Hidden Messages: Evaluating the Effectiveness of Code Elision in Program Navigation

Matthew Smith and Andy Cockburn
Human-Computer Interaction Lab
Department of Computer Science
University of Canterbury
Christchurch, New Zealand
{mjs171,andy}@cosc.canterbury.ac.nz

October 15, 2001

Abstract. Text elision is a user interface technique that aims to improve the efficiency of navigating through information by allowing regions of text to be 'folded' into and out of the display. Several researchers have argued that elision interfaces are particularly suited to source code editing because they allow programmers to focus on relevant code regions while suppressing the display of irrelevant information. There is, however, a lack of empirical evidence of the technique's effectiveness. This paper presents an empirical evaluation of source code elision using a Java program editor. The evaluation compares a 'flat text' version of this interface with two versions that diminished elided text to levels that were 'just legible' and 'illegible' (extremely small text). Subject performance was recorded in four tasks involving navigation through programs. Results show that programmers were able to complete their tasks more rapidly when using the elision interfaces, particularly in larger program files. Although the subjects indicated a strong preference for the just legible elision interface, their performance was best with the illegible elision system.

Keywords: Text elision, program visualisation, program navigation, programming environments, user interface evaluation

1. Introduction

Computer programs are richly interconnected hypertextual information spaces. To ease the task of creating and maintaining programs, programmers use tools that allow them to rapidly navigate and cross-reference between relevant areas of the source code. Typical facilities provided by software development systems include marking and searching facilities that ease navigation between two or more code regions, split- and multiple-windows that allow more than one code region to

© 2001 *Kluwer Academic Publishers. Printed in the Netherlands.*

be viewed simultaneously, and context-sensitive editing facilities that allow method names to be selected from object variables. Each of these techniques helps to overcome the programmer's problem of needing simultaneous access to more than one region in the source code. Another possible solution—the subject of the evaluation presented in this paper—is to tailor the information displayed in the window so that only information relevant to the programmer's task is shown.

Source code elision (also referred to as 'holophrasting') is a technique that hides certain areas of text based on the structure of a program. It allows programmers to tailor the level of abstraction at which they view or edit code, expanding and contracting detail as appropriate. The aim of eliding code editors is to increase the efficiency of program navigation and to improve the quality of program visualisation by providing a display that focuses on relevant information, without the 'clutter' of information that is irrelevant to the user's task. Eliding interfaces also reduce the amount of window scrolling required to browse between program regions.

Although there are several editors that support code elision, we are unaware of any empirical evaluations of their effectiveness. The aim of the evaluation presented in this paper is to answer the question: 'Does text elision allow programmers to solve realistic program navigation tasks more efficiently?'

Section 2 describes related work on eliding text editors, and introduces the 'Jaba' environment used in our evaluation. The experimental method is described in Section 3, with results and discussion in Sections 4 and 5. Section 6 concludes the paper.

```
16      do hi--; while ((data[hi] > cutval) && (hi != 0));
17      temp = data[lo]; data[lo] = data[hi]; data[hi] = temp;
18    } while (hi > lo);
19    data[hi] = data[lo]; data[lo] = data[right]; data[right] = temp;
20    qsort(data, 0, lo-1);
21    qsort(data, lo+1, right);
22  }
23 }
24
25 public static void main (String[] args) {
26   QuickDemo me = new QuickDemo();
27   boolean valid;
28   for (int i = 0; i < data.length; i++) {
29     System.out.println("Enter an integer: ");
30     valid = false;
31     while (!valid) {
32       try {
33         data[i] = Integer.parseInt(stdIn.readLine());
34         valid = true;
35       }
```

```
1  import java.io.*;
2  public class QuickDemo {
3    ...
7    public void qsort (int[] data, int left, int right) {
8      ...
23   }
24
25   public static void main (String[] args) {
26     QuickDemo me = new QuickDemo();
27     boolean valid;
28     for (int i = 0; i < data.length; i++) {
29       ...
44     }
45     me.qsort(data, 0, data.length-1);
46     for (int i = 0; i < data.length; i++) {
47       System.out.print(data[i] + " ");
48     }
49     System.out.println();
50   }
51 }
```

Figure 1. Twenty lines of a Java class in a normal (left) and elided (right) display.

2. Related Work

Text elision is found in many everyday office information systems. Microsoft's Word and PowerPoint systems, for instance, support 'outline' views that allow users to view documents at tailorable levels of abstraction, expanding and contracting sections, subsections, and so on, as required. These facilities stem from research systems developed in the early 1980s, most of which were primarily designed for editing computer programs. Figure 1 provides a simple example of text elision in source code: on the left there is a normal, non-elided, display of twenty lines of program code, and on the right there is an elided view of the same length. The elided version of the program shows the entire range of the program (from the first line to the last), with low-level regions of the code replaced with ellipses to depict the elision.

The Cornell Program Synthesizer (Teitelbaum 1981) was among the first interfaces to demonstrate text elision. It used syntax-directed editing, based on the grammar of the language, to ensure that programmers created syntactically legal programs. Programmers could expand and contract program constructs to provide successively more detailed or abstract views.

Syntax-directed editing tightly constrains the programmer into specifying programs in a top-down manner, which may not match their preferred approach to program expression. Following the Cornell Program Synthesizer, several systems used less constraining versions of elision based on program block-statements. Examples include Quips (Smith, Barnard & Macleod 1984), Tioga (Teitelman 1985) and EMILY (Barstow, Shrobe & Sandewall 1984). In his cornerstone paper on fish-eye views, Furnas (1986) presented an extension to elision interfaces in which an algorithm, called the 'degree of interest' formula, automatically selected the regions for expansion and contraction based on the user's current focal-point (cursor location in the program) and an 'a-priori interest' value associated with program structural elements. More recently, text elision has been proposed as a mechanism to improve access to the world wide web on mobile devices which have severely limited screen space (Buyukkokten, Garcia-Molina & Paepcke 2000).

The elision systems described above all provide binary mechanisms for elision: text is either shown or it is removed from the display. Scalable fonts allow greater levels of control over the degree to which text is 'removed' from the display, allowing text at the user's focal point to be displayed at full size, while becoming smaller further away. This technique is commonly used in graphical visualisations (for example, see **?**), but it has also been used in text-based menu selection (Bederson 1999) and in a groupware text editor DOME (Cockburn & Weir 1999).

The primary limitation of prior work is that few of these systems have been empirically evaluated. We are unaware of any formal evaluations of elision as a mechanism to support source code navigation.

2.1. The Jaba System

The evaluation of text elision described in the following section used the Jaba program editing environment as a test-bed. Jaba (Cockburn 2001) was designed to experiment with the integration of concepts from Literate Programming (Knuth 1992), fisheye views (Furnas 1986) and hypertext (Conklin 1988). In essence, it is an experimental dynamic and interactive version of Javadoc. A typical Jaba window is shown in Figure 2.

Jaba parses Java classes, extracting structural information concerning the class's methods, constructors, and statement blocks (such as loops, conditionals, and so on). It also parses user defined 'chunks' which can contain any arbitrary series of lines in the program; for instance, a chunk may be a group of related methods, a piece of documentation, or a series of declarations. Chunks can be defined at any level in the program structure, and they can be nested.

All of the parsed structural elements in the class can be elided. For example, in Figure 2, the only expanded element is the user-defined chunk `GuiConstructionMethods` which contains four method definitions: from `make_five_fields` to `make_dice_and_checkboxes`. The contents of each of these methods is elided, and shown in extremely small (illegible) text. Clicking on the name of any method or chunk toggles the elision of its contents (displaying it, then eliding it). Expanded method and chunk names are coloured blue; contracted ones red. Jaba supports many additional hypertextual facilities for linking between classes, but these did not feature in the evaluation and are not further described in this paper. To focus the experiment on navigation within the text editor, the graphical representation of program contents on the left-hand side of Figure 2 was removed.

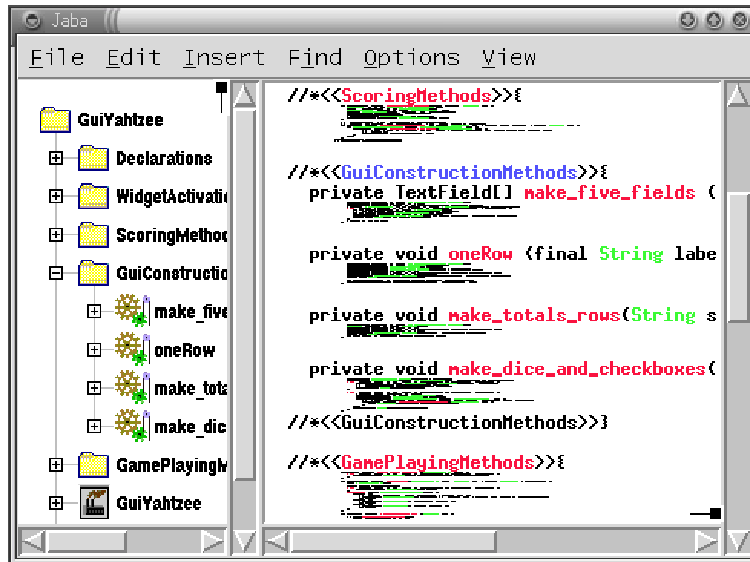

Figure 2. Full Jaba environment.

In the evaluation, three different levels of elision within Jaba were compared, as shown in Figure 3. Apart from the level of elision provided, the three interfaces were identical. The 'flat text' level (Figure 3(a)) provided no elision—it provides a control for comparison with the eliding conditions. The 'legible' level (Figure 3(b)) renders elided text in a font that is just large enough to read. The 'illegible' level (Figure 3(c)) uses a one-point 'greeked' font. The illegible level still provides more contextual information than previous elision systems, several of which simply replaced hidden text with an ellipsis (. . .).

3. Evaluation

The aim of the experiment was to determine whether text elision improves programmer efficiency in carrying out typical source code editing and browsing tasks. We also wished to scrutinise whether different

(a) Flat text.

(b) Legible elision.

(c) Illegible elision.

Figure 3. Jaba source editing windows for the same program with the three levels of elision for suppressed text, showing the same `clickScoreCell` method at the top of each window.

forms of elision were more or less useful as the size of the source code files increased. Finally, we were interested in the programmers' subjective preferences between flat text editors and elision editors.

The experimental design was a two-way analysis of variance (ANOVA) with repeated measures for independent variables 'interface type' (three levels) and 'file size' (two levels). The three levels of interface type were flat text, legible elision and illegible elision, as shown in Figure 3. The two levels of file size were 'small' and 'large', as described below.

In order to focus purely on the support provided by text elision, we used the same base interface for all three interface conditions. Many of Jaba's interface capabilities were removed or disabled, including the navigation tree that allows rapid shortcuts to the methods in the class (see the left-hand side of Figure 2). The potential confounding factors associated with the removal of the graphical navigation tree—which is a common feature in most modern source code editors—are discussed in Section 5. In all experiments the source code editor window was fixed at the same absolute size of 80 characters wide by 40 lines (full size) long.

In selecting Java classes for use in the experiment, we analysed several large Java projects to determine sizes for the 'small' and 'large' levels. We found many classes in the range of 160–200 lines, and chose this for the small level, providing 4–5 screenfuls when fully expanded. These 'small' classes often represented simple business objects. We also found many large classes of around 600–800 lines, especially where Graphical User Interface (GUI) objects were concerned. However, such classes are often very different in composition, and are not always edited manually. We therefore chose a size of 360–400 lines for the 'large' level (9–10 screenfuls), finding many classes representing more complex business objects in or near this range. We avoided classes with

only one or two very large methods, as this would provide an unfair advantage for the elision interfaces in certain tasks. Furthermore, to avoid confusion in tasks requiring a certain method to be found, we removed or changed the name of overloaded methods in each class.

Four different types of tasks were included in the evaluation. Each task involved navigation in a single Java class and was treated as a separate experiment. The design of each experiment and our predicted outcomes are discussed below.

Experiment One: Signature Retrieval

All tasks in this experiment were of the form: "Find the type of the < *x*th> argument to method <method name>."

This required subjects to retrieve information from the signature of a method. This is a common programming activity—when writing code to invoke a method, programmers often want to check the arguments and return type. In the files used in the experiment, the method signatures were always top-level structural elements, ensuring that they were never elided out of the text display.

We expected performance to be significantly faster with the two eliding interfaces (legible and illegible) than with the flat text interface, especially with larger files. The rationale for this prediction is that the eliding interfaces will suppress all methods' details, producing a less 'cluttered' display. Also, because the unneeded information is suppressed, less scrolling is necessary.

Experiment Two: Body Retrieval

In this case, tasks were of the form: "In method <method name>, find the first call to method <called name>."

This required subjects to find the method and inspect its contents. This is a typical debugging task—compilers often report an error at a certain clause of a certain method, and in some systems the programmer must find this manually.

This experiment includes the same method-signature search component as Experiment One. For this part of the task, we expected the elision interfaces to be significantly faster than flat text. Having found the required method, the subjects needed to find specific information within the method's detail. This second component of the task raises different predictions for the three interfaces. With the illegible elision interface, subjects must click on the method-signature to expand its contents. We therefore reasoned that for small files, this would cost similar amounts of time to that gained by a faster initial search. For large files, we predicted that the initial time saving would be greater, resulting in better overall performance with the illegible elision interface. With the legible elision interface we were interested to observe subject behaviour. Although the elided text is just large enough to read, we were unsure whether or not subjects would prefer to first expand the method to full size, thus adversely affecting overall performance.

Experiment Three: Combination of Body Search and Signature Retrieval

Tasks in this experiment were all of the form: "In method <method name>, find the return type of the method that is called last".

This required subjects to find a method signature, inspect its method details and retrieve another method signature within the class. It is equivalent to Experiment Two with an additional task from Experiment One. Subjects were instructed not to infer the return type from the method call, forcing them to perform the second search.

The task is indicative of navigation in source code, where the programmer follows a series of references and pointers until they find the desired information.

The scrolling demands of this task are relatively high. We therefore predicted that the illegible elision interface—which produces the least cluttered display and therefore requires the least scrolling—would allow the most rapid task completion. As for Experiment Two, we were unsure whether users of the legible elision interface would chose to read the small text or fully expand the method details, and we were therefore not confident in predicting its efficiency.

Experiment Four: Visualisation Search

The final experiment involved answering the question: “Determine the longest method in the class”.

The ability to better visualise the structure of a program is one of the key claims of elision interfaces. Although this task is artificial, we included it in order to test this claim.

To avoid subjects needing to count the number of lines, classes were chosen such that the largest method was clear from a visual scan of the code. Subjects were told that this was the case.

We predicted that the elision interfaces would allow more rapid completion of this task because they allow a greater fraction of the source code to be viewed within each window area and consequently require less scrolling. Further extending this argument, we predicted that the illegible elision interface would out-perform legible elision.

Subject Details and Procedure

The twelve subjects were all volunteer postgraduate Computer Science students. Although the number of subjects is relatively low, the repeated measures experimental design gives a relatively high degree of statistical power. All subjects had several years of experience with Java syntax. Each participant's experiment lasted approximately twenty-five minutes, including training time. Training involved explaining and demonstrating each of the three interfaces, then allowing subjects to familiarise themselves with each by navigating in a sample file.

Each of the four experiments required subjects to perform the same navigation task using all three interfaces and both file sizes, giving a total of 24 tasks. To control possible learning effects, a different class was used for each task (12 classes per file size), and the order subjects used each interface was rotated between subjects.

For comparability within experiments, it was necessary to choose similar method locations in the different files. For example, in Experiment One, all six methods chosen for retrieval were approximately 40 lines from the end of their respective classes. We were concerned that subjects might recognise this consistency and alter their behaviour accordingly. To control this, we randomised the sequence of the 24 tasks, so that the four experiments were actually interspersed.

Each task was presented to the subject in a command-line control interface. Once the subject had read the task and confirmed that they understood it, they pressed a key to begin. The control interface then opened the appropriate version of Jaba, with the appropriate class displayed. The timing was performed by the control interface, and began once the file had been fully loaded in Jaba, avoiding bias due to Jaba's parsing times. Once the subject had completed the task, they clicked

Table I. Mean (standard deviation) times in seconds for each experiment, across each level of interface type and file size.

	Experiment 1		**Experiment 2**		**Experiment 3**		**Experiment 4**	
	Small	**Large**	**Small**	**Large**	**Small**	**Large**	**Small**	**Large**
Flat	8.7 (1.5)	12.8 (2.4)	7.9 (2.9)	13.0 (3.7)	12.3 (1.6)	25.5 (5.3)	8.0 (2.5)	13.4 (5.1)
Legible	7.4 (1.4)	11.2 (2.5)	9.4 (2.8)	15.3 (5.9)	12.6 (2.2)	21.9 (4.2)	7.4 (2.2)	12.1 (3.0)
Illegible	6.6 (1.6)	9.9 (2.2)	8.5 (1.6)	11.6 (2.8)	13.2 (2.9)	18.2 (4.5)	7.5 (2.9)	12.3 (3.5)

a 'Done' button at the bottom of the control interface, at which point the timing ceased.

After each task, subjects were asked to respond to a 5-point Likert scale question: "The <flat/legible/illegible> interface was effective for the task" (1=disagree, 5=agree). Subjects were asked to provide comments after training, after each task, and at the end of the experiment.

4. Results

Overall, the subjects had few problems with the experimental method and with using the three interfaces. The tasks were solved quickly, with a mean task completion time of 12.0 seconds (standard deviation θ=5.5) across the 288 task pool (twelve subjects, four experiments, three interfaces and two file types).

Performance data for the four experiments are summarised in Table I. Subjective responses to the Likert-scale questions are summarised in Table II and Figure 4. The results of each individual experiment are described below.

Table II. Mean (standard deviation) responses for each experiment to the 5-point Likert scale question: "The interface was effective for the task". Ticks indicate a significant difference at the .05 level using Friedman Tests (df = 2, N = 12 in each case).

	Experiment 1		Experiment 2		Experiment 3		Experiment 4	
	Small	**Large**	**Small**	**Large**	**Small**	**Large**	**Small**	**Large**
Flat	2.8 (0.8)	2.5 (0.8)	3.3 (1.2)	3.0 (1.0)	3.1 (0.8)	2.5 (1.0)	3.0 (0.8)	2.7 (0.9)
Legible	3.3 (1.1)	3.0 (1.0)	3.6 (1.1)	3.3 (1.1)	3.3 (1.1)	3.6 (1.2)	3.7 (0.9)	4.0 (1.0)
Illegible	3.6 (1.1)	3.6 (1.0)	3.3 (1.0)	3.1 (0.9)	3.2 (0.9)	3.3 (1.1)	3.2 (1.0)	3.3 (1.1)
χ^2_r	5.5	7.5	0.9	0.8	0.8	7.3	3.8	9.5
p	0.06	0.02	0.64	0.67	0.67	0.03	0.15	0.01
Significant ?	8	4	8	8	8	4	8	4

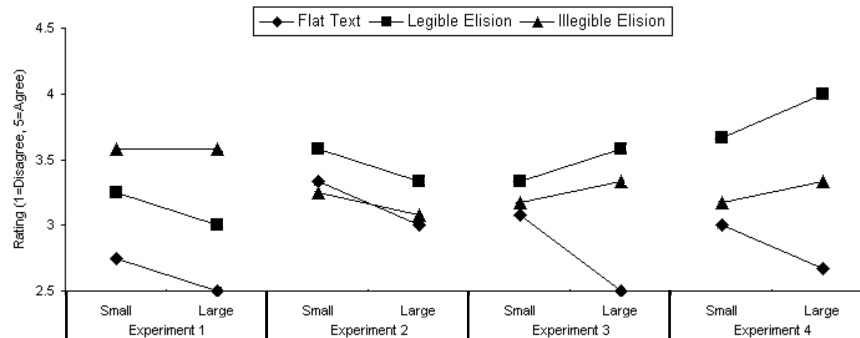

Figure 4. Likert responses for each experiment, across each level of interface type and file size.

Experiment One: Signature Retrieval

Experiment One compared the time taken to find a named method using the three interfaces. We predicted that elision interfaces would allow better performance than the flat text interface, and that illegible elision would out-perform legible elision.

The mean task times for small and large files were 7.58 (θ 1.7) and 11.28 (θ 2.58) seconds, providing a reliable difference ($F(1,11) = 95.5$, $p < .001$). This unsurprising result indicates that the subjects took longer

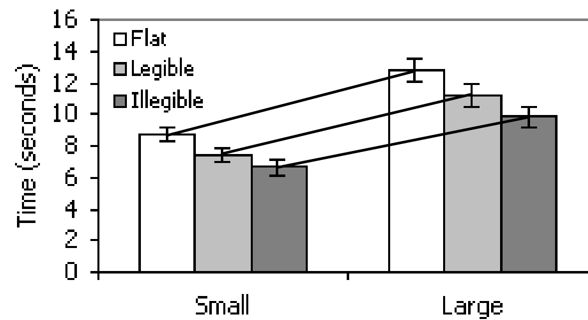

Figure 5. Experiment One: Mean task completion times and standard errors (above and below the mean).

to browse large files than short ones due to the additional scrolling required.

The means for the the flat, legible and illegible interfaces were significantly different at 10.74 (θ 2.8), 9.30 (θ 2.8) and 8.24 (θ 2.5) seconds respectively (F(2,22) = 11.6, $p < .001$). As shown in Figure 5, illegible elision performed best overall. This confirms our prediction that for retrieval from a method signature (a non-elided element), the suppression of irrelevant detail increases efficiency. The legible elision interface was not substantially slower than the illegible elision one.

Subjective responses to the Likert-scale question "The interface was effective for the task" reflected the performance measures. The subjects rated the illegible elision interface as more effective than the legible elision interface, with the flat text interface receiving the worst rating. These results are summarised in Table II and Figure 4.

There was no significant interaction between interface type and file size (F(2,22) = 0.33, p = .72). The absence of an interaction is clear in Figure 5, which shows that the mean task completion times degraded between the small and large file sizes at a similar rate for the three

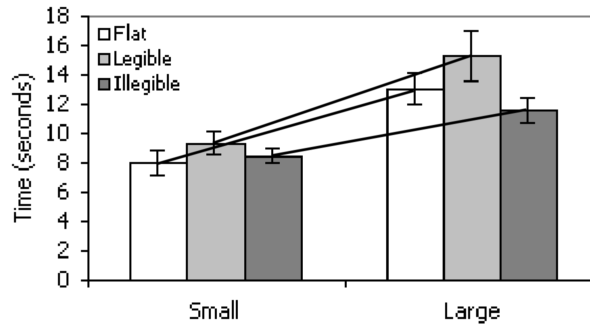

Figure 6. Experiment Two: Mean task completion times and standard errors.

interfaces. We were somewhat surprised by this. We had predicted that the benefits of the elision interfaces would become larger (in comparison to the flat text interface) as the file size increased.

The subjects' comments provided an explanation for the absence of a reliable interaction between factors 'interface' and 'file-size'. In Jaba, method signatures were the only program elements coloured red. When using the flat text interface, the subjects made heavy use of the red text to allow them to scroll rapidly to the method signatures, while ignoring all non-red detail. As a result, although the elision interfaces were reliably faster than flat text, their benefits did not increase relative to the flat text interface with larger files.

Experiment Two: Body Retrieval

Experiment Two compared the times taken to find a specific method call within the body of a named method. We predicted no difference between elision and flat text interfaces for small files, but suspected that elision interfaces would out-perform flat text in large files.

There was a significant difference between the mean task times for small and large files of 8.59 (θ 2.5) and 13.3 (θ 4.5) seconds (F(1,11) =

56.6, $p < .001$). Again, this is a result of tasks with longer files requiring more scrolling.

The main effect for interface type was not significant ($F(2,22) = 1.73$, $p = .2$), with mean times of 10.5 (θ 4.2), 12.32 (θ 5.4) and 10.02 (θ 2.7) seconds for the flat, legible and illegible interfaces. Furthermore, the interaction between file size and interface type was not significant ($F(2,22) = 1.95$, $p = .17$). Although these results do not provide a statistically reliable confirmation of our predicted results, the relative performances of the flat text and illegible interfaces with the small and large file sizes are as expected. Figure 6 indicates that for large files, the benefits of illegible elision appear to be realised in comparison to the negligible difference between the flat and illegible conditions for small files.

Figure 6 also shows the surprising result that the legible elision interface provided a slower mean task completion time than the flat text interface, for both file sizes (although this is not a statistically reliable effect). We observed that when using legible elision, ten out of twelve subjects did not expand the suppressed text. Instead, they choose to scan the small but just legible text to find the appropriate item. This had an adverse affect on their performance, as the items were much harder to read in the smaller font, and subjects reported the need to 'squint'. Only two of those ten subjects changed their behaviour after experiencing this problem. Interestingly, the subjects' Likert ratings for the three interfaces showed a small, but not significantly reliable, preference for legible elision with both file sizes (see Table II). Several subjects also commented that the legible interface provided a nice balance between the other two.

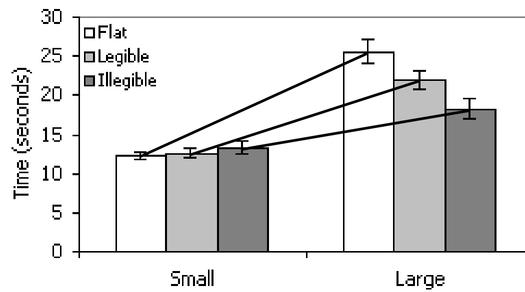

Figure 7. Experiment Three: Mean task completion times and standard errors.

Experiment Three: Combination

Experiment Three combined the tasks of Experiments One and Two, providing an indirect search through methods: first finding a method signature, then searching its body for a specific method invocation, and then finding its method signature. We predicted that the illegible elision interface would allow the most rapid task completion, and we were interested to see how the subjects would use the legible elision interface.

The main effect for file size was again significant ($F(1,11) = 88.9$, $p < .001$), but largely irrelevant as before. Mean task times for the flat, legible and illegible interfaces were 18.9 (θ 7.8), 17.3 (θ 5.8) and 15.7 (θ 4.5) seconds, providing a reliable difference ($F(2,22) = 7.8$, $p < .01$). Again, the illegible elision interface allowed the most rapid task completion.

When browsing small files, the mean task completion times across the three interfaces were similar. However, as shown in Figure 7, the benefits of the elision interfaces become marked when solving tasks in larger files, particularly with the illegible elision interface. This relative performance improvement with the illegible elision interface resulted

in a significant interaction between file size and interface type ($F(2,22) = 11.8$, $p < .001$). As predicted, this reflects the benefits of illegible elision when more extensive searching is required.

Table II shows a reliable difference between the subjects' ratings of the effectiveness of the three interfaces when navigating through large files, but not for small files. For both small and large files, the subjects rated the legible elision as most effective, even though it provided the worst mean performance. Subjects again mentioned the balance it provided between the other two interfaces. They also reported less trouble with 'squinting' at the just legible suppressed text than in Experiment Two. The most likely explanation for this is that unlike Experiment Two, this experiment did not require a specific method name to be found (rather, just the last method invocation in the method). From Figure 4, it is interesting to note that ratings for the flat text interface decreased dramatically with larger files, while ratings for both elision interfaces increased.

Experiment Four: Visualisation Search

Experiment Four compared the subjects' ability to find the largest method in a class file using the three interfaces. We predicted that the elision interfaces would allow more rapid completion of this task.

The mean task times for small and large files of 7.6 (θ 2.5) and 12.6 (θ 3.9) seconds were significantly different ($F(1,11) = 75.7$, $p < .001$).

Contrary to our prediction, there was no significant difference between interface types, with means of 10.7 (θ 4.8), 9.7 (θ 3.5), and 9.9 (θ 4.0) seconds for the flat, legible and illegible interfaces ($F(2,22) = 1.1$, $p = .36$). There was also no significant interaction between file size and interface type ($F(2,22) = 0.38$, $p = .7$).

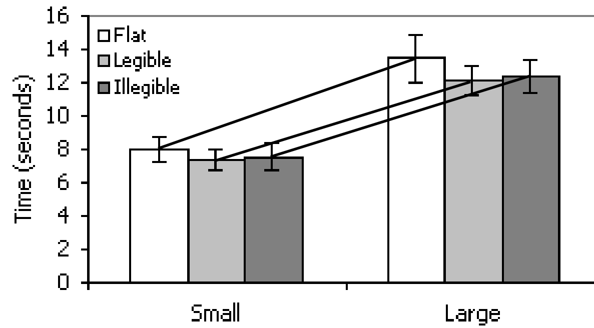

Figure 8. Experiment Four: Mean task completion times and standard errors.

We were very surprised by the similarity of performance across the interfaces (Figure 8). Even though the mean task completion times with the elision interfaces were lower, the differences were very small. Subjects again commented that with the flat text interface, they scrolled more rapidly, trusting their eyes to identify large blocks of text. In many cases, subjects did not need to compare similar methods, and could determine the answer from only a single scan.

Despite these similarities, the mean ratings for effectiveness again indicated a strong preference for the legible elision interface (Table II). As for the previous experiment, ratings for flat text reduced with large files, but increased for both elision interfaces (Figure 4).

5. Discussion

When first shown the illegible elision interface during training, several of the subjects mentioned that the interface was 'neat' or 'cool'. The performance results show that it is more than this—it can yield statisti-

cally significant performance improvements for source code navigation tasks.

To summarise the results, in all of the large file navigation tasks (Experiments One to Three), the illegible elision interface provided the most rapid mean task completion time. In two of these tasks, the flat text interface provided the slowest mean task completion time. The legible elision interface was less successful than the illegible elision interface in terms of task performance, yet it received the highest 'effectiveness' ratings from the subjects in three of the four experiments. Despite comments that the legible interface provided a good compromise between illegible elision and flat text, its primary limitation appeared to be that it encouraged users to solve tasks by 'squinting' at the tiny source code rather than expanding it to normal flat text.

5.1. Confounding factors and further work

Although the results are promising, there were several limitations in the study that we plan to address in further work.

In designing a carefully controlled experiment that focused solely on the efficiencies of elision interfaces, we excluded code browsing features that are normally available in commercial software development environments. In particular, we removed the graphical tree representation of the source code structure.

It remains unclear how the elision interfaces would have compared to flat text if the users had been able to use their normal techniques—such as graphical tree browsers—for navigating through the source code. We strongly suspect that many programmers frequently resort to scrolling within the text editor, in the manner tested by our experiments. However, we are also confident that many programmers will make heavy use

of tree browsers when available. In future work we wish to scrutinise programmer behaviour with and without elision interfaces when the interfaces support alternative schemes for moving through the code. Our prediction is that the benefits of elision interfaces will still be apparent, because we suspect that using graphical code browsers will raise cognitive, perceptual, and motor-coordination requirements due to the user switching their attention and cursor between the source code window pane and the graphical window pane.

Another area for further work is in evaluating the effectiveness of elision interfaces for working with other structured documents such as web pages. The successive display of increasingly detailed information provided by elision interfaces seems to be a particularly attractive solution to the problem of browsing web pages on small displays such as personal digital assistants (PDAs). Buyukkokten et al. (2000), for example, discuss a PDA interface for browsing web-pages.

6. Conclusions

Text elision interfaces provide ‘folding’ views of structured documents, allowing users to selectively reveal successive layers of detail within particular document regions. Several researchers have argued that elision interfaces are particularly suited to source code editors, because they allow programmers to focus on relevant detail while minimising the display of information that is superfluous to their task. It has also been argued that they can make navigation through source code more efficient.

Although several source code editors have supported text elision, we are unaware of prior research that empirically investigates its effectiveness.

The evaluation reported in this paper compared the performance and preferences of programmers when navigating through Java source code using three interfaces that differed only in their support for text elision. The first interface provided a normal 'flat text' view of the source code, with no support for text elision. The other two interfaces supported 'illegible' and 'legible' elision facilities, which diminished elided text to an extremely small and just legible degree respectively.

Results of the evaluation showed that users were able to complete navigation tasks more quickly with the eliding interfaces, particularly when working with larger source code files. Although the programmers rated the 'legible' type of elision more highly, their performance was reliably better when using the illegible elision interface.

Future work will focus on further evaluating elision capabilities when programmers are free to choose between a wide range of different tools for source code navigation.

Acknowledgements

This research is supported by a New Zealand Royal Society Marsden Grant.

References

Barstow, D., Shrobe, H. & Sandewall, E., eds (1984), *Interactive Programming Environments*, McGraw-Hill.

Bederson, B. (1999), Fisheye menus, Technical Report CS-TR-4138, UMIACS-TR-2000-31, HCI Lab, University of Maryland. http://www.cs.umd.edu/hcil/fisheyemenu/.

Buyukkokten, O., Garcia-Molina, H. & Paepcke, A. (2000), Seeing the Whole in Parts: Text Summarization for Web Browsing on Handheld Devices, *in* 'Proceedings of the Tenth International World-Wide Web Conference, 2000.'.
URL: *citeseer.nj.nec.com/buyukkokten00seeing.html*

Cockburn, A. (2001), 'Supporting Tailorable Program Visualisation Through Literate Programming and Fisheye Views', *Information and Software Technology* **43**(13), 745–758.

Cockburn, A. & Weir, P. (1999), 'An investigation of groupware support for collaborative awareness through distortion-oriented views', *International Journal of Human Computer Interaction* **11**(3), 231–255.

Conklin, J. (1988), Hypertext: An introduction and survey, *in* I. Greif, ed., 'Computer Supported Cooperative Work: A Book of Readings', Morgan Kaufmann.

Furnas, G. (1986), Generalized fisheye views, *in* 'Human Factors in Computing Systems III. Proceedings of the CHI'86 conference.', Amsterdam; North Holland/ACM, pp. 16–23.

Knuth, D. (1992), *Literate Programming*, Stanford, California: Center for the Study of Language and Information. CSLI Lecture Notes, no. 27.

Leung, Y. & Apperley, M. (1994), 'A Review and Taxonomy of Distortion-Oriented Presentation Techniques', *ACM Transactions on Computer Human Interaction* **1**(2), 126–160.

Sarkar, M. & Brown, M. (1992), Graphical fisheye views of graphs, *in* 'Proceedings of CHI'92 Conference on Human Factors in Computing Systems Monterey, May 3–7', Addison-Wesley, pp. 83–91.

Schaffer, D., Zuo, Z., Greenberg, S., Bartram, L., Dill, J., Dubs, S. & Roseman, M. (1996), 'Navigating Hierarchically Clustered Networks through Fisheye and Full-Zoom Methods', *ACM Transactions on Computer Human Interaction* **3**(2), 162–188.

Smith, S., Barnard, D. & Macleod, I. (1984), 'Holophrasted displays in an interactive environment', *International Journal of Man-Machine Studies* **20**(4), 343–355.

Teitelbaum, T. (1981), 'The Cornell Program Synthesizer: A Syntax-Directed Programming Environment', *Communications of the ACM* **24**(9), 563–573.

Teitelman, W. (1985), 'A tour through Cedar', *IEEE Transactions on Software Engineering* **11**(3), 285–302.