornbæk and Frøkjær, 2003). An advantage of fisheye interfaces that automatically change the view is that users can see document parts that are related to their focus, even if the parts are located far apart in the document. In practice, the effort required in collapsing and expanding document parts in elision interfaces may outweigh the benefits of faster navigation in the document (Cockburn and Smith, 2003). Finally, automatically changing the view based on changes in the user's focus may confuse or disorient users (paper 1 and 4) whereas a text representation that users can manually change is predictable and thus less disorienting.

In summary, our studies contribute empirical evidence in support of fisheye interfaces' usefulness in programming. Rich data were obtained by using multiple research methods. The data show how programmers use fisheye interfaces effectively to navigate and understand source code, and how a fisheye interface could be used across different activities in programmers' own work. However, the results are limited by methodological issues and also by investigating only particular designs of fisheye interfaces. Although alternative designs were initially explored—informed by my own experiences as a programmer and by informal evaluations with others—only one line of iterative refinement of fisheye interface was investigated, which arguably limits the potential outcome (Greenberg and Buxton, 2008). Furthermore, the fisheye interfaces studied here provide context to the user's focus only within a single source code file. Fisheye interfaces may provide context across the entire code base (Storey et al., 2000; Kersten and Murphy, 2006) and use diverse task information to establish the user's focus, such as previous navigation activity (DeLine et al., 2005) or task descriptions (Lawrance et al., 2008).

TRANSIENT VISUALIZATIONS

The fisheye interface investigated in paper 1 makes permanent changes to the source code view with the aim of supporting navigation and understanding of source code. However, programmers work on a diversity of tasks that may not all be equally well supported by the fisheye interface: for instance, programmers did not find it useful for debugging or writing new code (paper 4). This issue is a general one: visualizations often make permanent changes to the user interface with the aim of supporting specific tasks. However, many applications

are designed to support a variety of tasks in complex work settings, and changes made to an interface to improve its use in some tasks may have adverse effects on its use in other tasks.

We argue that transient use of visualizations may support specific infrequent tasks without permanently changing the interface. Earlier research has lead to similar ideas (Baudisch et al. 2004), and several lightweight interaction techniques that use transiency have been researched (e.g., Fekete and Plaisant, 1999; Zellweger et al., 2000; Baudisch et al., 2003; Bezerianos and Balakrishnan, 2005). To provide a basis for generalizing about transient use of information visualization, we compared transient and permanent use of visualization in two experiments (paper 2 and 3). Three notable findings were made that relate to the characteristics of transient visualizations described in paper 2.

First, we found that for particular navigation tasks, users' *immediate* and *close* access to a transient map overview can reduce sensory-motor efforts, as indicated by less mouse movement and perceived user effort, and lead to higher satisfaction and preference compared with a permanent overview (paper 2). Importance of closeness is also demonstrated by for instance Drag-and-pop (Baudisch et al., 2003) and Vacuum (Bezerianos and Balakrishnan, 2005): both techniques have been shown to enable quick access to remote objects in the display by bringing them closer. Minimal physical movement was one design principle underlying the design of Vacuum (Bezerianos and Balakrishnan, 2005).

Second, it seems a transient visualization may support a specific task more effectively by allowing users to call up a tailored representation of only the types of information useful in that specific task. That is, the *relatedness* of the information to the user's current focus of attention may affect the direct usefulness of the visualization for the specific task. In paper 3 we compared a transient fisheye view and a permanent fisheye view that were based on the same degree-of-interest function. Instead, it might be more effective if the user can call up different transient fisheye views that each provides only the information pertinent to the user's specific focus of attention in a particular task.

Third, the *transiency* or briefness of a particular transient visualization must match the briefness of the user's interaction with information needed for the supported tasks. Some participants found that the transient fisheye interface disappeared too easily—seemingly, it disappeared before participants were finished using the information provided in the fisheye interface (paper 3). In programming, users might benefit from being able to alternate between a fisheye view and a plain view as appropriate for the task at hand. In contrast, a transient representation can be appropriate for tasks that require only brief use of the information (e.g., paper 2; Fekete and Plaisant, 1999).

The two empirical studies in paper 2 and 3 provide a basis for elaborating on the characteristics of transient visualizations. However, the comparative evaluation of only two specific applications of transient visualization with counterpart permanent visualizations is not enough to reliably conclude about benefits of transient visualizations in general. More research is clearly needed. One question to consider in future research is how a transient visualization tailored for use in a specific task can be meaningfully compared with a permanent visualization that must support varied work activity.

VISUALIZATIONS TO SUPPORT TEAM COORDINATION

Developing a large system is too much work for one programmer. Fortunately, work can often be partitioned among the programmers in a team. Doing so, however, requires time consuming communication between team members, for instance to coordinate work (Brooks, 1975; Ko et al., 2007). One reason that coordination requires communication is that visibility of the work of individual team members, which is useful for maintaining awareness of what is going on, is lacking.

In paper 5, we aimed to support awareness of a team's work through visualization of work items. Work items are descriptions of work tasks partitioned so that individual team members can carry them out (Knudsen et. al, 1976). However, the systems used to manage work items are not designed to provide an overview of the work items or the ongoing changes made to work items. A visualization called WIPDash was designed to address these needs.

WIPDash was deployed with two collocated teams, running on a large display in each team's room and on individual team members' workstations. The teams were observed during one week. No effect was found of the visualization in measures of situational awareness and group satisfaction, but observations, logged activity data, and discussions with the teams gave insight into the teams' use of the visualization.

Based on comments from participants (paper 5) and from earlier research, it seems that programmer teams appreciate a combined view of the entire shared workspace and changes in it (Biehl et al., 2007). Our results suggest two types of use of such visualizations in a programmer team environment. First, teams can use the visualization during meetings. In our study, one team used WIPDash on the large display during daily standup meetings, and they would like a view that is tailored to those meetings (paper 5). Whereas visualizations aimed at individual users have long been researched (e.g., Eick et al., 1992; Froehlich and Dourish, 2004; Ellis et al., 2007), little is known about how visualization can support collaboration in teams, for instance during meetings as seen in our study. Interest is gaining, however, in the collaborative use of visualization (e.g., the CoVIS'09 workshop).

Second, team members may use the visualization to maintain awareness of the team's work by glancing at the display at natural breakpoints in their task (Biehl et al., 2007) or when entering or leaving the room (paper 5). One problem that detracted from WIPDash's usefulness for such occasional glancing seems related to the automatically cycling between different views in the visualization. An aim of the cycling was to limit the amount of information in the display, but in practice it was found that nothing of interest was shown in parts of the cycle. Participants suggested cycling only between views that contain recent changes, but alternative approaches may be needed to make the display convey an appropriate amount of information while being consistent. Design and evaluation of this type of display, often called peripheral display in the literature, is complicated by inconsistent use of terminology, and frameworks and guidelines have only recently started to emerge (e.g., Mankoff et al., 2003; Pousman and Stasko, 2006; Matthews et al., 2007).

In general, improving awareness has been the aim of much research (Schmidt, 2002). In programming, tools for helping programmers maintain awareness of team members' activity in the source code have been found useful for identifying and resolving conflicts (e.g., Biehl et al., 2007; Sarma et al., 2008). In contrast, WIPDash shows a higher-level view of a team's work based on the team's repository of work items. Grinter (1995) found such higher-level views in demand but lacking in code-centric tools. An interesting perspective for future research is to combine information about work items with information about code activity in one visualization. Finally, WIPDash was aimed at coordination in collocated teams, but visualization of work items may also be useful to help maintain awareness of work in distributed teams (Gutwin et al., 2004) and work of other teams (Begel et al., 2009).

EVALUATION OF VISUALIZATION TECHNIQUES IN COMPLEX WORK

Demonstrating the utility of an information visualization system or technique has been argued to be crucial to the adoption of the system or technique (Plaisant, 1995). To that end, laboratory experiments are weakened by their lack of realism, which long-term case studies in contrast are rich in. Yet, understanding the effects of particular design factors of a visualization technique may further development of the technique and eventually lead to its adoption and use in practice. To that end, laboratory experiments are arguably strong where case studies fail in producing precise results about the effects of individual factors in the design of a visualization (Lam and Munzner, 2008).

All research methods have weaknesses or limitations, but the limitations of different methods can be offset by using multiple methods (McGrath, 1995). We researched fisheye interfaces in programming using (1) two different strategies, laboratory experiment and field study, and (2) multiple data collection methods, or types of measures, in combination. The research allowed us to assess the methods' use, individually and in combination, for evaluating fisheye interfaces in programming.

First, the laboratory experiments provided basis for understanding participants' interaction with the fisheye interfaces and how the interfaces affect participants' performance in representative tasks (paper 1 and 3). Data from the field study did not facilitate such analysis, but allowed us to learn whether participants adopted the fisheye interface in real work, how the interface was used across a variety of tasks, and which types of information in the fisheye interface that were used (paper 4). The field study offset at least two limitations of the laboratory experiments: the tools available in the programming environment were controlled in the experiments, whereas participants in the field study could use any tool available in their programming environment; and only particular types of tasks were used in the experiments, whereas the field study investigates participants doing their own work tasks.

Second, the multiple data collection methods combined in the field study provided stronger evidence of adoption and use of the fisheye interface (paper 4). For instance, thinking aloud provided concrete situations of use that showed diverse uses of the fisheye interface, but it is hard to generalize about how participants used the fisheye interface in their work from the 55 observed incidents of use. In contrast, the 370 hours of activity logging showed overall patterns in how the fisheye interface was used throughout participants work, but it is hard to establish participants' intent from the activity data.

Altogether, the multiple methods provided us with richer data for understanding how fisheye interfaces are used in working with source code, for uncovering usability issues with the interfaces, and for further developing the theoretical foundations of fisheye views. However, some issues limit the conclusions that can be drawn from the results.

Learning is important to consider when interpreting our results. Participants in the experiments might not have had sufficient time to gain proficiency in use of the interfaces (paper 1 and 3). Consequently, the effects of the fisheye interfaces on participants' behavior and performance measured in experiments may not reflect effects that would be found in practice after longer time of use. Participants in the field study used the fisheye interface for weeks, but still did not consider themselves proficient (paper 4). Beyond initial learning of the interface, participants may need long time to gain experience with using the interface across work tasks. Moreover, it seems natural that people are focused on getting work done, not on learning how to work more effectively (Carroll and Rosson, 1987). Researchers and designers are thus challenged to improve the learnability of new interfaces. In the present research, it stresses the need for stimulating participants to use of the interface in their work in new and varied ways, some of which may develop into habits. Perhaps such learning can be impelled by the researcher's active involvement in participants' work in longer-term studies, as proposed by Shneiderman and Plaisant (2006). In any case, the field study approach does not allow us to precisely determine what effects the interface has on participants' productivity, however proficient participants become.

Further, limitations in our approach to method triangulation in the field study present opportunities for future research. First, to better understand how participants' intent relates to their actions, data from probes could be coupled more strongly to activity logging, for instance by probing participants conditionally based on their activity. We used conditional probes in a simple way, but the potential of this approach calls for further exploration. Second, to allow stronger extrapolation of the observed situations of use, thinking aloud recordings may be better coupled to quantitative activity data. Alternatively, a broader and more representative sample of participants' work may be collected either by using long-term screen recordings (Tang et al., 2006) or recordings that are triggered by activities (Akers et al., 2009).

Although participants adopted the fisheye interface and made comments that suggested that they would continue to use it after the field study, we did not investigate whether they did continue to use it. The field study reduces the reactivity of measures compared with the laboratory experiments, but does not eliminate it entirely (McGrath, 1995). Investigating whether participants use the fisheye interface after longer time using unobtrusive measures could help demonstrate the utility of fisheye interfaces more convincingly.

Finally, integrating a fisheye interface a code editor so that it is suitable and stable enough for real-life programming required extensive work and still participants experienced

problems. Earlier research has pointed out that visualization systems and tools must be stable and integrate with existing work practices (Gonzales and Kobsa, 2003; Shneiderman and Plaisant, 2006). But compared with standalone visualization systems, the difficulties of properly integrating a visualization technique in an existing tool present a potential barrier to conducting long-term studies. However, another approach challenges researchers to persuade participants to use a new tool that implements the visualization technique in question, instead of a tool they may already be proficient in using.

In all, our research demonstrates benefits and limitations of using multiple methods to evaluate fisheye interfaces in programming. Combined, the laboratory experiments and the field study provided more convincing evidence that fisheye interfaces are useful to programmers. Perhaps more important, however, the multiple methods provided richer data for better understanding how programmers use fisheye interfaces for navigating and understanding source code—understanding which characteristics of a visualization technique that contribute to improving users' work may be crucial to the further development of the technique and to its eventual adoption and use in practice. Our results indicate challenges for future work in evaluating visualization techniques in complex work. For instance, a key barrier to assessing a novel technique's potential is that participants may not learn and adopt the technique across tasks in their work. Opportunities for combining data from multiple methods have been suggested that may inspire future research in evaluation of visualization techniques in complex work.

CONCLUSION

The research in this thesis suggests that information visualization can improve programmers' tools and environments. Looking ahead, I think several problems and opportunities warrant further investigation. We need to better understand the characteristics of fisheye interfaces that contribute to enhancing programmers' abilities and how those characteristics can be more widely utilized in programming. Also, there are problems specific to the integration of fisheye interfaces in source code editors: tools that have many diverse uses may not be easily replaced and we need to better understand how visualizations may extend such tools without adverse effects. Further, questions fundamental to the design of fisheye interfaces still need answering: How can we determine degree of interest? How can we distort the view to fit all interesting information while also supporting effective interactions that feel natural to users? Lack of design guidelines is a barrier to wider adoption of fisheye interfaces. Finally, there is arguably a need for novel methods for evaluating visualization techniques. My hope is that the use of methods triangulation presented in this thesis will be replicated and extended in further research.

REFERENCES

Akers, D., Simpson, M., Jeffries, R., and Winograd, T. (2009). Undo and erase events as indicators of usability problems. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, ACM, 659–668.

Baudisch, P., Cutrell, E., Robbins, D., Czerwinski, M., Tandler, P., Bederson, B., and Zierlinger, A. (2003). Drag-and-Pop and Drag-and-Pick: Techniques for Accessing Remote Screen Content on Touch and Pen-operated Systems. In *Proc Interact'03*, 57–64.

Baudisch, P., Lee, B., and Hanna, L. (2004). Fishnet, a fisheye web browser with search term popouts: a comparative evaluation with overview and linear view. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, ACM Press, 133–140.

Begel, A., Nagappan, C. P. N. and Layman, L. (2009). Coordination in Large-Scale Software Teams. In *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, IEEE Computer Society, Washington, DC.

Bezerianos, A. and Balakrishnan, R. (2005). The vacuum: facilitating the manipulation of distant objects. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press, 361–370.

Biehl, J. T., Czerwinski, M., Smith, G., and Robertson, G. G. (2007). FASTDash: a visual dashboard for fostering awareness in software teams. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, 1313–1322.

Brooks, Jr., F. P. (1975). *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison Wesley.

Card, S. K., Mackinlay, J. D., and Shneiderman, B. (1999). *Readings In Information Visualization: Using Vision To Think*. Academic Press.

Carroll, J. M., & Rosson, M. B. (1987). The paradox of the active user. In J.M. Carroll (Ed.), *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*. Cambridge, Mass: MIT Press, 80-111.

Cockburn, A. and McKenzie, B. (2002). Evaluating the effectiveness of spatial memory in 2D and 3D physical and virtual environments. In *CHI '02: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, 203–210.

Cockburn, A. and Smith, M. (2003). Hidden messages: evaluating the efficiency of code elision in program navigation. *Interacting with Computers, 15*, 3, 387-407.

CoVIS'09 (2009). Workshop on Collaborative Visualization on Interactive Surfaces, In conjunction with VisWeek 2009 (Vis, InfoVis, VAST), October 11, Atlantic City, New Jersey, http://www.medien.ifi.lmu.de/covis09

Debian.org (2009). A Brief History of Debian: Debian Releases. Retrieved on 2009-07-21. http://www.debian.org/doc/manuals/project-history/ch-releases.en.html

DeLine, R., Czerwinski, M., and Robertson, G. (2005). Easing Program Comprehension by Sharing Navigation Data. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, IEEE Computer Society, 241–248.

Dunsmore, A., Roper, M., and Wood, M. (2000). Object-oriented inspection in the face of delocalisation. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, ACM Press, 467–476.

Eick, S. G., Steffen, J. L., and Sumner, E. E. (1992). SeeSoft - A Tool for Visualizing Line Oriented Statistics Software. *IEEE Transactions on Software Engineering, 18*, 957-968.

Ellis, J. B., Wahid, S., Danis, C., and Kellogg, W. A. (2007). Task and social visualization in software development: evaluation of a prototype. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, 577–586.

Fekete, J. and Plaisant, C. (1999). Excentric labeling: dynamic neighborhood labeling for data visualization. In CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems, ACM Press, 512–519.

Francel, M. A. and Rugaber, S. (1999). The Relationship of Slicing and Debugging to Program Understanding. In IWPC '99: *Proceedings of the 7th International Workshop on Program Comprehension*, IEEE Computer Society, 106-113.

Froehlich, J. and Dourish, P. (2004). Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, IEEE Computer Society, 387–396.

Furnas, G. W. (1981). The fisheye view: A new look at structured files. Bell Laboratories Technical Memorandum #81-11221-9, October 12.

Gonzáles, V. and Kobsa, A. (2003). A Workplace Study of the Adoption of Information Visualization Systems. In *Proceedings of I-KNOW'03: 3rd International Conference on Knowledge Management*, 92-102.

Gonzalez-Barahona, J. M., Robles, G., Michlmayr, M., Amor, J. J., and German, D. M. (2009). Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14, 3, 262-285.

Greenberg, S. and Buxton, B. (2008). Usability evaluation considered harmful (some of the time). In *CHI '08: Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, ACM, 111–120.

Grinter, R. E. (1995). Using a configuration management tool to coordinate software development. In *COCS '95: Proceedings of conference on Organizational computing systems*, ACM, 168–177.

Gutwin, C., Penner, R., and Schneider, K. (2004). Group awareness in distributed software development. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, ACM, 72–81.

Hornbæk, K. and Frøkjær, E. (2003). Reading patterns and usability in visualizations of electronic documents. *ACM Transactions on Computer-Human Interaction, 10*, 2, 119–149.

Kersten, M. and Murphy, G. C. (2006). Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, ACM, 1–11.

Knudsen, D. B., Barofsky, A., and Satz, L. R. (1976). A Modification Request Control System. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, IEEE Computer Society Press, 187–192.

Ko, A. J., DeLine, R., and Venolia, G. (2007). Information Needs in Collocated Software Development Teams. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, IEEE Computer Society, 344–353.

Kraut, R. E. and Streeter, L. A. (1995). Coordination in software development. *Commun. ACM, 38*, 3, 69–81.

Lam, H. and Munzner, T. (2008). Increasing the utility of quantitative empirical studies for meta-analysis. In *BELIV '08: Proceedings of the 2008 conference on BEyond time and errors*, ACM.

LaToza, T. D., Venolia, G., and DeLine, R. (2006). Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, ACM Press, 492–501.

Lawrance, J., Bellamy, R., Burnett, M., and Rector, K. (2008). Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *CHI '08: Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, ACM, 1323–1332.

McGrath, J. E. (1995). Methodology matters: doing research in the behavioral and social sciences. In *Human-computer interaction: toward the year 2000*, Morgan Kaufmann Publishers Inc., 152–169.

Plaisant, C. (2004). The challenge of information visualization evaluation. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, ACM Press, 109–116.

Saraiya, P., North, C., and Duca, K. (2005). An Insight-Based Methodology for Evaluating Bioinformatics Visualizations. *IEEE Transactions on Visualization and Computer Graphics, 11*, 4, 443-456.

Seo, J. and Shneiderman, B. (2006). Knowledge Discovery in High-Dimensional Data: Case Studies and a User Survey for the Rank-by-Feature Framework. *IEEE Transactions on Visualization and Computer Graphics, 12*, 3, 311-322.

Shneiderman, B. and Plaisant, C. (2006). Strategies for evaluating information visualization tools: multi-dimensional in-depth long-term case studies. In *BELIV '06: Proceedings of the 2006 conference on BEyond time and errors*, ACM Press.

Soloway, E., Lampert, R., Letovsky, S., Littman, D., and Pinto, J. (1988). Designing documentation to compensate for delocalized plans. *Communications of the ACM, 31*, 11, 1259–1267.

Storey, M. A. D., Wong, K., and Müller, H. A. (2000). How do program understanding tools affect how programmers understand programs? *Sci. Comput. Program., 36*, 2-3, 183–207.

Tang, J. C., Liu, S. B., Muller, M., Lin, J., and Drews, C. (2006). Unobtrusive but invasive: using screen recording to collect field data on computer-mediated interaction. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, ACM, 479–482.

Weinberg, G. M. (1971). *The psychology of computer programming*. New York, NY, USA: Van Nostrand Reinhold Co.

Weiser, M. and Lyle, J. (1986). Experiments on slicing-based debugging aids. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, Ablex Publishing Corp., 187–197.

Zellweger, P. T., Regli, S. H., Mackinlay, J. D., and Chang, B. (2000) The impact of fluid documents on reading and browsing: an observational study. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press, 249–256.

PAPER 1 – EVALUATING A FISHEYE VIEW OF SOURCE CODE

Jakobsen, M. R. and Hornbæk, K. (2006). Evaluating a fisheye view of source code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada, April 22 - 27, 2006). CHI '06. ACM, New York, NY, 377-386.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CHI 2006, April 22-27, 2006, Montréal, Québec, Canada.
Copyright 2006 ACM 1-59593-178-3/06/0004. . . $5.00.

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].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CHI 2006, April 22-27, 2006, Montréal, Québec, Canada.
Copyright 2006 ACM 1-59593-178-3/06/0004. . . $5.00.

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

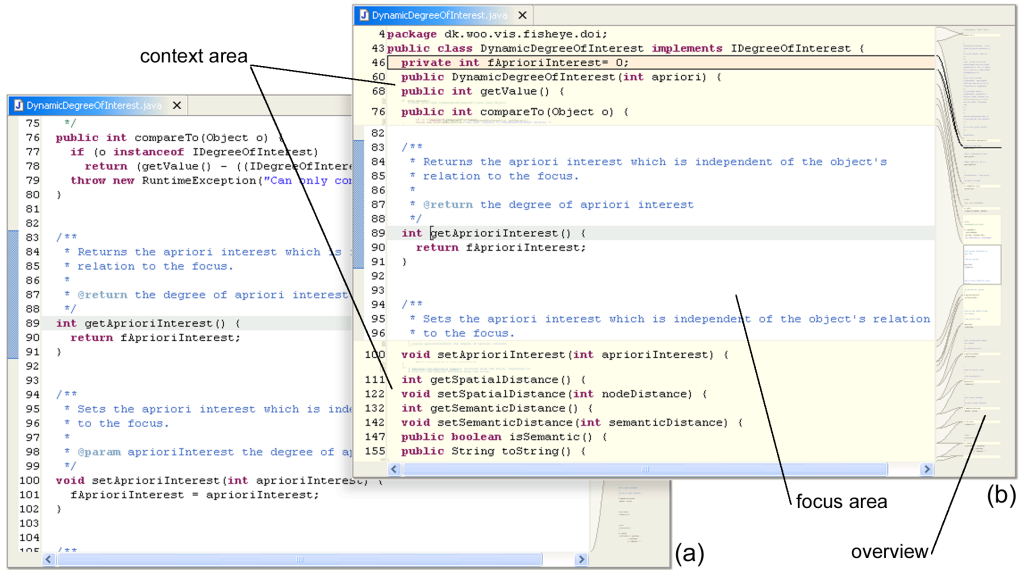


Figure 1. Screenshots of (a) the Linear and (b) the Fisheye interface showing the same source file of 161 lines.

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,

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.

but rather that it was confusing when searching by scrolling in the source code.

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.

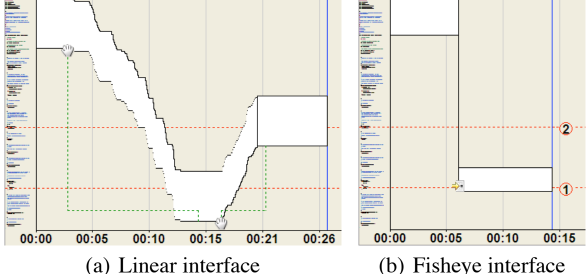
The focus area in the Fisheye interface was too small according to comments made by 12 out of the 16 participants. A few participants added that they would find this a problem when writing or editing the code.

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

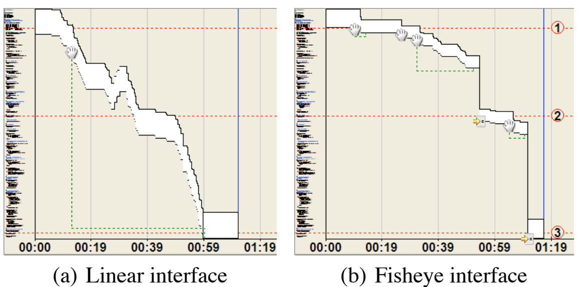
Figure 4. Progression maps, two-step-navigation tasks.

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.

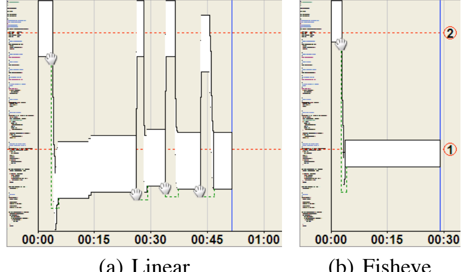
Typical interaction patterns can be seen in the representative progression maps for determine-delocalization tasks in Figure 6 (involving variables) and Figure 7 (involving methods). The progression maps confirm that participants made several searches and jumps in the source code with the Linear interface. 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.

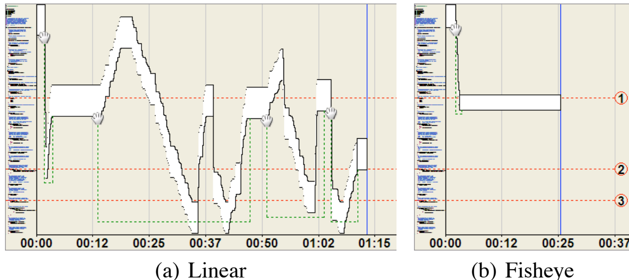
ments enclosing a given line. In the first task, all participants using the Linear interface scrolled down to the specified line, and were then able to answer the task without scrolling further, Figure 8(a). Six out of eight participants using the Fisheye interface continued to scroll the focus area 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

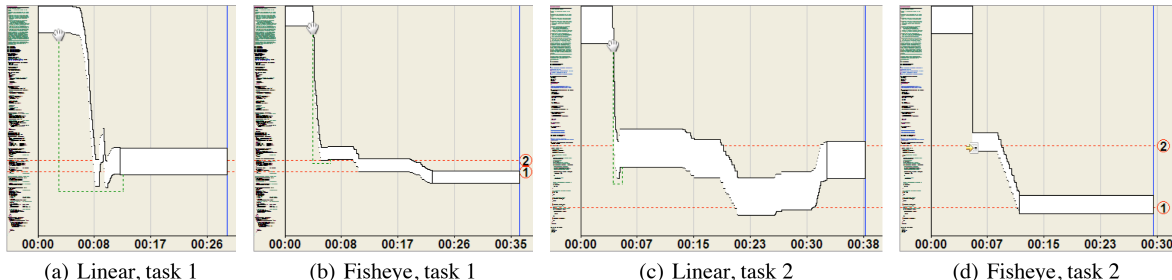
The results from our experiment show an overall improvement in task completion times with the fisheye interface for representative program navigation and understanding tasks. Yet, strong differences in task completion times were found among tasks. Participants were equally accurate in answering the tasks. 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.

PAPER 2 – TRANSIENT VISUALIZATIONS

Jakobsen, M. R. and Hornbæk, K. (2007). Transient visualizations. In *Proceedings of the 19th Australasian Conference on Computer-Human interaction: Entertaining User interfaces* (Adelaide, Australia, November 28 - 30, 2007). OZCHI '07, vol. 251. ACM, New York, NY, 69-76.

OzCHI 2007, 28-30 November 2007, Adelaide, Australia. Copyright the author(s) and CHISIG. Additional copies are available at the ACM Digital Library (http://portal.acm.org/dl.cfm) or can be ordered from CHISIG(secretary@chisig.org)

OzCHI 2007 Proceedings, ISBN 978-1-59593-872-5

Transient Visualizations

Mikkel Rønne Jakobsen
Department of Computer Science
University of Copenhagen
Universitetsparken 1, DK-2100, Denmark
+45 35321451
mikkelrj@diku.dk

Kasper Hornbæk
Department of Computer Science
University of Copenhagen
Universitetsparken 1, DK-2100, Denmark
+45 35321425
kash@diku.dk

ABSTRACT

Information visualizations often make permanent changes to the user interface with the aim of supporting specific tasks. However, a permanent visualization cannot support the variety of tasks found in realistic work settings equally well. We explore interaction techniques that transiently visualize information near the user's focus of attention. Transient visualizations support specific contexts of use without permanently changing the user interface, and aim to seamlessly integrate with existing tools and to decrease distraction. Examples of transient visualizations for document search, map zoom-outs, fisheye views of source code, and thesaurus access are presented. We provide an initial validation of transient visualizations by comparing a transient overview for maps to a permanent visualization. Among 20 users of these visualizations, all but four preferred the transient visualization. However, differences in time and error rates were insignificant. On this background, we discuss the potential of transient visualizations and future directions.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation (e.g., HCI)**]: User Interfaces, I.3.6 [**Methodology and Techniques**]: Interaction Techniques

General Terms

Design, Human Factors

Keywords

Interaction techniques, visualization, transient, lightweight, fluid, overview+detail, fisheye

1. INTRODUCTION

Many information visualizations make permanent changes to the way the visual structure of information appears in the user interface. Different mechanisms are used toward this change, such as transforming the visual structure, adding features to the visual structure, and using multiple views. For example, Fishnet [1] permanently applies a bifocal display transformation and adds search-term popouts to the visual structure of a web page, and Popout Prism [15] permanently adds a zoomed-out overview to the detail view of a web page.

Designing permanent visualizations that are suitable for realistic work environments is complicated by the diversity of tasks that need to be supported. For example, consider a fisheye view of source code that presents context information relevant to the current focus. Such as view may support navigation and understanding, but the same fisheye view is inappropriate for writing and editing code because programmers want a large view of source code for those tasks [11]. Based on the observation that a particular design of a permanent visualization may be suitable only in some scenarios, Baudisch et al. [1] recommended that users should be allowed to bring up visualizations on demand, for example by using a keyboard shortcut.

We discuss transient visualizations, interaction techniques for transient use of information visualizations close to the user's focus of attention. Many user interfaces successfully employ techniques that provide users with transient information in the context of their focus of attention, including tool tips and context menus. Also, the HCI literature presents numerous techniques that involve transiency, lightweight interaction, and visualization [e.g., 4,9,10]. However, we are unaware of any attempts at generalizing about using information visuali-zations transiently. Therefore, the general benefits of transient visualizations and the factors that advance and restrict their use are unclear. In this paper, we present examples to probe potential benefits of transient visualizations, and report an initial validation of one instance of a transient visualization.

Contributions of our work are (a) to direct researchers' awareness toward transient uses of information visualizations that may help avoid problems inherent in the design of permanent visualization interfaces, (b) to provide a basis for practitioners to consider how transient visualizations may be utilized in the work practices they seek to support, and (c) to present encouraging initial data about the usability of transient visualizations.

2. CHARACTERISTICS OF TRANSIENT VISUALIZATIONS

Transient visualizations have four characteristics:

- Immediacy; to bring the user into direct and instant involvement with the information representation.
- Transiency; information is only displayed tempo-rarily, and is easily dismissed, which means that no display space is used permanently.
- Closeness; the information is shown close to the region of focus in the display (e.g., cursor or caret), resulting in fast access to the information because of minimized sensory-motor efforts of the user.
- Contextuality; the information is related to the user's current focus of attention, for example by adding context for interpreting the information in focus.

OzCHI 2007, 28-30 November 2007, Adelaide, Australia. Copyright the author(s) and CHISIG. Additional copies are available at the ACM Digital Library (http://portal.acm.org/dl.cfm) or can be ordered from CHISIG(secretary@chisig.org)

OzCHI 2007 Proceedings, ISBN 978-1-59593-872-5

We contrast transient visualizations with permanent information visualization interfaces, such as overview+detail interfaces where permanent display space is allocated to an overview window [15]. First, designers are challenged with deciding what information is needed in various contexts of use and fitting the information into the limited display space of a permanent visualization. In contrast, using transient visualizations to facilitate infrequent and unpredictable contexts of use, the original permanent view can be dedicated to information used in frequent contexts of use.

Second, adopting permanent visualizations to improve an existing tool may break established uses of the tool. However, the means of invoking and interacting with transient visualizations can be tailored to particular contexts of use, thus supplementing established interaction habits.

Finally, rich and dynamic views in permanent visualizations may visually disorient and annoy the user. In contrast, using transient visualizations that appear only temporarily and under the user's control helps prevent visually complex and disorienting interfaces.

3. EXAMPLE APPLICATIONS

To provide concrete arguments for the potential of the idea of transient visualizations, we present sketches of transient visualizations that support tasks in three different domains, and describe a prototype of a transient visualization in a programming environment.

3.1 Searching in Documents

In conventional web browsers, 'Find' automatically jumps to the first instance of words as they are typed. However, scrolling between found instances may disorient the user [15]. Recent studies have shown overview+detail visualizations [15] and bifocal displays [1] to be efficient and preferred by users for searching in documents. Among the advantages experienced by participants using an overview+detail interface, Suh et al. [15] report that the interface gives "a sense of context, density, and the ability to see all occurrences of a keyword at once" and provides orientation support for navigating in the document.

In the design mock-up in Figure 1, we show a transient visualization to support in-document search; our approach extends a conventional browser window by calling up a thumbnail overview when the user invokes the 'Find'-bar. As the user begins to type keywords, instances of the words are highlighted in the overview. The user can move between highlighted words using the keyboard, or drag the field-of-view window using the mouse. Behind the overview, the original view scrolls the document accordingly, visually coupling the overview to the original view. Finally, the overview can be dismissed to scroll back to the original location in the document by suspending the 'Find' action (e.g., with the Escape-key).

We believe that our suggested design provides the same support for in-document search as a permanent overview by giving an overall sense of the location, density and co-occurrences of keywords. Additionally, in contrast to a permanent overview, (1) the overview does not compete for permanent display space, and (2) fluid keyboard interaction allows the user to complete their task without having to switch to mouse. In reading tasks, Hornbæk and Frøkjær [8] found overviews to support navigation and help to get a structural overview of the document, yet the overview may also distract the user. A study by Nekrasovski et al.

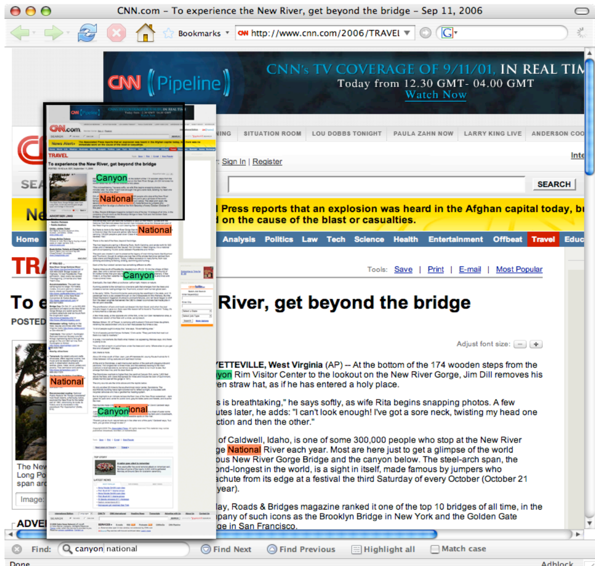


Figure 1: Sketch of transient overview of document with popout instances of words entered in the 'Find' bar.

[13] showed no performance effects of overviews used for navigating large hierarchical trees, but participants perceived them as beneficial. These results indicate that a transient overview may support particular uses such as searching or providing structural overview for navigation with less risk of distracting the user compared with a permanent overview.

3.2 Planning and Navigating Routes in Maps

Viewing and interacting with maps has received much attention in HCI research, and have been addressed by different visualization approaches including panning and zooming, overviews, and distortion techniques such as fisheye views [7]. A common use of maps is for planning and navigating a route to a destination. When navigating toward a remote destination, travelers commonly use a detailed map to orient themselves at their current location. However, an overview of the route to the destination may occasionally be needed to support a sense of direction and awareness in travels ahead. Getting an overview using conventional map applications may require considerable zooming in and out and panning the map to find road names and landmarks on the route ahead.

The mock-up presented in Figure 2 shows a way to extend a conventional map view with a transient visualization to address this problem. The user invokes the visualization by clicking on the route, calling up a map of a higher scale, thus showing the route farther toward the destination. In Figure 2, the user has further clicked three times on the route, to call up maps of continually higher scales, until the complete route is revealed. Finally, the visualization can be dismissed by clicking on the original map. The route provides fixing points for "stitching together" the maps of different scales, and the selected route can also be used to deduce contextual information, such as road names along the route that should be highlighted.

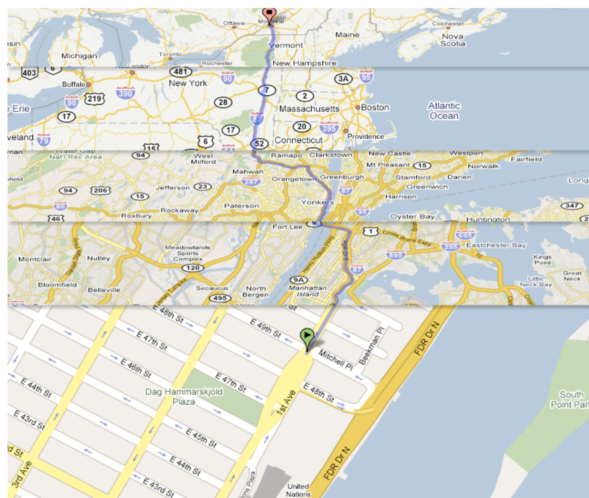


Figure 2: Route visualization where transient zoom-outs at progressively higher scales of a map have been called up by clicking repeatedly on the route to show the way to the destination.

This example shows how to extend the design space of information visualizations to transient use in a particular context, where permanent visualization techniques do not seem useful. Fisheye interfaces that geometrically distort maps are useful only to a limited degree of magnification and the distortion may inhibit users from recognizing shapes of roads and locations of landmarks [18]. Overview+detail interfaces may give an overview of the route, but to discern landmarks and road names along the route the user has to move the detail view. Zooming interfaces require the user to pan or continually zoom out to get an overview of the route, and then zoom in to see details of the route. In summary, while these different techniques may be useful for frequent contexts of use, it may be worthwhile to pursue transient visualizations for particular tasks such as the focus of this example.

3.3 Programming

Programming is a complex human activity that information visualizations potentially can support [11]. However, as mentioned earlier, applying a permanent visualization to source code can be complicated. Figure 3 shows a transient fisheye view of source code implemented as a plug-in for the Eclipse Java IDE. The visualization is invoked using a keyboard shortcut; popup views then appear above and below the editor window. The views contain lines with references to the variable that the user has currently selected with the caret. Arrow keys are used to select a line and pressing 'Enter' centers the view on the selected line. The visualization can be dismissed with the 'Escape' key.

Our design aims to support source code navigation and program understanding by providing lightweight access to contextual information relevant to the current focus in the source code. Compared with a permanent fisheye view, our design allows a large view of source code that programmers seem to prefer for writing and editing code. Furthermore, we aim to support fluid interaction with the transient fisheye view in programming by extending existing uses of the keyboard. A recent user study of programmers has shown extensive use of keyboard shortcuts for navigating in source code [13], and transient views showing outlines and type hierarchies are familiar in common programming environments such as Eclipse. We thus believe programmers may easily adopt transient visualizations that are invoked using keyboard shortcuts.

3.4 Writing

A very common task in writing is to find the right word at some point in a sentence. A thesaurus can be particularly effective for this task when writing in a language different from your mother tongue. In many word processors, finding the right word involves selecting a word, looking it up in the dictionary or thesaurus, browsing the definition and navigating links to synonyms.

Figure 4 shows a mock-up of a transient thesaurus visualization overlapping a text that the user is editing. The visualization is called up with a keyboard command to show words that are related to the word at the caret position. The user can interact with the visualization to explore more synonyms of a particular meaning; the highlight box can be moved with the cursor keys or mouse to one of the connected words, and selecting a word animates the visualization to center around that word, thus revealing more synonyms of that word. Also, the user can call up a window with the definition of a selected word. Finally, the visualization can be dismissed either to replace the original word in the sentence with the selected word (e.g., by hitting Enter) or without making changes to the text (by hitting Escape).

Our design utilizes the hierarchical organization of words in a thesaurus. In contrast to a linear textual representation, users can visually perceive from the visualization how synonyms of a word are grouped by similar meanings. Also, synonyms are presented close to the word and its surrounding text so that users can imagine how other words fit into the text. Finally, by making the visualization easy to invoke and dismiss, we aim for the use of the transient thesaurus to become an effortless part of writing.

4. RELATED WORK

We have aimed to demonstrate alternative uses of information visualizations by extracting and refining ideas from previous

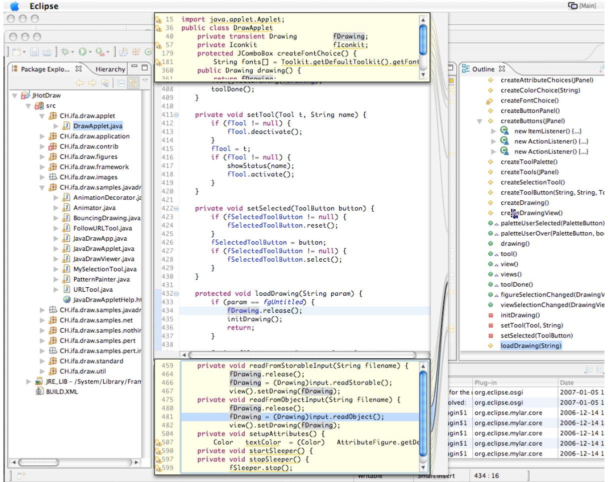


Figure 3: Prototype of transient fisheye view of source code that shows context information in popup windows above and below the editor window.

Figure 4: Transient thesaurus called up to show synonyms for the word "fresh" in a word processing application.

work. This section overviews such related work in HCI research that use transient representations of information and light-weight interaction.

Excentric labeling provides labels for a neighborhood of objects located around the cursor [5]. 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 [16]. Zellweger et al. [19] 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 centered around the cursor and with access to additional menus via mouse gestures [12]. See-through tools are another technique that provides close and contextual access to commands without requiring permanent display space [4].

Many information visualizations use brushing to highlight (or affect) instances in other views of an object that the user brushes over [2]. 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 [9]. These techniques provide immediate and non-intrusive visualizations through lightweight interaction.

Large and small displays accentuate problems in human-computer interaction, which have prompted HCI research to generate novel interaction techniques to temporarily bring objects that are otherwise hard to interact with closer to the user. The interaction technique called Vacuum helps reach remote objects through proxies that are transiently placed close to the cursor for easy manipulation, reducing the physical demands of the user [3]. Similar challenges in small displays have brought about techniques to visualize and navigate to off-screen targets with halos and proxies [10].

5. EXPERIMENT

To provide initial data about the usability of transient visualizations, we conducted a study comparing two interfaces for viewing maps, shown in Figure 5. Both interfaces include a view that can be panned to show different parts of a map; the user clicks and drags the mouse opposite the panning direction (i.e., the map follows the mouse). The interfaces also contain a semitransparent overview of the entire map. The overview partly covers the detail view so that it is possible but hard to discern map details in the detail view under the overview. However, it is not possible to "click through" the overview to interact with map details. Interaction with the overview differs between the interfaces. In the *Permanent interface (PI)*, the overview is permanently shown in the upper right corner of the detail view. The user can click and hold the left mouse button to drag a field-of-view box in the overview in order to pan the detail view. In contrast, the *Transient interface (TI)* does not permanently show an overview, but a transient overview can be invoked at the location of the mouse cursor by pressing and holding down the right mouse button; the overview appears so that the mouse cursor's location in the field-of-view box corresponds to the cursor's location in the detail view. Moving the mouse pans the detail view, and the overview disappears when the mouse button is released. Our primary goal is to compare the Permanent interface and the Transient interface. Therefore, we do not aim for our study to be realistic, but try to tease out differences in how users interact with the two interfaces.

5.1 Participants

20 students (4 female) at the authors' department participated in the experiment. The participants were between 21 and 50 years old (M = 29.3, SD = 7.9).

5.2 Tasks

Two types of task were used in the experiment. Both tasks involve maps of randomly placed circles with random names and randomly connecting lines. Maps are generated to resemble social networks. Colored circles are randomly scattered in the map, requiring participants to move the detail view to see them.

The first task type involves *selection* of 10 red circles in the map by finding and clicking on them. The selection task is designed to make participants alternate between navigating and interacting with objects in the map. Our hypothesis is that participants are slower with the Permanent interface, because they must move the mouse cursor between the overview for navigation and the detail view for clicking on circles, whereas in the Transient interface, the overview can be invoked and used immediately without first moving the mouse cursor.

The second task type involves *comparison* of 5 blue circles in the map and clicking on the smallest of them. The comparison task makes participants navigate and compare the size of blue circles at different locations in the map. We do not expect the Transient interface to have an advantage over the Permanent interface in this type of task. First, participants do not alternately navigate and interact with objects in the map; participants can navigate continually to the blue circles to compare them. Therefore the closeness of the transient overview is not important. Second, the overview may cover blue circles in the detail view that

participants must see to compare their size. Although the overview covers part of the detail view in both interfaces, the fixed corner position of the permanent overview may help participants learn to consistently move blue circles into the visible part of the detail view. In contrast, invoking the transient overview at different positions can make it harder for participants to consistently move blue circles into view. However, participants can simply dismiss the transient overview to get a clear view of a blue circle when it has been located.

Since the overview used in this experiment shows the entire map, large maps result in a higher zoom factor than small maps. We varied the size of the maps used in the tasks to investigate the effect of varying zoom factors and varying distances between colored circles used in tasks. First, selection tasks with large maps require more precision in mouse movement when interacting with the overview. For example, the field-of-view box is smaller at higher zoom factors, which makes it is harder to move the mouse cursor from the detail view and target precisely in the permanent overview. Thus, we expect participants to perform worse with the Permanent interface compared with the Transient interface in tasks with large maps. Second, multiple red circles may be visible simultaneously in the detail view if the map is small, whereas large maps require participants to move the detail view to show each of the red circles in turn. As a result, the cost of targeting the mouse pointer in the permanent overview increases. Consequently, we expect participants to complete tasks faster with the transient overview in selection tasks with large maps compared with small maps.

5.3 Materials

Participants used a MacBook Pro laptop computer with an optical wireless mouse for the experiment. The screen was set to a 1440 x 960 resolution, and the size of the window containing the map interface was 700 x 700 pixels. Participants were guided through the experiment by a task view to the left of the interface window.

Two sizes of maps were used in the experimental tasks: small maps of 2000 x 2000 pixels (containing 200 circles) and large maps of 5000 x 5000 pixels (containing 600 circles). In small maps, two or three red circles may be visible simultaneously in the detail view, whereas only one red circle may be visible in large maps.

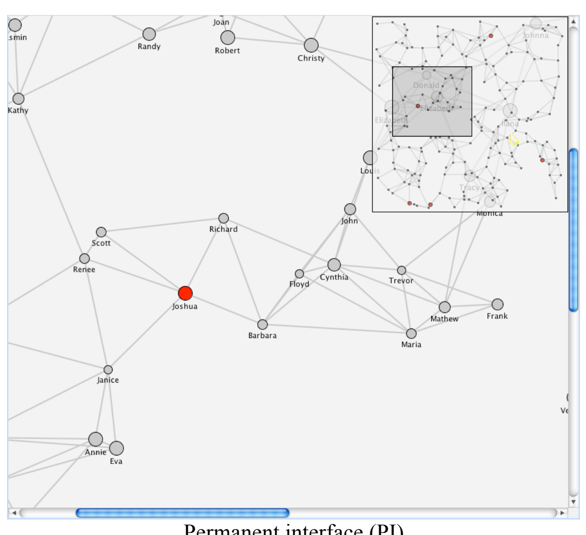
5.4 Design

We used a repeated measures design where four factors are varied within-subjects: interface type (PI, TI), size of the overview (O_{small}, O_{large}), task type (selection, comparison), and map size (small, large). Participants performed a set of 16 tasks with each interface. The order of interface and overview size was systematically varied across participants. The order of task type and map size for the eight tasks performed with each interface and overview size was also systematically varied. Thus 32 tasks with randomly generated maps were used; eight tasks for each combination of task type and map size.

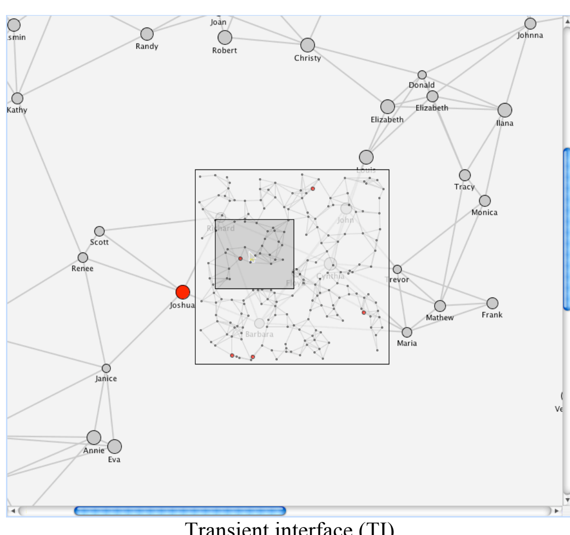
We used two sizes of overviews because the size of the overview may affect the usability of the two interfaces. We expect participants to prefer a small overview in the Permanent interface because it covers a smaller part of the detail view compared to large overview. In contrast, a large transient overview does not permanently cover part of the detail view, so we expect participants to prefer a larger overview to a small overview in the Transient interface. The small overview used is 25% the width of the detail view and the large overview is 40% of the width of the detail view.

5.5 Procedure

Initially, participants were given an introduction lasting about ten minutes. In the introduction, participants were explained how to use the two interfaces and given a few minutes to try them. Next in the introduction, participants performed 16 warm-up tasks; four selection-tasks with PI, four selection-tasks with TI, four comparison-tasks with PI, and four comparison-tasks with TI.



Permanent interface (PI)



Transient interface (TI)

Figure 5. The interfaces used in the experiment contain (left) a permanent overview in the upper-right corner and (right) a transient overview that is only visible when the right mouse button is pressed.

Participants performed two sets of tasks, one with each of the two interfaces. The participants were told to complete tasks correctly as quickly as possible. Following each set of tasks, a questionnaire about the interface just used was administered to the participants. The questionnaire contained six questions from QUIS [5] and five questions specific to the concerns of the experiment. A third questionnaire was administered after all tasks had been completed, asking the participants for their age and gender. The questionnaire also included three questions asking participants to compare the Transient interface with the Permanent interface on a comparative scale: first participants were asked which interface they preferred in general, then participants were asked which interface they found most appropriate for each type of task. Finally, participants were asked to write benefits and drawbacks of each interface and other comments. The entire experiment lasted between 30 and 45 minutes for each participant.

6. RESULTS

The results of the experiment consist of task completion times, accuracy and participant satisfaction. Of the 640 tasks that were completed across conditions, 13 tasks were discarded. First, due to an error in the experimental setup, two participants performed duplicate tasks and we discarded eight repeated tasks (two with TI, six with PI) because of possible learning effects. Second, we discarded three tasks (all with PI) where participants mistook a compare task for a selection task and clicked on the first blue circle that was visible. Third, two outlier tasks (both with TI), which either took more than 60 seconds for selection tasks or 30 seconds for compare tasks, were discarded.

6.1 Task Completion Times

Average completion times for the tasks are summarized in Table 1. We expected that participants would complete selection tasks faster using the Transient interface compared with the Permanent interface. In contrast, we did not expect comparison tasks to be performed faster with the Permanent interface. However, there was no significant difference in task completion times with the two interfaces for either type of task, $F(1, 19) = .293$, ns.

6.2 Accuracy

All of the selection tasks were completed correctly. In contrast, 273 of 310 comparison tasks were answered correctly. Accuracy is summarized in Table 2. Participants answer more tasks correctly with a large overview than a small overview, $F(1, 19) = 6.32$, $p < .05$. However, we find no influence of interface type on accuracy, $F(1, 19) = .812$, ns.

6.3 Satisfaction

Overall, participants preferred the Transient interface compared with the Permanent interface ($t = 3.387$, $df = 19$, $p < .005$), with 16 participants preferring the Transient interface and only four participants preferring the Permanent interface. There was no significant difference in what interface participants perceived to be most appropriate for selection tasks ($t = 2.070$, $df = 19$, $p > .05$) or comparison tasks ($t = 1.761$, $df = 19$, $p > .05$), although participants tended to prefer the Transient interface for both task types.

Table 2. Accuracy in comparison tasks for different interfaces, overview sizes and map sizes.

	Permanent interface			Transient interface		
	O_{small}	O_{large}	Avg.	O_{small}	O_{large}	Avg.
Small map	82.5%	97.4%	89.7%	90.0%	91.9%	90.9%
Large map	84.6%	97.2%	90.7%	80.0%	86.8%	83.3%
Average	83.5%	97.3%	90.2%	85.0%	89.3%	87.1%

Average satisfaction scores for the two interfaces are summarized in Figure 6 for the eleven questions that the participants answered. Overall, participants scored the Transient interface higher as assessed by multivariate analysis of variance, Wilk's lambda = .421, $F(1, 19) = 3.00$, $p < .05$. The results confirm our expectations that a transient overview reduces mental and physical efforts required of the user compared with a permanent overview. We had hypothesized that participants would prefer a large overview in the Transient interface and a small overview in the Permanent interface, but there was no significant difference between the interfaces in the size of overview that participants preferred.

6.4 Interaction Patterns

We analyzed the interaction data logged during the experiment to uncover differences in the use of the two interfaces. In selection tasks, interaction alternated between using the overview to bring circles into view and clicking on circles in the detail view. We expected the Transient interface to help participants complete these tasks with less mouse movement compared with the Permanent interface. To investigate this, we summed the distances that the mouse pointer traveled between mouse button events. Distance was calculated as the diagonal between screen coordinates of the mouse pointer. There was a substantial difference in the average distance per task for the two interfaces; a decrease of 60% from the Permanent interface to the Transient interface. Thus, the Transient interface appears to have reduced the sensory-motor efforts of the participants.

In comparison tasks, participants navigated between blue circles in the map to compare their sizes. The overview covered part of the detail view, especially in the large overview condition. Thus participants had to move the detail view, or dismiss the overview in the Transient interface, to get a clear view of the circles.

Table 1. Task completion times in seconds for different interfaces, overview sizes and task types.

		Permanent interface			Transient interface		
		O_{small}	O_{large}	Average	O_{small}	O_{large}	Average
Selection tasks	M	30.2	28.6	29.4	29.5	29.2	29.3
	SD	6.0	5.0	5.6	6.4	6.8	6.6
Comparison tasks	M	13.3	12.9	13.1	12.6	12.7	12.7
	SD	5.3	3.5	4.5	4.2	4.5	4.3
Average	M	21.8	21.0	21.4	21.0	21.2	21.1
	SD	10.2	9.0	9.6	10.0	10.0	10.0

Interestingly, participants mostly completed the tasks using the transient overview by continuously holding down the mouse button while navigating between the blue circles to compare them (in only 20 of 160 tasks, participants invoked the transient overview more than once). However, informal observations showed that participants using the Transient interface sometimes had trouble moving the blue circles clear of the overview—they did not dismiss the transient overview to get a clear view of the circle.

In the Permanent interface, most participants mainly clicked in the overview to move the detail view, a mode of interaction not supported in the Transient interface. Only three out of 20 participants dragged the field of view box as the main way of moving the detail view, which was the interaction mode also supported by the Transient interface.

7. DISCUSSION

The study reported in this paper provides initial insight into the general benefits of transient visualizations. We used tasks that focus on navigation to tease out differences between the interaction with the transient and with the permanent overview. In all, the results of our study suggest that having immediate and close access to the overview reduces sensory-motor efforts of the user. Surprisingly, we did not find this to reduce task completion times and error rates.

Even though participants preferred the Transient interface and completed the tasks with less mouse movement by accessing the overview immediately at the location of the cursor, they did not complete selection tasks faster. It seems that whereas the Transient interface helps moving red circles into the detail view, it does not help in acquiring the circles with the mouse. It is hard to move the map precisely using the overview in either interface: participants must make fine adjustments to position a target close to the overview if not move the mouse farther to acquire the target. However, compared with the Permanent interface where the overview is placed in a corner of the detail view, it is possible that positioning part of the map into the detail view demands more effort when the transient overview appears at different screen locations.

Some limitations must be considered when interpreting the results. Maps used were limited to sizes that allowed the entire map to fit in the overview. Larger maps require overviews with multiple levels of magnification. Furthermore, we focused on simple navigation tasks and participants used the interfaces for only a short period of time. Thus, our findings may not reflect varied, long-term use of the overviews. Additionally, three problems detracted from the usability of the Transient interface. First, we saw participants struggle with the overview when invoking the overview near the border of the detail view, making the overview only partly visible. Four participants commented on this problem in the questionnaire. Second, an implementation problem caused the transient overview to "stick" to the mouse cursor when dragging the field-of-view box out of the window, requiring participants to click in the detail view to make the overview disappear. Third, the data describing the interaction with the Permanent interface suggests that participants preferred to click in the overview to navigate in the map. However, the Transient interface only allowed users to drag the field-of-view box, because the overview was only visible while holding down the mouse button. Support for both interaction modes might improve the usability of the Transient interface. Toggling the

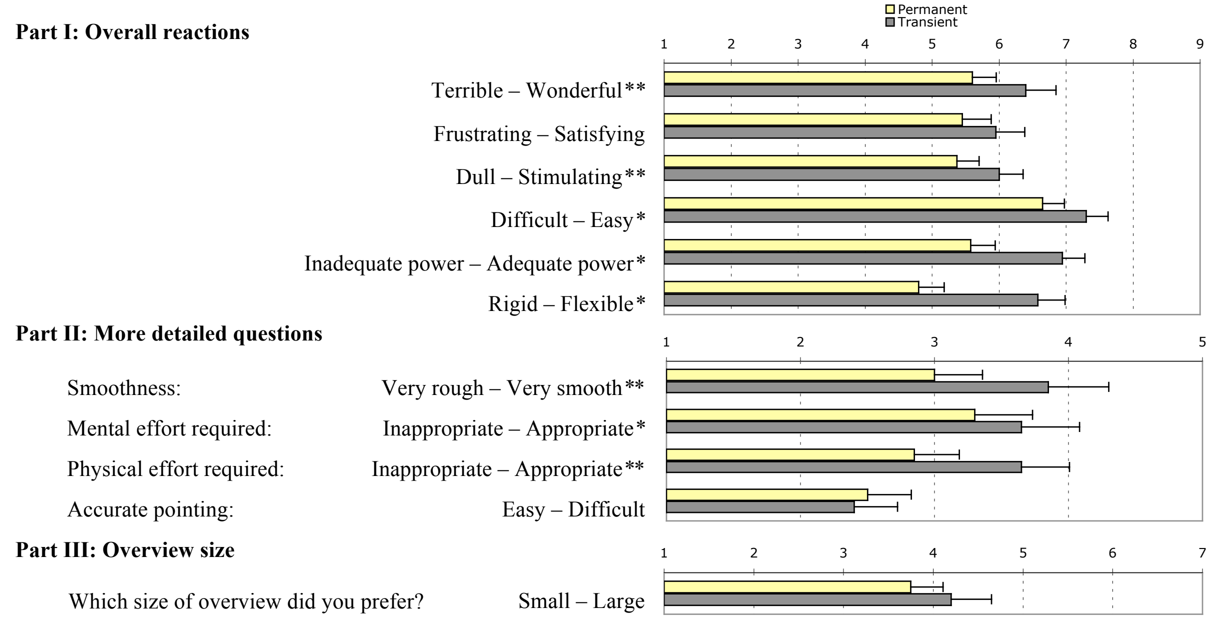


Figure 6: Average satisfaction scores (and standard error of the mean) for the eleven questions for the two interfaces. The anchor points on a semantic differential scale are shown for each question. Asterisks denote questions where the Transient interface scored significantly better (* = $p < .05$, ** = $p < .01$).

transient overview when the right mouse button is pushed is one possible solution.

More work is needed to further understand the general benefits and limitations of transient visualizations. Specifically, in the examples of transient visualizations presented in this paper, we have suggested the usefulness of transiently presenting contextual information related to the user's focus. Empirical evidence is needed to support this claim.

In complex work activity, transient visualizations may be useful to support sporadic tasks for which permanently changing the visual structure of information in the interface can impede frequent tasks. Studies are needed to understand what types of task that transient visualizations are suitable for. Evaluation of our transient fisheye view of source code may provide insights into the use of transient visualizations in expert tools.

Finally, conditions that limit the use of transient visualizations need to be examined. For example, transient visualizations that give no hint about their use are not accessible to novice users. Also, design and evaluation of transient visualizations must take into account that users may need longer practice time to make effective use of them compared to permanent visualizations that more readily afford their use.

8. CONCLUSION

We have characterized transient visualizations as interaction techniques that make immediate and transient use of information visualization close to, and in the context of, the user's focus of attention. In summary, transient visualizations offer a way of utilizing information visualizations to support specific contexts of use without making a permanent change to the user interface. We have presented examples of transient visualizations to support tasks in different domains.

To uncover how immediacy, transiency and closeness translate to actual and perceived improvements in the user experience, we conducted an experiment with map interfaces containing overviews. The results did not show significant improvements in time and accuracy with a transient overview compared to a permanent overview. However, our data suggest that tasks were performed with less sensory-motor efforts of the user, and 16 of the 20 participants preferred the transient overview.

Further studies are required to examine the general benefits and limitations of transient visualizations, to understand what types of task that transient visualizations are suitable for, and to provide design guidelines. Our initial data, however, suggest that transient visualizations may be useful, and that they are preferred by users to give immediate and close access to overviews.

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. *Proc. AVI '04*, 133–140, 2004. ACM Press.

[2] R. A. Becker and W. S. Cleveland. Brushing Scatterplots. *Technometrics*, volume 29, 127–142, 1987.

[3] A. Bezerianos and R. Balakrishnan. The vacuum: facilitating the manipulation of distant objects. *Proc. CHI '05*, pages 361–370, 2005. ACM Press.

[4] E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. D. DeRose. Toolglass and magic lenses: the see-through interface. *Proc. SIGGRAPH '93*, 73–80, 1993. ACM Press.

[5] 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, 1988. ACM Press.

[6] J.-D. Fekete and C. Plaisant. Excentric labeling: dynamic neighborhood labeling for data visualization. *Proc. CHI '99*, 512–519, 1999. ACM Press.

[7] K. Hornbæk, B. B. Bederson, and C. Plaisant. Navigation patterns and usability of zoomable user interfaces with and without an overview. *ACM Trans. Comput.-Hum. Interact.*, 9(4):362–389, 2002.

[8] K. Hornbæk and E. Frøkjær. Reading of electronic documents: the usability of linear, fisheye, and overview+detail interfaces. *Proc. CHI '01*, 293–300, 2001. ACM Press.

[9] T. Igarashi, J. D. Mackinlay, B.-W. Chang, and P. T. Zellweger. Fluid Visualization of Spreadsheet Structures. *Proc. VL '98*, 118-125, 1998. IEEE Computer Society.

[10] P. Irani, C. Gutwin, and X. D. Yang. Improving selection of off-screen targets with hopping. *Proc. CHI '06*, 299–308, 2006. ACM Press.

[11] M. R. Jakobsen and K. Hornbæk. Evaluating a fisheye view of source code. *Proc. CHI '06*, 377–386, 2006. ACM Press.

[12] G. Kurtenbach, G. W. Fitzmaurice, R. N. Owen, and T. Baudel. The Hotbox: efficient access to a large number of menu-items. *Proc. CHI '99*, 231–237, 1999. ACM Press.

[13] G. C. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[14] D. Nekrasovski, A. Bodnar, J. McGrenere, F. Guimbretière, and T. Munzner. An evaluation of pan & zoom and rubber sheet navigation with and without an overview. *Proc. CHI '06*, 11–20, 2006. ACM Press.

[15] B. Suh, A. Woodruff, R. Rosenholtz, and A. Glass. Popout prism: adding perceptual principles to overview+detail document interfaces. *Proc. CHI '02*, 251–258, 2002. ACM Press.

[16] M. Terry and E. D. Mynatt. Side views: persistent, on-demand previews for open-ended tasks. *Proc. UIST '02*, 71–80, 2002. ACM Press.

[17] A. Woodruff, A. Faulring, R. Rosenholtz, J. Morrison, and P. Pirolli. Using thumbnails to search the web. *Proc. CHI '01*, 198–205, 2001. ACM Press.

[18] A. Zanella, M. S. T. Carpendale, and M. Rounding. On the effects of viewing cues in comprehending distortions. *Proc. NordiCHI '02*, 119–128, 2002. ACM Press.

[19] P. T. Zellweger, S. H. Regli, J. D. Mackinlay, and B.-W. Chang. The impact of fluid documents on reading and browsing: an observational study. *Proc. CHI '00*, 249–256, 2000. ACM Press.

PAPER 3 – TRANSIENT OR PERMANENT FISHEYE VIEWS: A COMPARATIVE EVALUATION OF SOURCE CODE INTERFACES

Jakobsen, M. R. and Hornbæk, K. (2009). Transient or Permanent Fisheye Views: A Comparative Evaluation of Source Code Interfaces.

This paper has been submitted for publication. Copyright may be transferred without further notice and the accepted version may then be made available by the publisher.

Transient or Permanent Fisheye Views: A Comparative Evaluation of Source Code Interfaces

Mikkel R Jakobsen and Kasper Hornbæk

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 tasks of both high and low complexity so as to investigate varied programming activity. All participants used each of 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 analyze 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.

Index Terms—Information visualization, fisheye view, transient visualizations, user study, programming.

1 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 [5]. Seesoft, for example, maps source code into a 1-dimensional representation aimed at helping users understand changes to the code [8]. 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 [18].

The user's control of the visual structures and view transformations used is central to visualization [5]. However, often visualizations are designed to support a specific task and make fixed mappings and transformations that are effective in that task. In contrast, real life applications often support a variety of tasks in complex work settings. Integrating a visualization aimed at supporting a specific task in existing applications 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 make 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 [15]. By using transient visualizations to support infrequent and unpredictable contexts of use, the permanent view can be dedicated to information used in frequent contexts of use. However, empirical data on the relative benefits of transient and permanent interfaces are lacking.

This paper studies fisheye views of source code – a visualization that has been shown to help programmers in navigating and understanding source code [14]. The fisheye view as originally proposed by Furnas balances in a single view "the need for local detail against the need for global context" [10]. A fisheye view of source code does so by displaying only those parts of the code with the highest degree of interest given the user's current focus. However, information shown because it has a high degree of interest 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 [15], we present richer experimental data from a much more complex domain. We use the study to discuss both how to advance research in information visualization and to make concrete suggestions for using fisheye interfaces and other information visualizations to support complex domains.

2 Related Work

The idea of transient visualizations was introduced in [15]. The idea is motivated by the observation that information visualizations often make permanent changes to a user interface with the aim of supporting specific tasks. However, a permanent visualization cannot support the variety of tasks found in realistic work settings equally well. Thus, it may be more useful to display visualizations transiently. According to [15], 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. [1] 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 [20]. Zellweger et al. [23] 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 [16]. See-through tools are another technique that provides close and contextual access to commands without requiring permanent use of display space [4].

Many information visualizations use brushing to highlight (or affect) instances in other views of an object that the user brushes over [2]. 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 [12]. 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 of the user [3]. Similar challenges in small displays have brought about techniques to visualize and navigate to off-screen targets with halos and proxies [13].

Despite the above motivations for transient visualizations, the use of transient visualizations has to our knowledge only been empirically investigated in an experimental study of overview+detail map interfaces [15]. 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 [15].

3 Comparing Transient and Permanent Fisheye Views

To investigate how transient visualizations can be used to support complex work settings such as programming, we implemented a transient fisheye view of source code in Eclipse, a widespread development environment (see Fig. 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.

3.1 Fisheye View of Source Code

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

3.1.1 Degree of Interest

A degree of interest (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:

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

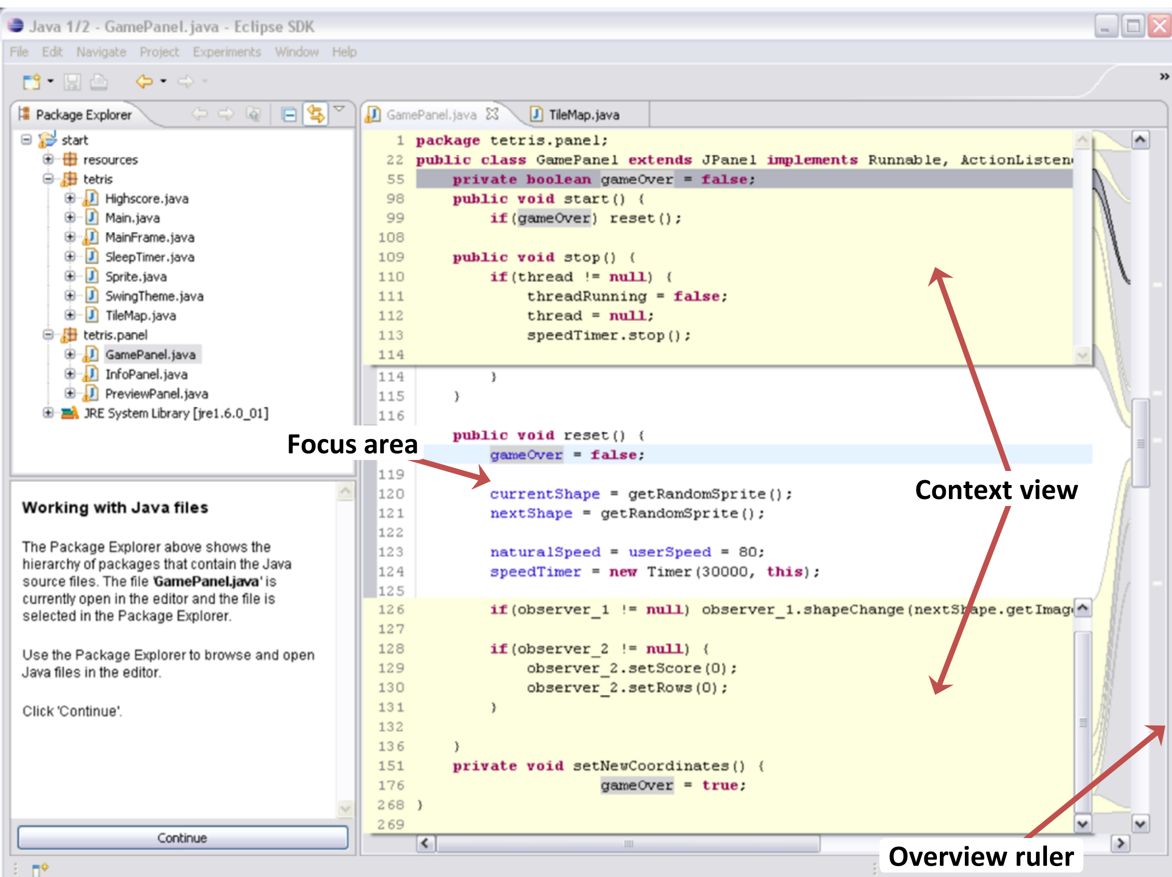


Fig. 1. Transient fisheye view called up in Eclipse to divide the Java editor window into a focus area and a context view. Lines containing occurrences of a selected variable are shown in the context view and in the overview ruler to the right of the scrollbar (as white rectangles).

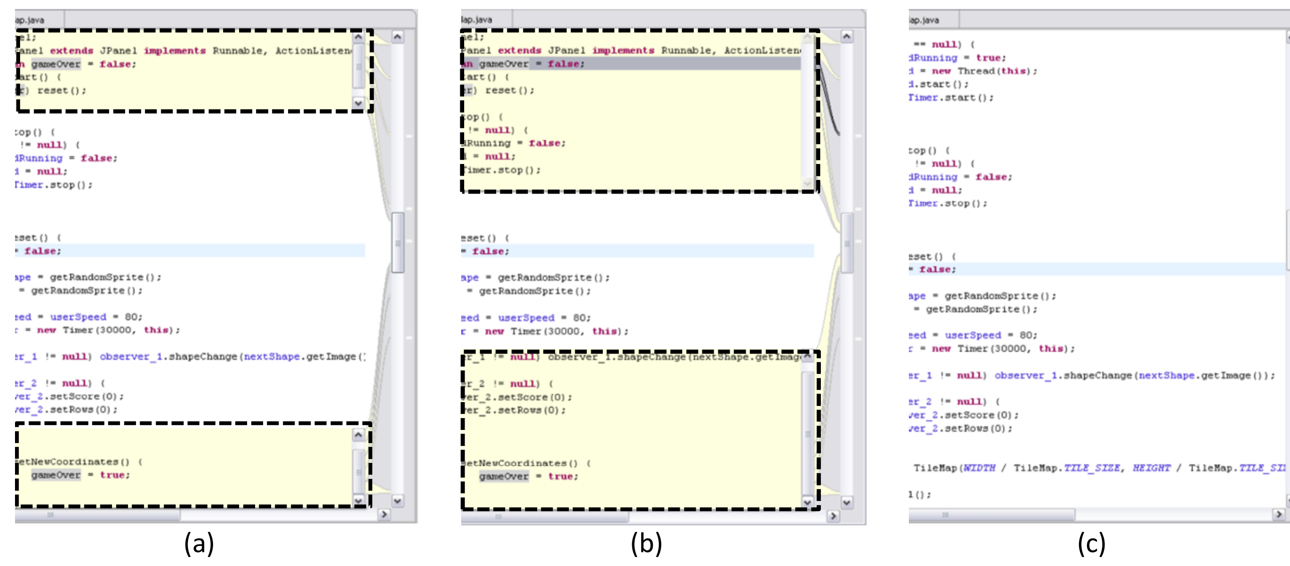


Fig. 2. The three Java editor interfaces used in the experiment: (a) Permanent interface in which permanently context view is permanently shown, (b) Transient interface in which the context view has been temporarily called up – otherwise it looks like the Baseline interface – and (c) Baseline interface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

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 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 enclose 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.

3.1.2 Source Code Elision in the Context View

Lines are included in the context view if they have a degree of interest above the threshold k. If there are not enough lines with $DOI > k$ to use all the space available in the context view, lines with $DOI <= k$ are added to the context view in descending order of DOI. This includes 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 view can be scrolled. The context view automatically scrolls to show lines closest to the focus area when its contents change.

3.2 Interfaces

Three interfaces to a Java editor were used in the experiment (see Fig. 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. 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. When 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.

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 [15]. 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 in 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 Fig. 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.

3.3 Participants

The 14 participants (one female) were computer science students enrolled at the authors' department. They were between 24 and 44 years of age (M = 30.1). All had at least one year of experience programming in an object-oriented language and all but two participants had experience with Java. Half of the participants had used Eclipse before, but only one had used Eclipse within the last month.

3.4 Tasks

Two sets of tasks were used in the experiment. *High-complexity tasks* involved investigating the source code of a program. *Low-complexity tasks* involved five types of understanding and navigation task. Tasks of high complexity 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. Tasks of low complexity are well structured, have a single path to a single precise answer, and limited information is needed to answer the task.

High-complexity 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 give rise to individual approaches of participants to seek the information they need to answer the tasks. In contrast, low-complexity 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.

Participants performed high-complexity tasks before low-complexity tasks in the experiment. Participants thus had time to learn to use the interfaces before performing low-complexity tasks, which may increase the reliability of the results in those tasks. Below we describe each set of tasks in detail.

3.4.1 High-complexity tasks

The high-complexity 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.

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 thousand lines of code), 11 tasks used JDraw (jdraw_v1.1.5.src.zip downloaded from http://jdraw.sourceforge.net/ contained 23K-LOC) and 10 tasks used Magelan (magelan-1-3.zip downloaded from http://sourceforge.net/projects/magelan/ contained 39K-LOC). Some tasks involved more than one file, for example: "Classes 'AbstractNode' and 'AbstractConnection' (in org.tinyuml.draw) are diagram elements. What is the field 'parent' used for in the two classes?"

The difficulty of high-complexity 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.

3.4.2 Low-complexity tasks

Five types of low-complexity task were used. These tasks involved navigating and understanding source code. The order in which these types of task were used in the experiment was systematically varied. Tasks were taken from previous studies of programming activity [7][14]. 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.
- **Determine-control-structure tasks** that required finding a control structure within a single method, for instance: "In the method 'mergeTermInfos' (line 201–238), how many for, while and if/else statements enclose line 233?"
- **Determine-dependencies tasks**, for instance determining calls to a particular method: "How many methods in this file contain calls to 'computeProposals' declared on line 470?"
- **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?"
- **Determine-delocalization tasks** involved determining delocalization in the source code, for example: "The method 'update' (line 207–214) contains 6 method calls. How many of the methods called are declared in this file (that is, excluding inherited methods)?"

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

3.5 Materials

The experiment was conducted in a laboratory with six identical computers with 19" CRT monitors at a resolution of 1280 x 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 (see Fig. 1). Participants typed their answer to the tasks in the task view and clicked a button to continue. In the set of high-complexity 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 low-complexity 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.

3.6 Design

Two factors were varied in a within-subjects design: interface (Permanent, Transient, Baseline) and task complexity (High-complexity, Low-complexity). 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 (see Fig. 3). In each block, participants used one of the three interfaces. The order of interface was systematically varied across participants.

3.7 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 high-complexity 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 low-complexity tasks. For these tasks,

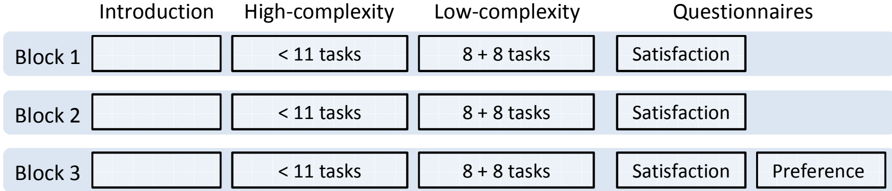


Fig. 3. The experimental design in which interface was varied between the three blocks.

participants were 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 QUIS [5] 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.

Participants met in the laboratory on three days for each of the three blocks of the experiment, except one participant completed two blocks in one day. The experiment lasted between four and six hours per participant. The experiment was conducted over a period of one week. Hence up to six participants were present in the laboratory at a time. 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.

4 Results

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

4.1 High-Complexity Tasks

In the first task set comprising high-complexity 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 in that 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 did not have time 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 high-complexity tasks with each interface. Because of the 55-minute limit for solving the high-complexity 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, $F_{1,13}$ = .243, ns. If anything, participants appeared to complete fewer tasks using the Transient interface than Permanent or Baseline.

Table 1 shows average completion times for high-complexity 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, $F_{2,243} = 4.34$, $p < .05$. Although participants appeared to complete fewer tasks using the Transient interface, participants who completed all tasks spent less time with Transient compared with Permanent ($p < .05$ in Bonferroni adjusted post-hoc tests). Completion times did not differ significantly between Transient and Baseline.

4.2 Low-Complexity Tasks

Data from 25 low-complexity tasks (out of 336) were discarded from our analysis because participants did not appear to understand the question (7), wrote ambiguous answers (5), or wrote verbose answers (12). Finally, an outlier that was more than three times above the inter-quartile range was discarded.

Table 1. Average Task Completion Times in Seconds for Different Interfaces and Task Complexity

	Permanent			**Transient**			**Baseline**		
	N	*M*	*SD*	*N*	*M*	*SD*	*N*	*M*	*SD*
High-complexity tasks (participants completing all tasks)	106	267	132	64	210	111	74	248	118
Low-complexity tasks	112	47.1	20.1	112	49.4	24.3	112	49.1	24.9

Table 2. Summary of Answers Given to High-Complexity Tasks Using the Three Interfaces

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

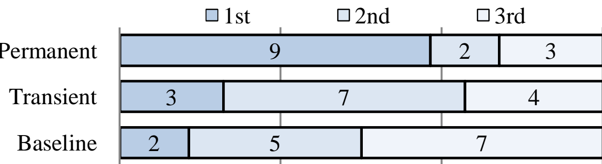


Fig. 4. Number of participants ranking each interface as 1st, 2nd or 3rd choice (from left to right). For example, nine participants ranked Permanent as their first choice.

Overall, 85% of the low-complexity tasks were answered correctly. There was no difference in accuracy with the three interfaces, $F_{1,13}$ = .089, ns. Nor were task completion times different between interfaces, $F_{1,13}$ = .310, ns. Average task completion times are summarized in Table 1. While interface was found to interact with task type, $F_{1,13}$ = 2.19, p < .05, there were no significant differences in completion times with the interfaces for any type of task.

4.3 Satisfaction

After having used all three interfaces, participants completed a questionnaire to rank the interfaces. Participants' ranking of the interfaces differed significantly, $F_{1,13}$ = .035, p < .05. Fig. 4 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 .05 level as found by a multivariate analysis of variance, Wilk's Lambda = .027, $F_{1,13}$ = 1.78, p = .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 < .05), finding both Permanent and Transient more fun to use than Baseline (p < .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] wouldn't 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.

4.4 Interaction with the Interfaces

We analyzed the data logged during the experiment to understand how participants used the interfaces. We summarized interaction data from high-complexity tasks to measure how participants adopted and used the context view in the fisheye interfaces. We visualized the interaction data from low-complexity tasks in progression maps (similar to [11][14]) and analyzed these maps to understand how participants used each interface to solve the tasks. The progression maps show which part of the file was visible in the focus area at a given time during the task (see Fig. 5). Dashed horizontal lines indicate program lines that hold part of the answer to the task, and symbols in the progression maps annotate certain types of interaction (e.g., a hand symbol when the user dragged the scrollbar thumb; a text caret when the user placed the caret in the focus area, for instance to highlight a method). Other interaction forms are directly discernable from the map, such as scrolling by

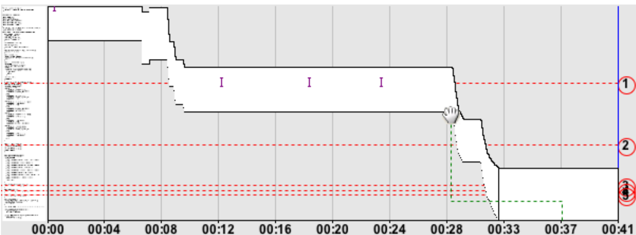


Fig. 5. Progression map for a Determine-dependencies task using Permanent interface.

page up/down keys. Interpreting the progression maps is not always straightforward. For instance, the task shown in Fig. 5 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 high-complexity tasks varied in structure, and in some cases involved multiple files, we did not use progression maps to analyze those tasks.

4.4.1 High-Complexity Tasks

In average, participants interacted with the context view in 64% of the high-complexity tasks they completed using the Permanent interface and 70% of the tasks using the Transient interface. In all, participants used the context view an average of eleven times to navigate in the code, 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 in all tasks.

Using the Permanent interface, ten 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 high-complexity tasks. Using the Transient interface, two participants did not once use the context view, whereas the other 12 participants used the context view in more than half of the tasks.

4.4.2 Low-Complexity Tasks

Analysis of progression maps for low-complexity tasks revealed patterns in the participants' interaction with the interfaces. In all low-complexity 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. Fig. 6 shows progression maps that are representative of this type of interaction using each of the interfaces. Using the Permanent interface or the Transient interface, participants could often find the lines in the context view that contained the answer to

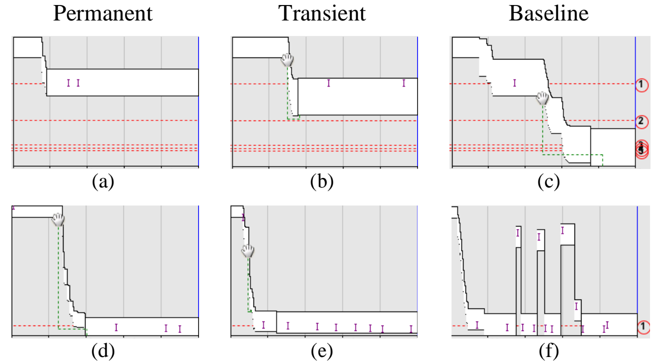


Fig. 6. Progression maps representative of participants' navigation when using the three interfaces in (a-c) a Determine-dependencies task involving method calls and (d-f) a Determine-delocalization task.

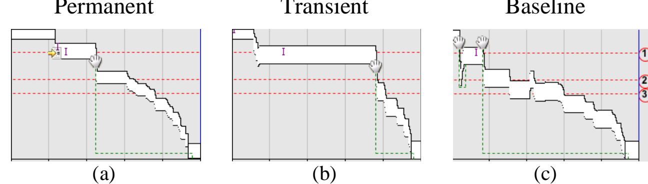


Fig. 7. Example progression maps where participants scrolled through the entire file to solve a Determine-dependencies task involving variable assignments.

the task without navigating further. Using the Baseline interface, participants often seemed to navigate to lines with highlighted occurrences, which might contain the answer to the task. 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 (see Fig. 6 (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 (see Fig. 6 (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 Fig. 6 (c) (39 tasks), or by clicking in the overview ruler like in Fig. 6 (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 Fig. 7), 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 effectively using occurrences to solve Determine-dependencies tasks where they should determine which methods contained calls to a particular method (see Fig. 6 (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 high-complexity 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 low-complexity tasks. In high-complexity tasks, two of these participants used it frequently, whereas one used it only occasionally.

5 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.

5.1 Focus+Context Interfaces for Source Code

Results from the study confirm earlier empirical findings in support of fisheye views of source code [14] and code elision [7]. 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 [14], 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.

5.2 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 the present study may be contrasted to the empirical findings of [15]. 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 in [15], which focused narrowly on navigation, participants in the present 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 calling 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 degree-of-interest 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 degree-of-interest function underlying the fisheye view.

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 favoured the Permanent interface.

6 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 [14][7]. 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 over 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 degree-of-interest 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] P. Baudisch, B. Lee, and L. Hanna. Fishnet, a fisheye web browser with search term popouts: a comparative evaluation with overview and linear view. *Proc. AVI '04*, 133–140, 2004. ACM Press.

[2] R. A. Becker and W. S. Cleveland. Brushing Scatterplots. *Technometrics*, volume 29, 127–142, 1987.

[3] A. Bezerianos and R. Balakrishnan. The vacuum: facilitating the manipulation of distant objects. *Proc. CHI '05*, 361–370, 2005. ACM Press.

[4] E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. D. DeRose. Toolglass and magic lenses: the see-through interface. *Proc. SIGGRAPH '93*, 73–80, 1993. ACM Press.

[5] S. K. Card, J. D. Mackinlay, and B. Shneiderman. *Readings In Information Visualization: Using Vision To Think*. Academic Press, 1999.

[6] J. P. Chin, A. Virginia, and K. L. Norman. Development of an instrument measuring user satisfaction of the human-computer interface. *Proc. CHI '88*, 213–218, 1988. ACM.

[7] A. Cockburn and M. Smith. Hidden messages: evaluating the efficiency of code elision in program navigation. *Interacting with Comp.*, 15:387–407, 2003.

[8] S. G. Eick, J. L. Steffen, and E. E. Sumner. SeeSoft - A Tool for Visualizing Line Oriented Statistics Software. *IEEE Transactions on Software Engineering, 18*, 957-968, 1992.

[9] J.-D. Fekete and C. Plaisant. Excentric labeling: dynamic neighborhood labeling for data visualization. *Proc. CHI '99*, 512–519, 1999. ACM Press.

[10] Furnas, G. W. The Fisheye View: A New Look at Structured Files, Bell Laboratories Technical Memorandum #81-11221-9, 1981.

[11] K. Hornbæk and E. Frøkjær. Reading of electronic documents: the usability of linear, fisheye, and overview+detail interfaces. *Proc. CHI '01*, 293–300, 2001. ACM Press.

[12] T. Igarashi, J. D. Mackinlay, B.-W. Chang, and P. T. Zellweger. Fluid Visualization of Spreadsheet Structures. *Proc. VL '98*, 118-125, 1998. IEEE Computer Society.

[13] P. Irani, C. Gutwin, and X. D. Yang. Improving selection of off-screen targets with hopping. *Proc. CHI '06*, 299–308, 2006. ACM Press.

[14] M. R. Jakobsen and K. Hornbæk. Evaluating a Fisheye View of Source Code, *Proc. CHI '06*, 377–386, 2006. ACM Press.

[15] M. R. Jakobsen and Hornbæk. Transient Visualizations, *Proc. OZCHI '07*, 69–76, 2007. ACM.

[16] G. Kurtenbach, G. W. Fitzmaurice, R. N. Owen, and T. Baudel. The Hotbox: efficient access to a large number of menu-items. *Proc. CHI '99*, 231–237, 1999. ACM Press.

[17] D. Nekrasovski, A. Bodnar, J. McGrenere, F. Guimbretière, and T. Munzner. An evaluation of pan & zoom and rubber sheet navigation with and without an overview. *Proc. CHI '06*, 11-20, 2006. ACM Press.

[18] G. G. Robertson and J. D. Mackinlay. The document lens. In UIST '93: *Proc. UIST '93*, 101–108, 1993. ACM.

[19] B. Suh, A. Woodruff, R. Rosenholtz, and A. Glass. Popout prism: adding perceptual principles to overview+detail document interfaces. *Proc. CHI '02*, 251–258, 2002. ACM.

[20] M. Terry and E. D. Mynatt. Side views: persistent, on-demand previews for open-ended tasks. *Proc. UIST '02*, 71–80, 2002. ACM Press.

[21] A. Woodruff, A. Faulring, R. Rosenholtz, J. Morrison, and P. Pirolli. Using thumbnails to search the web. *Proc. CHI '01*, 198–205, 2001. ACM Press.

[22] A. Zanella, M. S. T. Carpendale, and M. Rounding. On the effects of viewing cues in comprehending distortions. *Proc. NordiCHI '02*, 119–128, 2002. ACM Press.

[23] P. T. Zellweger, S. H. Regli, J. D. Mackinlay, and B.-W. Chang. The impact of fluid documents on reading and browsing: an observational study. *Proc. CHI '00*, 249–256, 2000. ACM Press.

PAPER 4 – FISHEYES IN THE FIELD: USING METHOD TRIANGULATION TO STUDY THE ADOPTION AND USE OF A SOURCE CODE VISUALIZATION

Jakobsen, M. R. and Hornbæk, K. (2009). Fisheyes in the field: using method triangulation to study the adoption and use of a source code visualization. In *Proceedings of the 27th international Conference on Human Factors in Computing Systems* (Boston, MA, USA, April 04 - 09, 2009). CHI '09. ACM, New York, NY, 1579-1588.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CHI 2009, April 4–9, 2009, Boston, Massachusetts, USA.
Copyright 2009 ACM 978-1-60558-246-7/09/04...$5.00.

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

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

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

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

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

INTRODUCTION

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

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

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

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

RELATED WORK

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

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

Evaluating Information Visualization

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

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

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

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

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

Fisheye Interfaces as a Case

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

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

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

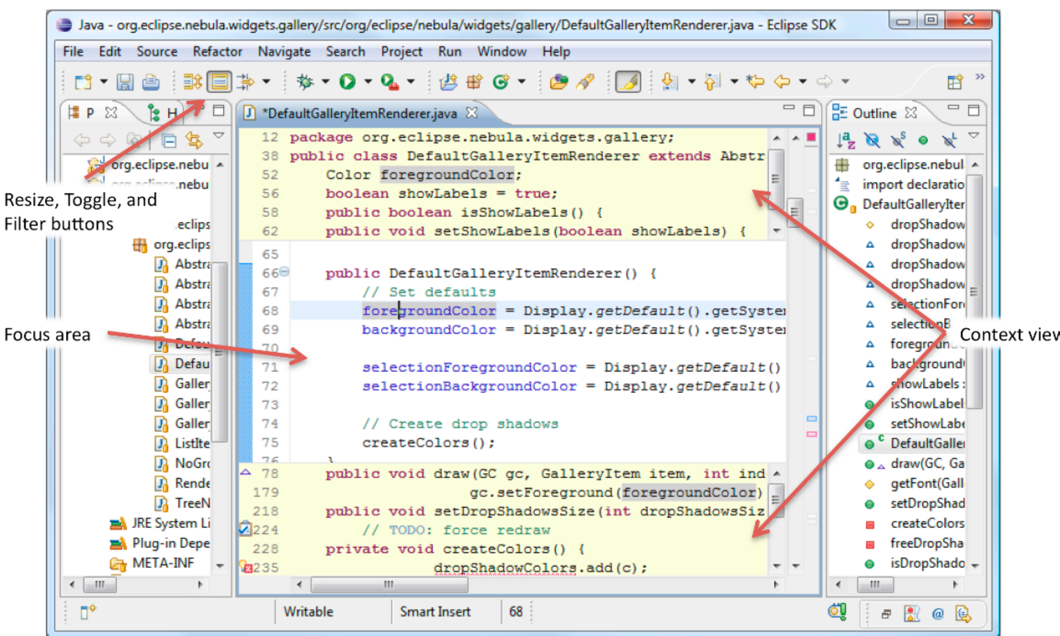


Figure 1: The Fisheye Java editor in Eclipse.

FISHEYE JAVA EDITOR

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

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

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

Degree of Interest

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

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

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

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

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

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

Source code elision in the context view

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

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

Interacting with the Fisheye Java editor

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

Filtering and customizing the context view

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

FIELD STUDY WITH PROFESSIONAL PROGRAMMERS

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

Participants

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

Method

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

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

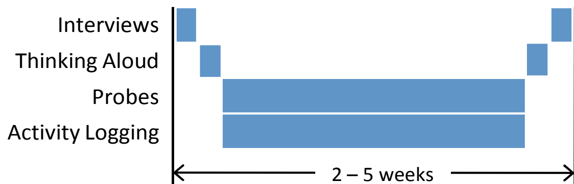


Figure 2: Use of methods to gather data about participants' programming activity and their experience using the Fisheye Java editor.

Thinking Aloud

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

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

Activity Logging

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

Probes

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

Interviews

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

Procedure

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

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

RESULTS

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

Thinking Aloud

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

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

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

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

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

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

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

Activity Logging

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

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

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

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

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

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

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

Probes

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

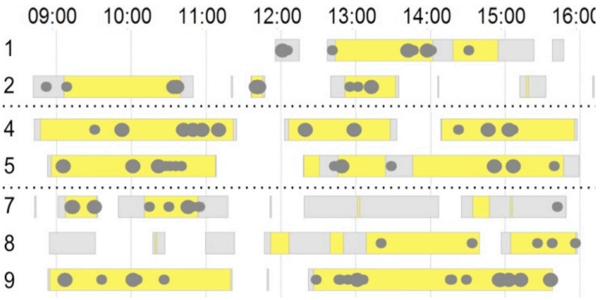


Figure 3: Example of timeline visualization of seven days (y-axis) of interaction with Eclipse and the Fisheye Java editor. Periods of editing are yellow; periods of interaction with Eclipse are gray; gray circles indicate use of the context view.

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

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

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

Interviews

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

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

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

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

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

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

DISCUSSION

Findings on Fisheye Interface in Programming

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

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

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

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

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

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

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

Strengths of Methods in Combination

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

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

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

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

Laboratory Experiment vs. Field Study

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

CONCLUSION

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

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

REFERENCES

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

PAPER 5 – WIPDASH: WORK ITEM AND PEOPLE DASHBOARD FOR SOFTWARE DEVELOPMENT TEAMS

Jakobsen, M. R., Fernandez, R., Czerwinski, M., Inkpen, K., Kulyk, O., and Robertson, G. (2009). WIPDash: Work Item and People Dashboard for Software Development Teams. 14 pages to appear in *Proceedings of INTERACT 2009 - 12th IFIP TC13 Conference in Human-Computer Interaction* (Uppsala, Sweden, August 24-28, 2009).

The copyright to the paper has been transferred to Springer-Verlag GmbH Berlin Heidelberg. The copyright transfer covers the sole right to print, publish, distribute and sell throughout the world the said Contribution and parts thereof, including all revisions or versions and future editions thereof and in any medium, such as in its electronic form (offline, online), as well as to translate, print, publish, distribute and sell the Contribution in any foreign languages and throughout the world (for U.S. government employees: to the extent transferable). Springer has given the author the right to self-archive an author-created version of his article on his/her personal website.

Permission to reprint granted by Springer.

This is the author's version of a paper published by Springer.

The copyright to the paper has been transferred to Springer-Verlag GmbH Berlin Heidelberg. The copyright transfer covers the sole right to print, publish, distribute and sell throughout the world the said Contribution and parts thereof, including all revisions or versions and future editions thereof and in any medium, such as in its electronic form (offline, online), as well as to translate, print, publish, distribute and sell the Contribution in any foreign languages and throughout the world (for U.S. government employees: to the extent transferable). Springer has given the author the right to self-archive an author-created version of his article on his/her personal website.

WIPDash: Work Item and People Dashboard for Software Development Teams

Mikkel R. Jakobsen[1], Roland Fernandez[2], Mary Czerwinski[2], Kori Inkpen[2], Olga Kulyk[3], and George G. Robertson[2]

[1] Department of Computer Science, University of Copenhagen, Denmark
mikkelrj@diku.dk
[2] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
{rfernand, marycz, kori, ggr}@microsoft.com
[3] Human Media Interaction, University of Twente, 7500 AE, Enschede, The Netherlands
okulyk@utwente.nl

Abstract. We present WIPDash, a visualization for software development teams designed to increase group awareness of work items and code base activity. WIPDash was iteratively designed by working with two development teams, using interviews, observations, and focus groups, as well as sketches of the prototype. Based on those observations and feedback, we prototyped WIPDash and deployed it with two software teams for a one week field study. We summarize the lessons learned, and include suggestions for a future version.

Keywords: Information visualization, software development, large display, cooperative work, CSCW, situational awareness, field study.

1 Introduction

Team collaboration and coordination in software development is difficult [14]. First, it may require frequent coordination to plan and review progress of a team. Second, completing a task often involves team collaboration because knowledge is divided between team members who have different roles or own different parts of the system. Team members may work on multiple task items at a time, or belong to more than one team, adding to the challenge of coordination. Thus, team members need to be aware of what others on the team are doing [10,14].

In this paper, we present WIPDash (Work Item and People Dashboard), a visualization of work items in a team's software repository. Our goal is to help software teams be aware of the overall status of a project, and understand ongoing activities related to the team. Initially, we conducted interviews and field observations within a software development organization in order to understand the needs of collocated software teams. We then discussed these results in a series of focus group to iteratively design WIPDash. Finally, we deployed WIPDash with two software teams in an attempt to observe which features and functions the team actually used, and how they used those features. Our findings led us to a number of design lessons, and yet another design iteration, which we introduce at the end of the paper.

The main contributions of this work include (1) detailed findings about how developers maintain team awareness using existing techniques and tools, (2) a novel awareness visualization based on developers' needs, and (3) lessons learned from a deployment with two teams along with a conceptual overview of new design ideas based on that deployment.

2 Supporting Software Team Awareness

Many organizations adopt Agile Software Development methodologies that promote shorter iterations and daily stand-up meetings to improve coordination [15]. Also, collocation of a team in a shared team space may improve productivity [19]. Yet, software teams are still challenged with maintaining awareness of ongoing activity [10], and find collaborative tools that support their development work useful [14].

Software teams typically store information about work items (e.g., tasks and bugs) in software repositories, such as Microsoft Team Foundation Server (TFS), to support team coordination. However, such repository systems are not designed to give an overview of the state of a project or to keep team members aware of the team's current activities. In addition, it is not easy to see changes to work items in a software repository. Developers may not feel that they get a proper return on the time they invest on updating work items and often the status of work items is not up to date.

One way to improve team awareness is to show data from a team's repository on a large display in a shared workspace [e.g., 1,8]. FASTDash [1] showed developers' current activities in a code base. A field study showed that FASTDash increased communication within the team by 200%, and helped participants know who had which files checked out, who was blocked and needed assistance, and helped resolve conflicts with checked out code. Improvements were suggested based on the study, such as using metrics other than file size to allocate screen space, and to add support for people to track work items that are assigned to them. Still, it was not clear which types of information were most useful or how visualizations could be best designed to get awareness information at a glance. O'Reilly et al. [12] visualized checked-in code changes on a multi-monitor display. The authors concluded that the visualization helped to inform developers about progress and overall effort of the team. However, it is not clear how participants used the display. De Souza, Froelich and Dourish [16] have shown that source code could be mined to visualize both social and technical relationships of projects. We were inspired by these findings and focused our visualization on support for work item awareness.

Fitzpatrick et al. [6] described a long-term study of a software team using a tickertape tool where messages from CVS, a revision control system, were displayed. The authors found that the tickertape tool stimulated more focused discussion about source code changes, reduced the number of empty check-in messages, and helped coordinate and negotiate work within the team. The authors mentioned the modest screen real estate requirements of the tickertape tool as an important benefit.

Hill and Holland [9] describe the concept of showing the history of a user's interactions with files as part of the representation of the files. Recent research has empirically studied use of interaction history to help software development teams

[3,7]. TeamTracks [3] directs the attention of a programmer to important parts of the source code based on the history of programmers' interactions with the code. Augur [7] combines information about code activity with a line-oriented source code visualization similar to SeeSoft [4]. Froehlich and Dourish [7] present case studies of four developers who used Augur to gain insight into their code and their development activities. Their findings support the idea of combining information about activity and code in one view that is based on spatial organization of the code. However, Augur's potential usefulness as an awareness tool for a collocated team remains unknown.

The research efforts mentioned so far involved representations of source code, check-ins, and the use of code files. In contrast, Ellis et al. [5] aimed at helping large distributed software teams to coordinate their work on change requests by visualizing bugs. They presented SHO, a visualization with bugs shown as circles ordered, colored, and sized by different importance metrics. Participants in an experiment were more successful at completing tasks using SHO than using Bugzilla. Another recent study by Sarma et al. [14] presented a desktop awareness system based on code activity and check-ins. Their Palantír tool addressed mainly artifact changes in order to prevent potential conflicts. The visualizations were useful for exploring databases of bugs to identify areas of concern. However, it is not clear how useful these types of visualizations are for maintaining awareness of work items activity and project progress in collocated software teams, which is the focus of this paper.

3 User-Centered Design

The research presented in this paper follows a user-center design approach. We began by gathering observations and conducting semi-structured interviews with software developers to gain insight into their work practices and needs in collocated team workspaces. Our goal was to understand how collocated teams coordinate their work and to get input on what visualization features and views would be most useful for them. We then sketched an initial design of a visualization to support team awareness, which we presented to a focus group for feedback.

3.1 Interviews and Field Observations

In situ observations were performed with two Agile teams (see Figure 1). The teams were observed for five hours during one week. Observations were carried out at different times of the day and included morning stand-up meetings and iteration planning meetings. One team typically had five to ten members present, while the other team typically had eight to fifteen members present. We also carried out semi-structured interviews with ten individuals from these teams (eight males). Each interview lasted 35-40 minutes and was audio recorded with the participants' permission. Interviewees received a free lunch coupon as gratuity for their participation. Each interviewee had two to fifteen years of experience in software development and ranged in age from 20 to 46. Based on questions from related

Fig. 1. Shared team rooms for the two software teams we observed.

studies [10,17] and questions motivated by our in situ observations, the interviews focused on the following aspects:

- experiences working in a collocated team workspace;
- what tools and alerts are currently used to keep track of what other team members are working on and what is missing in existing tools;
- how work items and tasks are currently managed;
- types of meetings and the use of a projector in the team room;
- how progress and project health in general is monitored;
- wishes on what to display on the large shared screen and how to support work flow.

3.2 Interview and Field Observation Results

The interviews and observations showed that the teams work in iterations, which are blocks of time typically one to two weeks long. Daily stand-up meetings in the morning help the team to keep track of who works on what, and they are considered an important time for team bonding. Work items are created in a repository.

The teams used many software tools to coordinate their activities including Team Foundation Server (TFS), email, instant messaging, Live Meeting, and SharePoint. The teams varied in work style, size, and physical workspace arrangement. Both teams adopted a seating arrangement that corresponded to individual roles on the team: developers, testers, writers, and support (including program managers).

In addition to regular stand-up meetings, iteration planning, and bug triages (where resolved bugs are discussed and new bugs are assigned to team members), ad-hoc conversations frequently occurred in the team rooms. Some team members used chat or email, but often team members just shouted out a question or rolled their chairs over to talk to each other. Moreover, other people came in and out of the team spaces.

Although shared team rooms can be noisy and distracting, and offer less privacy than private offices, most team members felt that the team room was more effective for team work. An exception to this is documentation writers (two out of ten interviewees) who said that they preferred to work in a private office or from home; they needed to concentrate and only came to the team room for meetings or when they

needed to speak to a team member. Since they were frequently absent from the team room they tended to be less aware of activities that were going on within the team.

Both teams used shared whiteboards and sticky notes on the walls. Team members defined, categorized, and prioritized work items during iteration planning meetings and used sticky notes to represent tasks or work items. The teams also used a projector on the wall to display information for various tasks such as work items during iteration planning, when assigning new tasks, or for code reviews.

The dynamic nature of stand-up meetings requires a quick, glanceable overview of recent activity. Teams currently do not have a suitable tool for displaying important information. TFS gives no overview of unassigned tasks, and does not allow more than one person to be assigned to a task. Also, team members have to make a burn down chart or a task list for each meeting, which is time consuming. It is possible to export charts and work items from TFS, but this often results in a long Excel table with no way of synchronizing changes back with work items in TFS.

3.3 Focus Group Feedback

Based on what we learned from the interviews and observations, we sketched an initial design for a team awareness visualization. This initial design sketch, based on the current iteration of one of the team's work items repository, was presented to a focus group on a large projection screen in a meeting room to help participants imagine how the visualization would look when deployed in their own team room. Eight participants from two Agile development teams took part, including an architect, program managers, developers, lead developers and testers ranging in age from 30-48. The focus group session was video recorded with the teams' permissions.

Participants found that the awareness display presented information in a new perspective they had not seen before and they liked being able to see an overview of the whole project in one view. Team members expressed the need for different filters and view modes, since some work items might be irrelevant for their role.

We derived three key requirements for our design based on this feedback. The awareness visualization should (1) give an overview of iteration progress, with the ability to summarize over the last day, week, month, or version, (2) give details on individual work items and the people these are assigned to, and (3) list current and recent activities, either of people or on work items.

4 Work Item and People Visualization

Based on all of these results, we developed WIPDash, a visualization suitable for a large shared display in a collocated software team space. The visualization was implemented as a Windows application that reads data about work items from TFS. Our intention was that team members could glance at the shared display to see the overall status of the project and the recent changes made (especially from the past 24 hours). We also wanted team members to be able to use WIPDash on their individual

machines, where they could switch between different views and filters, and get details on demand.

The WIPDash window consists of two parts (see Figure 2). The left part of the window contains a spatial representation of areas of the project and the work items in those areas. For instance, the area labeled ‘Docs’ contains items related to project documentation. The right part of the window consists of a list of view modes, a team panel, and drop-down lists of iterations, work item states and types. These lists can be used to filter and highlight work items in the left view.

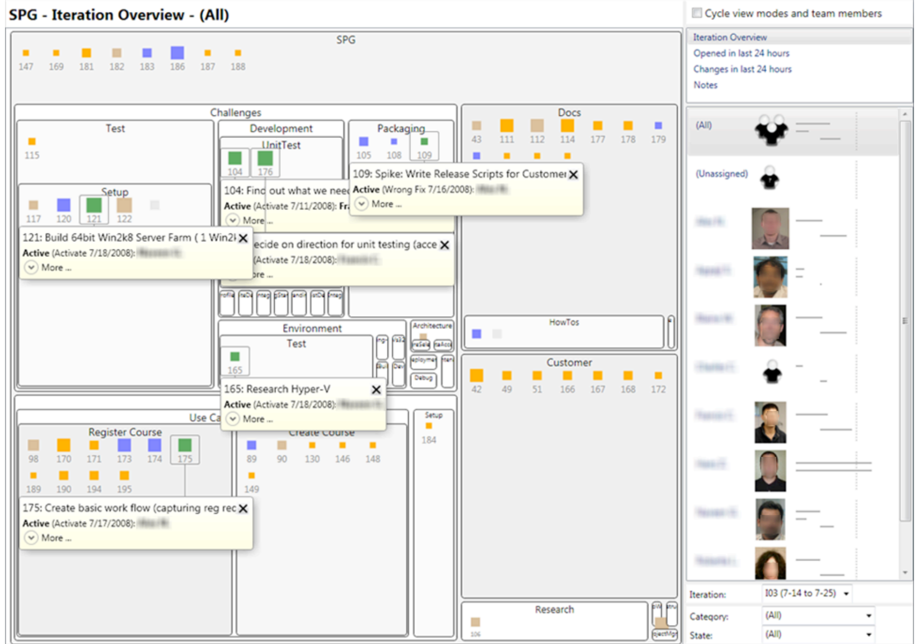


Fig. 2. WIPDash showing the iteration overview for the current iteration of a project.

4.1 Work Item Treemap

WIPDash uses a squarified treemap for laying out project areas as rectangles [2]. The treemap is a scalable approach to spatially organizing hierarchically structured data such as a hierarchy of project areas. Each rectangle is sized proportionally to the number of open work items in the area. A minimum threshold is used to ensure that areas that do not contain any open items are shown in the map. Rectangles are labeled with the name of the project area.

We wanted to preserve the spatial layout of project areas and work items to make it easier for users to remember where areas are located in the visualization. However, treemap algorithms can cause the spatial layout to change considerably when the data changes. Since WIPDash would be shown on both a large display and on individual team members’ displays (which may have different screen dimensions), the layout had to vary across instances of WIPDash. In order to keep the layout consistent for the purpose of the field study, the treemap was fixed and then shared by all instances

of WIPDash. This layout could be explicitly updated and the treemap would render again. Since the relative size of an area does not change dynamically to reflect the number of open work items, we color a rectangle darker as more open work items are associated with the area.

Each rectangle in the treemap contains icons that represent the work items associated with that area. The icons in a rectangle are evenly spaced in a grid, placed in the order they were created starting from the top-left corner. Space between icons is reduced to fit all icons within the rectangle, and if there is enough space between icons, the ID number of the item is shown below the icon. The color of an icon indicates the state of the work item (e.g., proposed, active, resolved, or closed) and the shape of an icon indicates the type of the work item (e.g., feature, bug, or task). Icon size represents either priority level or estimated hours of the work item, with larger sized icons representing items of higher priority or higher estimate of work hours, as designated by the user or team.

Moving the mouse cursor over a work item icon shows a tooltip with details about the item. Clicking on an icon shows a popup window with details about the work item. An 'Add note' section in the detail window can be expand to add a sticky note to the work item. A small yellow sticky note symbol is displayed on the work item to indicate that it has a note attached.

4.2 Iteration Filtering and Highlighting

Selecting an iteration in the "iteration list" shows all work items that are assigned to that iteration and highlights them on the treemap. Since teams are usually only interested in closed items for the current iteration, we removed closed items that were not assigned to the selected iteration in order to avoid clutter.

4.3 Team Panel

The team panel (see Figure 3a) contains the names and pictures of the team members. Clicking on a team member shows the icons for all work items assigned to or closed

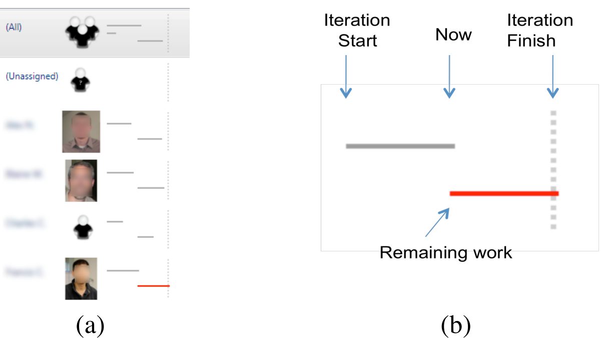


Fig. 3. (a) Team panel showing names and pictures of team members, and (b) a graphical representation of the amount of work each member has assigned, completed and remaining.

by the selected team member. The team panel contains two options in addition to the team members: (1) “all” which is used to select all items regardless of whom they are assigned to, and (2) “unassigned”, which is used to select all unassigned work items.

For each person, WIPDash shows horizontal lines that represent the total amount of work, the amount of completed work, and the work remaining in the iteration that is assigned to that person (Figure 3b). The x-axis measures work hours and a dotted vertical line represents the end of the iteration, corresponding to the total number of work hours in the iteration. The remaining work line is colored red if the estimated hours of remaining work exceeds the time left in the iteration.

4.4 View Modes

One goal of our design was to make the information on the display glanceable. Thus, to avoid cluttering the display by showing too much information in one view, users can choose between four different view modes.

The *Iteration overview* mode highlights all work items in the current iteration. This view aims to provide an overview of the iteration status. Team members can see how much work has been done and how many work items remain in the iteration. Details are automatically displayed for work items that are currently being working on, including who is working on the item. More details about an item can be shown by clicking on ‘*More*’ (see Figure 2).

The *Opened in last 24 hours* view is designed to keep the team aware of incoming tasks and issues. Work items opened within the last 24 hours are highlighted with a yellow border and background. The opacity of the border and background varies to distinguish recently opened items from items that were opened less recently. Also, similar to the iteration overview, details are shown for recently opened items, including when an item was opened and by whom.

The *Changed in last 24 hours* view aims to keep the team aware of recent changes to work items. Similar to the *Opened* view mode, a yellow border and background is shown around icons of work items that changed within the last 24 hours. Again, details are automatically displayed for recently changed work items, including the change made and who made the change. For simplicity, the same color coding is used for both *Opened* and *Changed* view modes in order to draw users' attention to work items with any recent activity.

The *Notes* view calls up the sticky notes for all work items with a note attached. The Notes view allows users to spot work items that need attention, for example if a team member makes a request to pair up on a specific task.

In order to provide continuous awareness and to allow passive use, WIPDash allows cycling through view modes and through team members within each view mode. A person remains selected for ten seconds before the next person on the team panel is selected. The next view mode is selected after cycling through all team panel selections. We were interested to see if this cycling behavior was useful or distracting to the teams we studied. The visualization updates with new or changed work items by querying Team Foundation Server (TFS) once per minute. All of the information used in our visualization came from the teams’ data entries in TFS.

5 In Situ Deployment

We deployed WIPDash with two collocated Agile software teams and observed its use for one week. Our aim was to understand the usefulness of the WIPDash and the effect it had on team members' situational awareness and on group processes. Team A had eight members (seven male), with an age range of 22-46. Team B had 16 members (14 male), with an age range of 27-48. Individual roles on both teams included lead developer, developer, tester, test lead, program manager, writer and group manager. WIPDash was installed on a large display in the team rooms and on team members' individual workstations. Data were automatically collected in WIPDash in order to describe how participants interacted with the visualization throughout the study. WIPDash was installed on a Thursday and an orientation session was given the following day. We observed the teams the following week (Monday through Friday). Afterwards, we met with each team for post-usage discussions. Each participant received a $50 gratuity coupon for their participation. In the Team A room, the awareness visualization ran on a projected wall display on the most accessible wall to the whole team. In the Team B room, the awareness display was installed on a 52-inch plasma touch screen toward the front and right side of the team room. Some members of Team B had their backs to this display.

5.1 Supporting Daily Stand-Up Meetings

Team A used WIPDash daily on their large display during stand-up meetings to coordinate meetings. Specifically, they found the *Iteration overview* useful, both in terms of status and also to jog their memories about work items from the previous day. During stand-up meetings, one of the team members selected each person from the team panel to display his information, and that person then talked about his work. The team considered details about the active items assigned to a person especially useful. Team members said they would have found it beneficial to have team members displayed in a random order during stand-up meetings—just to make it more fun. They also referred to the display to view the status of a remote team member when she called in for the standup meeting. This suggests that WIPDash could be useful for supporting collaboration with distant team members. Some team members commented that they liked to look at the awareness display first thing in the morning to see what team members in Argentina had been doing for the last ten hours. Some team members who were on vacation for most of the time during our study also said that they used the awareness visualization when they got back to get a sense of what the team had been doing for the past week and "*Where are we now*?" in the iteration.

We observed that the available information in WIPDash was not completely sufficient for reviewing what had been worked on during the previous day. Specifically, if a developer had completed his work on a work item and then reassigned the item to somebody else for testing, that item no longer showed in the *Changed* view for that developer. This was discussed during a stand-up meeting, and the team suggested a view where all work items that a person had worked on would be highlighted, even if they had been reassigned. The team further elaborated on an idea of one, concise overview containing all the information they would need for their

standup meetings, including recently resolved items that were reassigned to other members and items that members had worked on yesterday. Finally, the team expressed a wish for extending the shared wall display with an additional projector to show project information like spreadsheets or code next to WIPDash.

5.2 Automatic Cycling

The automatic cycling between views was found to be problematic. WIPDash cycled between all members in the team panel, including people with no items assigned or without any recent activity. Thus, nothing of interest is shown in parts of the cycle. Members of Team A suggested that cycling would make sense if done only between views that contain recent changes. Team B had only one work item assigned per person at a time and therefore cycling through each view for each team member was not useful.

Team members commented that notifications needed to be more assertive when a change or an update happened, such as an audio herald combined with a fisheye notification message about the change on the awareness display. Also, team members wanted to configure which views were displayed on their personal displays and when notifications should appear. An RSS-style feed would probably be a useful option for the personal workstations. We are pursuing that idea in the next iteration.

5.3 Use on Large Display and Individual Displays

We analyzed data logged by WIPDash to see how the two teams used WIPDash on the large display and on their individual machines. In all, we collected log data from ten personal machines, five members on Team B and five members on Team A, in addition to the two machines running the large shared displays. When a user started interacting with WIPDash, the default cycling between views was suspended. The log data showed that Team A used the large display (showing the iteration overview) during their stand-up meeting every morning around 9:30 am. Apart from the stand-up meetings, Team A physically interacted with the large display only twice during the study. In contrast, Team B interacted with the large display on average five times per day. In Team B some of the participants did not have WIPDash installed on their own machines. For this team WIPDash was running on a new touch screen display, which had a lot of appeal. One possible reason why Team A did not interact much with the large display was that it was projected high up on one of the walls, making personal display interaction more reasonable than going to the laptop in the corner of the room that controlled the view in order to interact with the group view.

5.4 Types of Interaction

In all, 596 selections were made in the right panel of WIPDash. Selections were primarily made in the team panel (66%). The view mode panel (17%), and iteration panel (15%) were also used to change the view. *Iteration overview* was the most

frequently selected view mode, selected more than 50% of the time. The *Changes* view and the *Opened* view were each selected between three and eight times, whereas the *Notes* view was only selected once.

Participants from Team A clicked on items to call up details 21 times while participants from Team B brought up additional details 22 times. Only two sticky notes were created in WIPDash, both by a lead developer on Team A. Follow-up discussions related to sticky notes revealed that users would rather use the existing 'comments' data structure in TFS to view and add notes to work items in WIPDash.

5.5 Questionnaires

Satisfaction with WIPDash was assessed using a questionnaire administered to team members at the post-usage meeting. The questionnaire contained nine questions from [13] and eight questions to address distraction and awareness, using five-point Likert-scale with lower scores reflecting negative responses. We balanced the valence of our satisfaction questions. For negatively phrased questions (marked with an asterisk in Table 1), we reversed the rating so that higher was always positive.

Ratings on usefulness and satisfaction with the system were mostly neutral to positive (see Table 1) and there were no significant differences between the two teams (paired t-test). Team members said that they had confidence that the information was displayed correctly and on time. From the ratings, it was clear that the notifications of changes were not grabbing attention well enough, and that WIPDash was seen as less reliable than we would have liked. However, the teams were not embarrassed to show their personal work item information and they leaned towards using a future version of WIPDash.

Table 1. User Satisfaction

	Question	Average Rating (SD)
1.	I have difficulty understanding WIPDash.*	4.10 (0.7)
2.	WIPDash is easy to use.	3.63 (0.8)
3.	WIPDash is reliable.	2.63 (1.1)
4.	I have confidence in the information provided by WIPDash.	3.90 (0.7)
5.	I need more training to understand WIPDash.*	4.05 (0.9)
6.	WIPDash is informative.	3.32 (0.9)
7.	WIPDash is comprehensible.	3.37 (0.8)
8.	Overall, I am satisfied with WIPDash.	3.00 (1.1)
9.	I would be happy to use WIPDash in the future.	3.11 (1.1)
10.	I find WIPDash distracting.*	3.95 (0.9)
11.	WIPDash grabs my attention at the right time.	2.84 (0.9)
12.	It's worth giving up the screen space to run WIPDash on my PC.	2.74 (0.9)
13.	WIPDash helps me stay aware of information that's critical.	2.72 (1.0)
14.	I like being notified when a work item gets reassigned.	3.22 (1.2)
15.	WIPDash's notifications often distract me.*	3.83 (0.8)
16.	Having WIPDash displayed in front of the team is embarrassing.*	4.17 (1.0)
17.	I would rather have WIPDash displayed only privately.*	4.18 (1.1)

A questionnaire on situational awareness and group satisfaction was administered to team members before in situ deployment and at the post-usage meeting. The questionnaire included questions from [11,18] and used five-point Likert-scale with lower scores reflecting negative responses. Paired t-tests showed no significant differences between the Before and After conditions. The results could be affected by a particular iteration stage or the day of the week the questionnaire was completed.

6 Discussion, Lessons Learned and Conceptual Redesign

Several factors may have affected the use and adoption of WIPDash. First, size and location of the shared display affected how team members made use of the display by glancing at it or physically interacting with it. During observations of Team A, we saw that team members often looked at the display when entering or leaving the room. This was not the case in the Team B room. One reason may be that the Team A room had a large display, projected high up on a wall, that was visible to everyone in the room. In contrast, Team B's shared display was smaller, located in a corner of the room, and was not directly visible to all team members.

Second, the two teams in our study organized their work differently. Team A assigned work items to team members during an iteration planning meeting, and had daily stand-up meetings to follow up on progress of the team. Using the shared display during stand-up meetings may have influenced Team A's familiarity with the display, and consequently increased their use of the display. In contrast, Team B did not have daily stand-up meetings; instead, they talked with each other about progress and status throughout the day. Also, Team B assigned only one work item to each person and viewed work in terms of releases that span several iterations, not single iterations. This suggests that Team B could make even better use of an overview of activities, but using different time spans.

Third, Team B had many proposed items in their repository, but most items were not scheduled to be worked on. Also, many of the project areas, which were shown in WIPDash because they contained proposed work items, were simply not considered relevant by the team at the time of our observations. Thus, as Team B had many more items to track than did Team A, and since WIPDash showed all work items in 'proposed' state, the display for Team B was more cluttered.

Our study has limitations that should be considered when interpreting the results. Although very different work styles were observed, the two teams were from the same organization. Therefore, our results may not be generalizable to software teams everywhere. Also, we only observed the teams for one week. While we learned much from the initial feedback, it would be interesting to see the long term effects of WIPDash. Another concern is that we do not know how the collocated Agile teams that we have focused on compare with larger, distributed teams. It could be that a focus on collocated, Agile software teams may reveal some issues in team coordination that also apply to distributed software teams. For example, Gutwin et al. [8] suggest a need for awareness of areas of expertise within a distributed open source project team because developers work on all parts of the code. This might relate to the information needs we seek to provide with our visualization (e.g., what people are

working on and have been working on). For example, a glance at WIPDash first thing in the morning to see what team members in Argentina have been doing for the last ten hours could be very useful in terms of setting daily priorities or offering more timely assistance. We intend to extend our focus to distributed team awareness in our future work.

Informed by the insights we have gained from our study of WIPDash, the design of the next version of the dashboard will include the following:

- A more glanceable display;
- sound cues for users to look at the dashboard when information changes;
- on a user's display, it will only show notifications (e.g., 'toaster' type alerts in the system tray) as information changes;
- a wide diversity of project data field definitions and usages;
- support for add-on visualizations developed by a dashboard community;
- data caching to reduce the load on the repositories from dashboard clients.

Conclusion and Future Work

This study on WIPDash suggests benefits from providing awareness of work item status. Earlier work on FASTDash [1] showed benefits from providing a team situation awareness display based on code activity. An interesting perspective for future work is to combine information about work items with information about code activity in one visualization [14]. A potential disadvantage of such an approach is that the display gets too busy. However, linking code activity and the state of work items could give the team a solid shared context if it focused simply on what is or has just recently changed. That is the main goal for our next iteration. We intend to create a simple list of recently changed work items, and the people who are actively related to them. In addition, the new WIPDash will have integrated chat and RSS feeds for notifications to the desktop. The ethnographic study of Souza and Redmiles [17] confirms our observations: (a) a need for awareness of who made the changes; (b) a need to peripherally integrate awareness notifications into an existing work items repository, and to link them to code changes in order to keep work items' status better up to date and thus coordinate work more proactively.

In this paper, we have reported on a human-centered design approach to developing a situational awareness dashboard visualization to help software development teams track people and work items. From observations and interviews of development teams we learned about their current work practice and what might be provided to improve situation awareness. We then developed a dashboard for supporting awareness in teams, called WIPDash (Work Item and People Dashboard). We gathered feedback on an initial design from a focus group, which drove the detailed design and the implementation of WIPDash. Finally, we have studied the use of WIPDash in situ with two development teams, and reflected on the observations and data we gathered. While questions remain to be answered, the results from our study provide initial insights about use of a shared display to support team awareness of work item data in software repositories.

References

1. J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson (2007). FASTDash: a visual dashboard for fostering awareness in software teams. In *Proc. CHI '07*, 1313–1322, ACM.
2. M. Bruls, K. Huizing, and J.J. van Wijk (2000). Squarified Treemaps. In *Proc. TCVG '00*, 33-42. IEEE Press.
3. R. DeLine, M. Czerwinski, and G. Robertson (2005). Easing program comprehension by sharing navigation data. In *Proc. VL/HCC '05*, 241-248. IEEE Computer Society.
4. S. C. Eick, J. L. Steffen, E. E. Sumner Jr. (1992). Seesoft - A Tool for Visualizing Line Oriented Software Statistics. In *IEEE Trans. on Software Engineering*, 18 (11), 957-968.
5. J. B. Ellis, S. Wahid, C. Danis, and W. A. Kellogg (2007). Task and social visualization in software development: evaluation of a prototype. In *Proc. CHI '07*, 577–586, ACM.
6. G. Fitzpatrick, P. Marshall, and A. Phillips (2006). CVS integration with notification and chat: lightweight software team collaboration. In *Proc. CSCW '06*, 49-58, ACM.
7. J. Froehlich and P. Dourish (2004). Unifying artifacts and activities in a visual tool for distributed software development teams. In *Proc. ICSE '04*, 387–396, IEEE.
8. C. Gutwin, R. Penner, and K. Schneider (2004). Group Awareness in Distributed Software Development, *Proc. CSCW '04*, 72-81. ACM.
9. C. W. Hill, J. D. Hollan, D. Wroblewski, and T. McCandless (1992). Edit wear and read wear. In *Proc. CHI '92*, 3-9, ACM, New York, NY, USA.
10. A. J. Ko, R. DeLine, and G. Venolia (2007). Information Needs in Collocated Software Development Teams. In *Proc. ICSE '07*, 344-353. IEEE Computer Society.
11. B. A. Olaniran (1996) A Model of Group Satisfaction in Computer Mediated Communication and Face-to-Face Meetings, Behaviour and Information Technology, 15 (1), 24-36. Taylor and Francis.
12. C. O'Reilly, D. Bustard, and P. Morrow (2005). The war room command console: shared visualizations for inclusive team coordination. In *Proc. SoftVis '05*, 57-65, ACM.
13. S. Paul, P. Seetharaman, and K. Ramamurthy (2004) User Satisfaction with System, Decision Process, and Outcome in GDSS Based Meeting: An Experimental Investigation. In *Proc. HICSS '04,* vol. 1. IEEE Press.
14. A. Sarma, D. Redmiles, and A. van der Hoek (2008). Empirical evidence of the benefits of workspace awareness in software configuration management. In *Proc. SIGSOFT '08/FSE-16*, 113-123.
15. K. Schwaber and M. Beedle (2002). *Agile Software Development with Scrum*. Prentice Hall.
16. C. de Souza, J. Froehlich, and P. Dourish (2005). Seeking the source: Software source code as a social and technical artifact. In *Proc. GROUP '05*, 197-206, ACM.
17. C. R. de Souza and D. F. Redmiles (2008). An empirical study of software developers' management of dependencies and changes. In *Proc. ICSE '08*, 241-250, ACM.
18. R. M. Taylor (1989). Situational Awareness Rating Technique (SART) The Development of a Tool for Aircrew System Design. *Proceedings of the Symposium on Situational Awareness in Aerospace Operations*, AGARD-CP-478.
19. S. Teasley, L. Covi, M. S. Krishnan, and J. S. Olson (2000). How does radical collocation help a team succeed? In *Proc. CSCW '00*, 339-346. ACM.