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

Andy Cockburn and Matthew Smith

Human-Computer Interaction Lab
Department of Computer Science
University of Canterbury
Christchurch, New Zealand
`andy,mjs171@cosc.canterbury.ac.nz`

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. Elision features are now appearing in commercial systems for software development. There is, however, a lack of empirical evidence of the technique's efficiency. This paper presents an empirical evaluation of source code elision using a Java program editor. The evaluation compared a normal 'flat text' editor with two versions that diminished elided text to levels that were 'just legible' and 'illegible'. 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 several participants indicated a preference for the 'just legible' elision interface, performance was best with illegible elision.

Key words: Text elision, program navigation and visualisation, fisheye views, scrolling, 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 be viewed simultaneously, and context-sensitive editing facilities that allow method names to be selected from object variables. Techniques such as these help 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 text editor so that only information relevant to the programmer's task is shown.

Source code elision is a technique that hides or diminishes 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. Eliding code editors aim to improve the quality of program navigation by providing a display that focuses on relevant information, without the 'clutter' of irrelevant information. They also have the potential to reduce the amount of window scrolling required in program browsing.

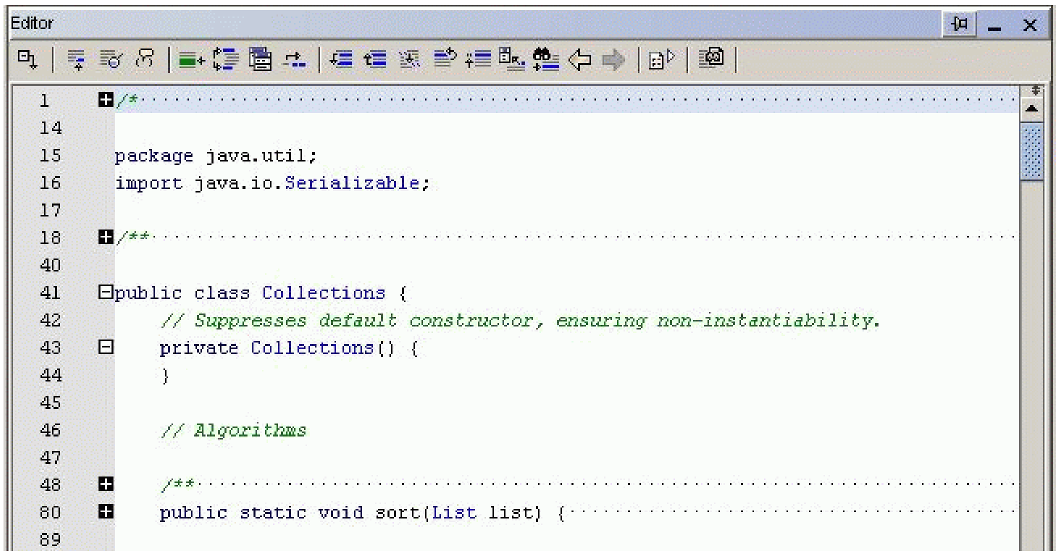
Although several editors support code elision (for example, the recently released version of TogetherControlCenter 6.0 [1], shown in Figure 1), we are unaware of any empirical evaluations of its efficiency. The aim of the evaluation presented in this paper is to answer the question: 'Does text elision allow programmers to solve program navigation tasks more efficiently?'

Section 2 describes related work on improving program navigation, including research on program typography and style, and on eliding interfaces. Section 3 describes the 'Jaba' environment used in our evaluation, and details the theoretical benefits and costs of navigating through programs with eliding editors. The experimental method is described in Section 4, with results and discussion in Sections 5 and 6. Section 7 concludes the paper.

2 Related Work

Spence (1999) defines navigation as "the creation and interpretation of an internal (mental) model, and its component activities are browsing, modelling, interpretation and the formulation of browsing strategy". Using this definition, navigation and comprehension of information spaces (such as computer programs) are strongly related. Although our experiment focuses purely on how quickly users can navigate from one region of a program to another, there has been extensive prior research on program comprehension, program navigation, and on the relationship between them. This section reviews this work and summarises prior research on eliding interfaces. Related work on

[1] www.togethersoft.com



```
/*·····································

package java.util;
import java.io.Serializable;

/**···································

public class Collections {
    // Suppresses default constructor, ensuring non-instantiability.
    private Collections() {
    }

    // Algorithms

    /**·······························
    public static void sort(List list) {···························
```

Fig. 1. Method and comment elision supported by Together ControlCenter 6.0. Clicking on the + and - symbols to the left of methods or comments expands or collapses the display of the method body or comments.

alternative zooming and scrolling mechanisms that may be able to enhance program navigation is briefly discussed later in Section 6.2.

2.1 Typography and Style in Programming

Several studies have shown that program comprehension can be enhanced by using appropriate means for formatting and presenting code. Miara, Musselman, Navarro & Shneiderman (1983), for example, confirmed the consensus view that indented programs are 'better' by empirically showing that it aids comprehension for both novice and experienced programmers. Beyond simple indentation, Baecker (1988) and Baecker & Marcus (1990) present a wide range of schemes for improving the visual presentation of source code, including the use of colour and variable font sizes. Baecker (1988) also provides an experimental validation of the guidelines by showing that comprehension scores are improved when C program code is formatted using their 'SEE' tool. Similarly, Oman & Cook (1990) provide empirical evidence that the typographic principles implemented in their 'book' source code format improve programmers' ability to understand and maintain software. Gellenbeck & Cook (1991) isolated three specific features for examination in their study of program comprehension: meaningful versus nonsense function names, the presence versus absence of function comments, and large versus normal font for function headers. They found that all three features aided comprehension, but that larger fonts for function headers had only a marginal impact. Finally, beyond statically formatting source code to display its structure and semantics, Tapp & Kazman (1994) investigated whether larger fonts and colouring are effective in revealing information associated with dynamic program behaviour. In a code optimisation task, colour and font size were used to encode the number of times statements were executed, and in a code coverage task, colour and font size were used to encode whether or not each statement had been executed. Their results showed that both colour and font size significantly improved performance (measured over several dependent variables) compared to an interface without either cue. Furthermore, colour seemed to provide greater improvements than font size.

Although far more than a mechanism for formatting source code, literate programming (Knuth 1984, Knuth 1992) is another technique that aims to ease the exposition of software so that it is easier to navigate and more comprehensible for the author and subsequent readers. It is an elegant technique that allows programmers to design, document, and construct their programs in whatever order best aids human understanding. Literate programs consists of 'cognitive chunks' of code and documentation, which need not correspond to the programming language's syntactic constructs. For example, a loop chunk may contain a set of variable assignments that establish pre- and post-conditions as well as the code for the loop itself. Literate programs are 'tangled' to produce code that is ready for processing by a compiler or interpreter, or 'woven' to produce pretty-printed documentation that aids navigation through extensive cross-referencing and indexing of program elements.

There have been several evaluations of literate programming as an educational tool, with Soloway (1986) arguing that learning to program involves not only learning to build computer solutions, but also to construct explanations. Thimbleby (1986) reported that student projects written as literate programs had higher quality documentation that was better integrated with the code. Shum & Cook (1994) compared sixteen student programming assignments, half written with and half without support of a literate programming tool called AOPS (Shum & Cook 1993). Results showed that literate programming promoted more and higher quality documentation. These evaluations all investigate the amount and the quality of documentation created by program authors. We are unaware of prior evaluations of the degree to which the cross-referenced and indexed 'woven' versions of literate programs aid program navigation.

2.2 Elision Interfaces

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 by expanding and contracting sections, subsections, and so on. Figure 1 provides a simple example of text elision in source code: the line numbers on the left shown that, for instance, lines 2 to 13 are hidden from view.

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, however, tightly constrains the programmer into specifying programs in a top-down manner, which may not match the programmer's preference. Several systems have used less constraining versions of elision based on program block-statements. Examples include Quips (Smith, Barnard & Macleod 1984), Tioga (Teitelman 1985) and EMILY (Hansen 1984). To our knowledge, none of these systems has been formally evaluated in user studies, so the efficiency of elision remains unclear. In describing 'fisheye views', Furnas (1986) extended the elision concept by using an algorithm, called the 'degree of interest' formula, to automatically select which program regions are elided based on the user's focal-point. Although a study indicated that users were able to navigate through hierarchical file structures more efficiently with fisheye views, their efficiency with program code was not evaluated.

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. Similar 'distortion-oriented' techniques are commonly used in graphical visualisations. Sarkar & Brown (1992) describe the most commonly used fisheye distortion transformation, and Leung & Apperley (1994) provide a taxonomy and review of graphical distortion-oriented techniques.

Research on the efficiency of graphical fisheyes has produced divergent results. In tasks involving navigation through hierarchical information, Schaffer, Zuo, Greenberg, Bartram, Dill, Dubs & Roseman (1996) show a significant performance advantage for fisheyes over normal 'full-zoom' interfaces, while Lamping, Rao & Pirolli (1995) showed no significant advantage over a normal scrolling window. In a programming task, Griswold, Chen, Bowdidge, Cabaniss, Nguyen & Morgenthaler (1997) evaluated their 'star diagram' interface, which provides a tree-diagram of the computations on data structures. The star diagram allows programmers to selectively remove unneeded nodes from the tree display based on a variety of properties including the syntactic class of the nodes, the depth of the tree, and string matches on their labels. Observations revealed that elision was used extensively to control the number of items shown in the visualisation, but performance measures were not subjected to rigorous analysis.

There are few evaluations of textual elision, and none (that we know of) in a programming context. Buyukkokten, Garcia-Molina & Paepcke (2000) compared interfaces with and without text elision for access to the world wide web on mobile devices, and showed that users were three to four times faster at answering questions about web page content when using elision. Bederson (2000) proposed 'fisheye menus' to reduce the time to select items from long menus. With fisheye menus items close to the cursor are shown at a normal font while those further away are shown at small font sizes. Their users were faster at selecting items with fisheye menus than with two forms of scrolling menus, but slower than with an alphabet-based cascading menu.

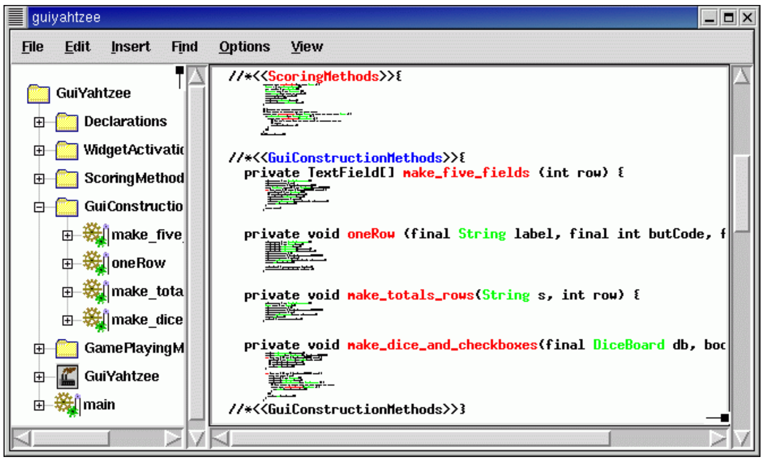
The research most closely associated with the evaluation reported in this paper is by Hornbaek & Frokjaer (2001). Their experiment compared how well users could write and comprehend essays using three text editors: a 'normal' linear editor, a fisheye editor, and an overview+detail editor. The fisheye editor allowed regions of text to be elided to an illegible (or 'greeked') font, and the overview+detail editor provided a miniaturised display of the entire document to the left of a normal text edit window. The miniaturised overview display could be used to navigate directly to regions in the document. Their results showed that users read documents most quickly using fisheyes, but that their writing quality and comprehension was highest with the overview+detail interface.

3 The Jaba System

The evaluation of text elision described in the following section used the Jaba program editing environment as a test platform. Jaba (Cockburn 2001) was designed to experiment with the integration of concepts from Literate Programming (Section 2.1), fisheye views (Section 2.2) and hypertext (Conklin 1988). In essence, it is an experimental dynamic and interactive version of Javadoc (Friendly 1995). A typical Jaba window is shown in Figure 2.

Jaba parses Java classes, extracting structural information concerning methods, constructors, statement blocks (such as loops, conditionals, and so on), and user defined 'chunks'. Normally, all of the parsed structural elements can be elided, but in the evaluation, only method elision was supported. In Figure 2, for example, the only expanded element is the user-defined chunk `GuiConstructionMethods` which contains four method definitions: from `make_five_fields` to `make_dice_and_checkboxes`. Clicking on any method name toggles the elision of its contents. Expanded method 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.

In the evaluation, three different levels of elision within Jaba were compared,



```
//*<<ScoringMethods>>{


//*<<GuiConstructionMethods>>{
  private TextField[] make_five_fields (int row) {

  private void oneRow (final String label, final int butCode, f

  private void make_totals_rows(String s, int row) {

  private void make_dice_and_checkboxes(final DiceBoard db, boc

//*<<GuiConstructionMethods>>}
```

Fig. 2. Full Jaba environment.

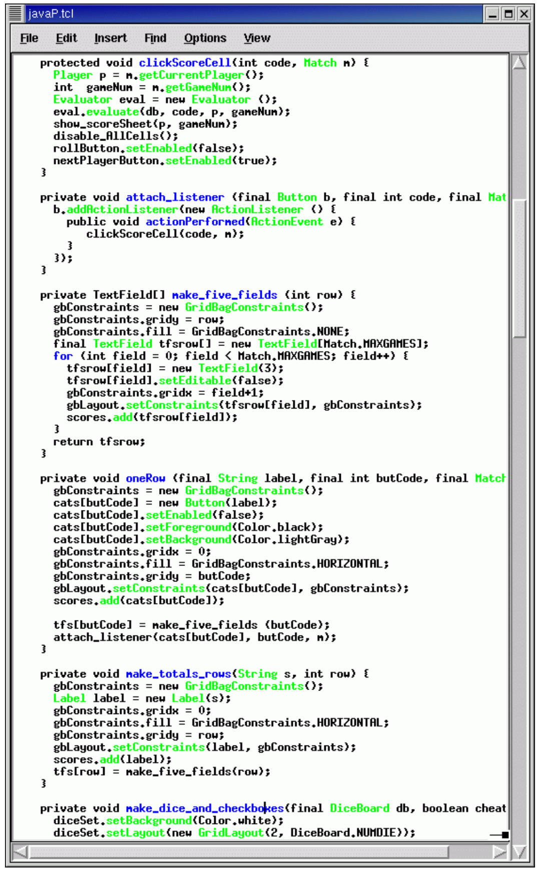
as shown in Figure 3. Apart from the level of elision, the three interfaces were identical. The 'flat text' level (Figure 3(a)) provides a non-eliding control for comparison with the eliding conditions. The 'legible' level (Figure 3(b)) renders elided text in a font that is just large enough to read. The 'illegible' level (Figure 3(c)) uses a one-point greeked font. The miniaturised depiction of method contents displayed by the illegible interface is intended to convey contextual information about each method, such as its length, contents and format (for example, the structure of the contained code as indicated by indentation).

3.1 *Theoretical pros and cons of elision for program navigation*

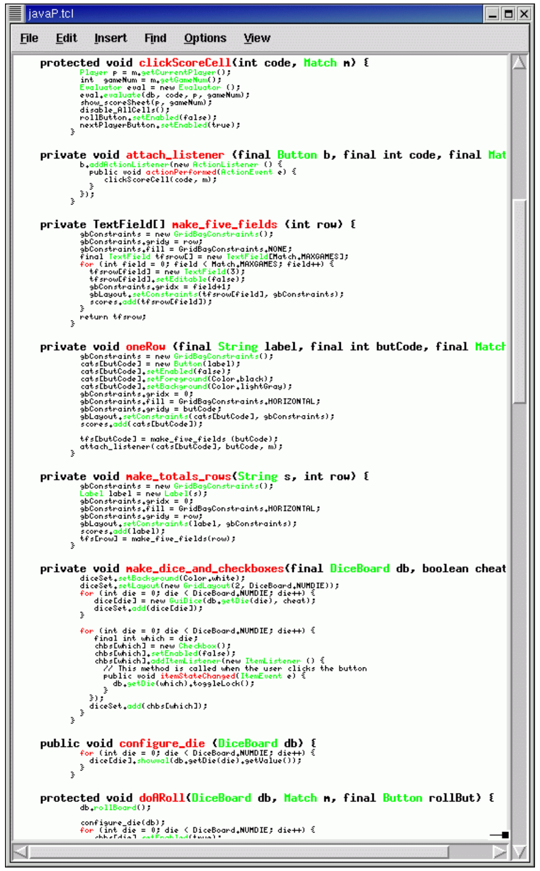
This section discusses the theoretical benefits and costs of using elision capabilities in program editors. Both cognitive and motor issues are discussed.

Programmers need effective editors for working with the low-level details of program code, but they also need tools that provide abstract views of program units. Common tools for abstract views include UML class diagrams and Javadoc. There are cognitive and motor costs associated with moving between the abstract-level tools and code editors. For example, moving between the Javadoc presentation of a class in a web-browser and the underlying code is likely to require several window management actions, as well as (possibly) loading the class into a text editor. Even if the different views are supported by subcomponents of the same system, and the target code is scrolled into view, there is still a degree of re-orientation in moving attention and cursor between windows. Analytical tools such as GOMS/KLM (Card, Moran & Newell 1983) could be used to model these actions.

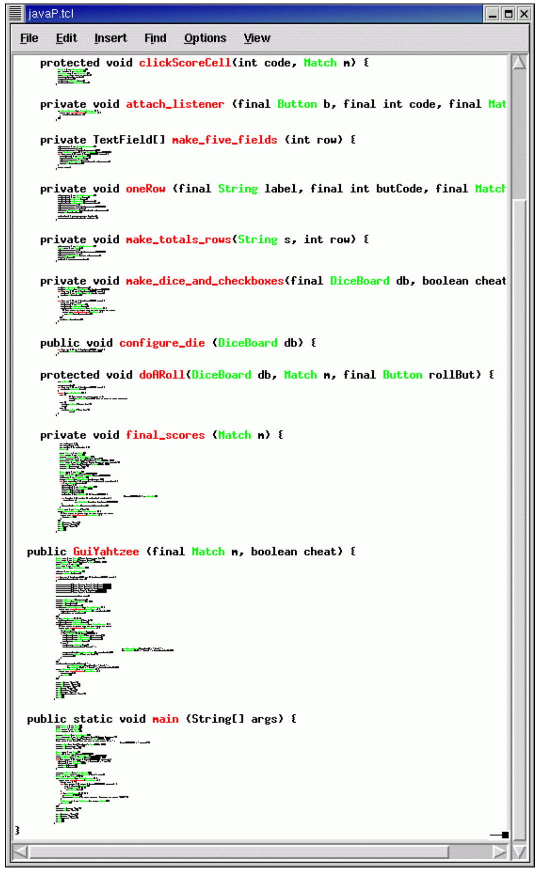
Elision systems may reduce these costs by supporting both abstract (elided)



```
  protected void clickScoreCell(int code, Match m) {
    Player p = m.getCurrentPlayer();
    int  gameNum = m.getGameNum();
    Evaluator eval = new Evaluator ();
    eval.evaluate(db, code, p, gameNum);
    show_scoreSheet(p, gameNum);
    disable_AllCells();
    rollButton.setEnabled(false);
    nextPlayerButton.setEnabled(true);
  }

  private void attach_listener (final Button b, final int code, final Mat
    b.addActionListener(new ActionListener () {
      public void actionPerformed(ActionEvent e) {
         clickScoreCell(code, m);
      }
    });
  }

  private TextField[] make_five_fields (int row) {
    gbConstraints = new GridBagConstraints();
    gbConstraints.gridy = row;
    gbConstraints.fill = GridBagConstraints.NONE;
    final TextField tfsrow[] = new TextField[Match.MAXGAMES];
    for (int field = 0; field < Match.MAXGAMES; field++) {
      tfsrow[field] = new TextField(3);
      tfsrow[field].setEditable(false);
      gbConstraints.gridx = field+1;
      gbLayout.setConstraints(tfsrow[field], gbConstraints);
      scores.add(tfsrow[field]);
    }
    return tfsrow;
  }

  private void oneRow (final String label, final int butCode, final Match
    gbConstraints = new GridBagConstraints();
    cats[butCode] = new Button(label);
    cats[butCode].setEnabled(false);
    cats[butCode].setForeground(Color.black);
    cats[butCode].setBackground(Color.lightGray);
    gbConstraints.gridx = 0;
    gbConstraints.fill = GridBagConstraints.HORIZONTAL;
    gbConstraints.gridy = butCode;
    gbLayout.setConstraints(cats[butCode], gbConstraints);
    scores.add(cats[butCode]);

    tfs[butCode] = make_five_fields (butCode);
    attach_listener(cats[butCode], butCode, m);
  }

  private void make_totals_rows(String s, int row) {
    gbConstraints = new GridBagConstraints();
    Label label = new Label(s);
    gbConstraints.gridx = 0;
    gbConstraints.fill = GridBagConstraints.HORIZONTAL;
    gbConstraints.gridy = row;
    gbLayout.setConstraints(label, gbConstraints);
    scores.add(label);
    tfs[row] = make_five_fields(row);
  }

  private void make_dice_and_checkboxes(final DiceBoard db, boolean cheat
    diceSet.setBackground(Color.white);
    diceSet.setLayout(new GridLayout(2, DiceBoard.NUMDIE));
```



```
  protected void clickScoreCell(int code, Match m) {
        Player p = m.getCurrentPlayer();
        int  gameNum = m.getGameNum();
        Evaluator eval = new Evaluator ();
        eval.evaluate(db, code, p, gameNum);
        show_scoreSheet(p, gameNum);
        disable_AllCells();
        rollButton.setEnabled(false);
        nextPlayerButton.setEnabled(true);
      }

  private void attach_listener (final Button b, final int code, final Mat
        b.addActionListener(new ActionListener () {
         public void actionPerformed(ActionEvent e) {
            clickScoreCell(code, m);
         }
        });
      }

  private TextField[] make_five_fields (int row) {
        gbConstraints = new GridBagConstraints();
        gbConstraints.gridy = row;
        gbConstraints.fill = GridBagConstraints.NONE;
        final TextField tfsrow[] = new TextField[Match.MAXGAMES];
        for (int field = 0; field < Match.MAXGAMES; field++) {
          tfsrow[field] = new TextField(3);
          tfsrow[field].setEditable(false);
          gbConstraints.gridx = field+1;
          gbLayout.setConstraints(tfsrow[field], gbConstraints);
          scores.add(tfsrow[field]);
        }
        return tfsrow;
      }

  private void oneRow (final String label, final int butCode, final Match
        gbConstraints = new GridBagConstraints();
        cats[butCode] = new Button(label);
        cats[butCode].setEnabled(false);
        cats[butCode].setForeground(Color.black);
        cats[butCode].setBackground(Color.lightGray);
        gbConstraints.gridx = 0;
        gbConstraints.fill = GridBagConstraints.HORIZONTAL;
        gbConstraints.gridy = butCode;
        gbLayout.setConstraints(cats[butCode], gbConstraints);
        scores.add(cats[butCode]);

        tfs[butCode] = make_five_fields (butCode);
        attach_listener(cats[butCode], butCode, m);
      }

  private void make_totals_rows(String s, int row) {
        gbConstraints = new GridBagConstraints();
        Label label = new Label(s);
        gbConstraints.gridx = 0;
        gbConstraints.fill = GridBagConstraints.HORIZONTAL;
        gbConstraints.gridy = row;
        gbLayout.setConstraints(label, gbConstraints);
        scores.add(label);
        tfs[row] = make_five_fields(row);
      }

  private void make_dice_and_checkboxes(final DiceBoard db, boolean cheat
        diceSet.setBackground(Color.white);
        diceSet.setLayout(new GridLayout(2, DiceBoard.NUMDIE));
        for (int die = 0; die < DiceBoard.NUMDIE; die++) {
           dice[die] = new GuiDice(db.getDie(die), cheat);
           diceSet.add(dice[die]);
        }

        for (int die = 0; die < DiceBoard.NUMDIE; die++) {
           final int which = die;
           chbs[which] = new Checkbox();
           chbs[which].setEnabled(false);
           chbs[which].addItemListener(new ItemListener () {
             // This method is called when the user clicks the button
             public void itemStateChanged(ItemEvent e) {
               db.getDie(which).toggleLock();
             }
          });
          diceSet.add(chbs[which]);
        }
      }

  public void configure_die (DiceBoard db) {
        for (int die = 0; die < DiceBoard.NUMDIE; die++) {
          dice[die].showval(db.getDie(die).getValue());
        }
      }

  protected void doARoll(DiceBoard db, Match m, final Button rollBut) {
        db.rollBoard();

        configure_die(db);
        for (int die = 0; die < DiceBoard.NUMDIE; die++) {
```



```
  protected void clickScoreCell(int code, Match m) {

  private void attach_listener (final Button b, final int code, final Mat

  private TextField[] make_five_fields (int row) {

  private void oneRow (final String label, final int butCode, final Match

  private void make_totals_rows(String s, int row) {

  private void make_dice_and_checkboxes(final DiceBoard db, boolean cheat

  public void configure_die (DiceBoard db) {

  protected void doARoll(DiceBoard db, Match m, final Button rollBut) {

  private void final_scores (Match m) {

public GuiYahtzee (final Match m, boolean cheat) {

public static void main (String[] args) {

}
```

(a) Flat text.

(b) Legible elision.

(c) Illegible elision.

Fig. 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.

and detailed (expanded) views within the same display. This is one of the objectives of fisheye visualisation systems (Section 2.2). Conversely, it is feasible that the integration of abstract and detailed views in elision systems may increase the cognitive and motor costs of browsing programs. If the cognitive and motor costs of continually configuring the elision display outweigh the benefits of integrating the views, then elision systems will fail.

In terms of motor control of the interface, another theoretical advantage of elision is that it should reduce the amount of scrolling required to navigate between regions of text. As the scrollbars in Figure 3 show, elision allows a greater proportion of the text to be displayed within a single window, meaning that less scrolling is required to reach targets. Once the correct region is located, elision users must expand the code, requiring a further cursor pointing task. There is, however, research evidence suggesting that the benefits of rapid scrolling will outweigh the costs of the additional pointing task. This evidence stems from prior investigations into the Fitts' Law (1954) efficiency of pointing and scrolling tasks. MacKenzie (1991) showed that the throughput, or 'bandwidth', of human mouse control is approximately 5 'bits/second' for pointing tasks. Hinckley, Cutrell, Bathiche & Muss (2002) showed that for scrolling tasks, which are also accurately modelled by Fitts' Law, have a dramatically lower throughput of approximately 1.5 bits/second. Essentially, these results show that users are less efficient at reaching targets when scrolling to them than when moving the cursor directly.

4 Empirical Studies of Code Elision

The aim of the experiment was to determine whether text elision improves programmer efficiency in typical source code editing and browsing tasks. We also wished to compare the performance of different levels of elision as the size of the source code files increased. We did not scrutinise the independent theoretical costs and benefits described in the previous section. Instead, we chose to first determine whether elision could provide statistically reliable performance improvements.

Four different types of navigation tasks were included in the evaluation. Each task involved navigation in a single Java class. Data from each task was analysed separately using a 2×3 factorial experimental design 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 in Section 4.2.

4.1 Participants

The twelve participants were volunteer postgraduate Computer Science students. Although the number of participants is relatively low, the repeated measures experimental design gives a relatively high degree of statistical power. All were males, with eleven in their early twenties, and one 45 years old. All participants had at least three years experience with Java syntax as part of their studies. Three of the participants held part-time jobs as programmers with local software companies.

Each participant's involvement lasted approximately twenty-five minutes, including training. Training involved explaining and demonstrating each of the experimental interfaces, then allowing participants to familiarise themselves with each by browsing through a sample Java file.

4.2 Materials

In selecting Java classes for use in the navigation tasks, 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 in Graphical User Interface (GUI) code. However, such classes are often constructed with the aid of direct manipulation GUI builders, and are not always edited manually. We 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 (such as 'find the largest method'). Furthermore, to avoid confusion in tasks requiring a named method to be found, we removed or changed the name of overloaded methods in each class.

In order to focus 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 searching and the navigation tree that allows rapid shortcuts to the methods in the class (see the left-hand side of Figure 2). These modifications were made to focus the experiment on navigation within the text editor. In all tasks the source code editor window was fixed at the same absolute size of 80 characters wide by 40 lines long (measured when the text is displayed at full size).

The experimental tasks, described below, focus on how rapidly users can navigate through source code using differing levels of elision. Although we based these tasks on activities that we believed programmers carry out frequently, they are artificially constrained versions of programmers' real work. This issue of ecological validity is further discussed in Section 6.1

4.2.1 Task One: Signature Retrieval

All subtasks in this task were of the form: "Find the type of the $< x$th> argument to method <method name>." This required the participants to retrieve information from the signature of a method. This task is intended to resemble a common programming activity—for example, when writing code to invoke a method, programmers often want to check argument types, and they may refer to the method signature to do so. 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, and consequently there is less information that must be visually searched. Also, because the unneeded information is suppressed, it is more likely that the target code will be displayed within the window, and if not, the average scroll distance will be lower.

4.2.2 Task Two: Body Retrieval

In this case, subtasks were of the form: "In method <method name>, find the first call to method <called name>." This required the participants to find the method and inspect its contents. This is indicative of a 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 task 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 participants 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, participants needed to 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 the participants' behaviour. Although the elided text is just large enough to read, we were unsure whether the participants would expand the method to full size or solve the task by reading the small text.

4.2.3 Task Three: Combination of Body Search and Signature Retrieval

Subtasks of this task were all of the form: "In method <method name>, find the return type of the method that is called last". This required the participants to find a method signature, inspect its method details and retrieve another method signature within the class. It is equivalent to Task Two with an additional task from Task One. Participants were instructed not to infer the return type from the method call, forcing them to perform the second search.

The task is intended to be indicative of source code navigation, 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 Task Two, we were unsure whether users of the legible elision interface would choose to read the small text or fully expand the method details, and we were therefore not confident in predicting its efficiency.

4.2.4 Task Four: Program Browsing

The final task involved answering the question: "Determine the longest method in the class".

One of the theoretical advantages of elision systems (Section 3.1) is that they can integrate both focused information and contextual overviews within the same display. Although this experimental task is artificial, we included it in order to partially test whether the presence of elision allowed programmers to more rapidly assess contextual information about the source code.

To avoid the need to count the number of lines, classes were chosen such that the largest method was clear from a visual scan of the code. Participants 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.

4.3 Procedure

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

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

Each task was presented to the participant in a command-line control interface. Once they had read the task and confirmed that they understood it, they pressed a key to begin. The control interface then opened Jaba, displaying the class at the appropriate level of elision. The timing was performed by the control interface, and began once the file had been fully loaded in Jaba. On completing each task, the participant clicked a 'Done' button at the bottom of the control interface, which recorded the task time in a log file. The experimenter ensured that the subtask really was complete, and requested that the participant continue the task if their solution was incorrect. The clock ran a cumulative task time, so that subsequent clicks of the 'Done' button recorded the total task time.

After each task, participants were asked to to express any free-form comments on their experience with the interface or with the task.

5 Results

Overall, the participants 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 s.d.=5.5) across the 288 task pool (twelve participants, four tasks, three interfaces and two file types). Errors occurred very seldom, and were not analysed. Performance data for the four tasks are summarised in Table 1.

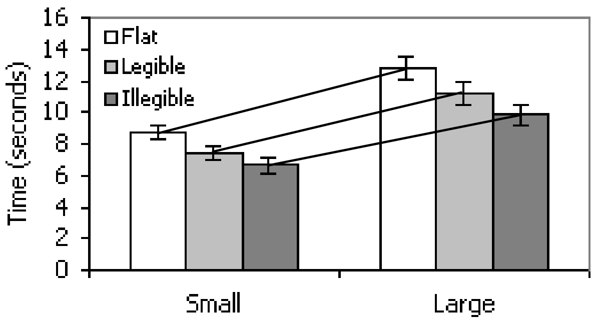


Fig. 4. Task One: Mean task completion times and standard errors (above and below the mean).

5.1 Task One: Signature Retrieval

Task 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 (Section 4.2.1).

The mean task times for small and large files were 7.58 (s.d. 1.7) and 11.28 (s.d. 2.58) seconds, providing a reliable difference (F(1,11) = 95.5, $p < .001$). This unsurprising result shows that the participants took longer to browse large files than short ones, probably due to the additional scrolling required.

The means for the flat, legible and illegible interfaces were significantly different at 10.74 (s.d. 2.8), 9.30 (s.d. 2.8) and 8.24 (s.d. 2.5) seconds respectively (F(2,22) = 11.6, $p < .001$). As shown in Figure 4, illegible elision performed best overall. In post-hoc comparison, a Tukey test (maintaining α at .05) yields an Honest Significant Difference (HSD) of 1.87, confirming a significant difference between performance with the flat and illegible interfaces. This confirms our prediction that for retrieving method signatures (non-elided elements), the suppression of irrelevant detail increases efficiency.

There was no significant interaction between interface type and file size (F(2,22)

	Task 1		**Task 2**		**Task 3**		**Task 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)

Table 1
Mean (standard deviation) times in seconds for each task, across each level of interface type and file size.

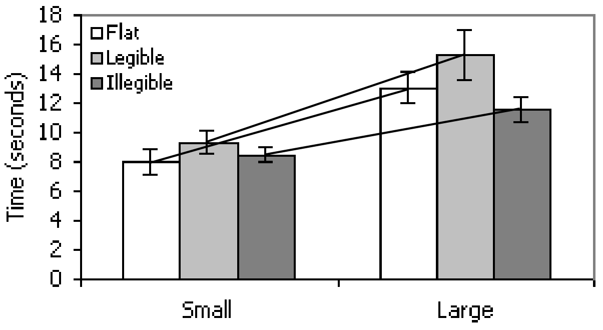


Fig. 5. Task Two: Mean task completion times and standard errors.

= 0.33, p = .72). The absence of an interaction is clear in Figure 4, which shows that the mean task completion times degraded between the small and large file sizes at a similar rate for the three interfaces.

5.2 Task Two: Body Retrieval

Task 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 (see Section 4.2.2).

There was a significant difference between the mean task times for small and large files of 8.59 (s.d. 2.5) and 13.3 (s.d. 4.5) seconds (F(1,11) = 56.6, p < .001). This is a natural 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 (s.d. 4.2), 12.32 (s.d. 5.4) and 10.02 (s.d. 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 5 shows that for large files, the benefits of illegible elision are larger than for small files.

5.3 Task Three: Combination

Task Three combined Tasks 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

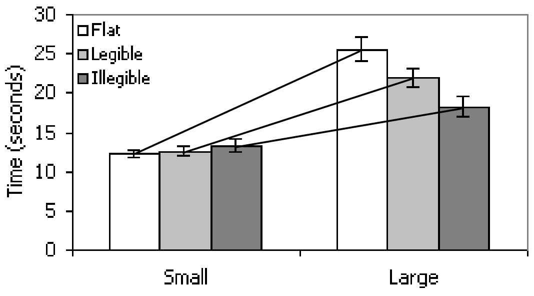


Fig. 6. Task Three: Mean task completion times and standard errors.

that the illegible elision interface would allow the most rapid task completion, and we were interested to see how the participants would use the legible elision interface (Section 4.2.3).

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 (s.d. 7.8), 17.3 (s.d. 5.8) and 15.7 (s.d. 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. Post-hoc comparison gives a Tukey HSD value of 2.89, showing a significant difference ($p < .05$) between performance with the flat and illegible interfaces.

When browsing small files, the mean task completion times across the three interfaces were similar. However, as shown in Figure 6, 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.

5.4 Task Four: Program Browsing

Task Four compared the participants' 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 (Section 4.2.4)..

The mean task times for small and large files of 7.6 (s.d. 2.5) and 12.6 (s.d. 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 (s.d. 4.8), 9.7 (s.d. 3.5), and 9.9 (s.d. 4.0) seconds for the flat, legible and illegible interfaces (F(2,22) = 1.1, $p = .36$).

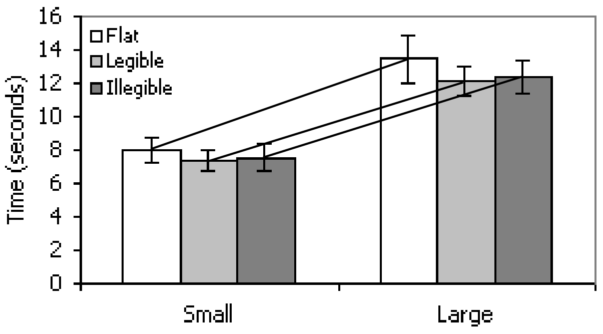


Fig. 7. Task Four: Mean task completion times and standard errors.

There was also no significant interaction between file size and interface type (F(2,22) = 0.38, p = .7).

5.5 Observations and Participants' Comments

When first shown the illegible elision interface during training, several of the participants mentioned that the interface was 'neat' or 'cool'. The performance results show that it is more than this—it yielded statistically significant performance improvements.

In Task One, we were surprised by the significant interaction between interface type and file size (Figure 4). 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 participants' comments, however, explained the effect. In Jaba, method signatures were the only program elements coloured red. When using the flat text interface, the participants 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 flat text when using larger files. This observation—that coloured 'signals' in the program code may have improved performance with both the flat and elision interfaces—is consistent with the study by Tapp & Kazman (1994) which found that coloured code segments aided certain programming tasks (see Section 2.1).

In Task Two, Figure 5 shows that the legible elision interface was on average slower than the flat text interface (although this is not statistically reliable). We observed that when using legible elision, ten out of twelve participants 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 participants reported the need to 'squint'. Only two of those ten participants changed their behaviour after experiencing this problem. Despite encouraging them to squint, several participants commented that they preferred the legible interface because it provided a good balance between flat text and illegible elision. Similar comments were made in Task Three.

In Task 4, we were surprised by the similarity of performance across the three interfaces (Figure 7). Even though the mean task completion times with the elision interfaces were lower, the differences were very small. Participants commented that with the flat text interface, they scrolled more rapidly, trusting their eyes to identify large blocks of text. In many cases, participants did not need to compare similar methods, and could determine the answer from only a single scan.

6 Discussion and further work

Although the participants' performance mostly agreed with our predictions, the discrepancies between prediction and performance provide insights into the accuracy and levels of the theoretical pros and cons of elision outlined in Section 3.1. The theoretical advantages of elision primarily stem from the reduced cognitive load of searching through 'flat' information and from the reduced motor task of scrolling shorter distances. The theoretical disadvantages primarily stem from the cognitive costs of deciding which details to expand and contract, and from the motor costs of doing so. The performance measures support the existence of these benefits and costs, but indicate that the costs of configuration may be higher than expected.

According to this theory, Task One (find a method signature) should provide the benefits with none of the costs because there is no need to configure the level of elision. The results agree, showing that greater elision results in better performance. Task Two (find a method invocation within a method), however, has the same theoretical advantage for elision, but introduces the cost of configuring the level of elision when reading the method contents. Although the results showed no significant difference between the interfaces, the legible elision interface performed particularly poorly (Figure 5). The most likely explanation is the 'squinting' effect, where the participants avoided the cost of configuring the level of elision, but incurred a greater one by trying to read the tiny text. With the illegible elision interface, the participants had no choice but to incur the cost of expanding the text, yet they solved the task faster (on average) than the flat and legible interfaces. This result is related to those of Tapp & Kazman (1994) and Gellenbeck & Cook (1991) who found that changing the size of text within legible levels is of dubious value in support of programming tasks (see Section 2.2). This interpretation suggests that legible elision should be avoided.

Task Three (find the signature of a method that is invoked on the last line of a named method) also agrees with the theory. In the task, the elision advantage of reduced scrolling and information search applies twice: first when finding the original method, and again when finding the signature of the method referred to on the method's last line. Like Task Two, the cost of configuring the level of elision only applies once, when expanding the method body. In this case, the double application of the search advantage appears to have outweighed the configuration cost, resulting in a significant efficiency benefit for elision.

The results of Task Four are harder to explain with respect to the theoretical costs and benefits, which predict a significant advantage for elision due to improved search with no cost in elision configuration. A possible explanation is that the participants' had not yet gained sufficient familiarity with diminished text to feel confident in comparing text lengths. A more likely explanation, however, is due to the 'jerky' scroll velocity that occurs when scrolling over diminished text. In our implementation (like most text editors), there is a constant mapping that determines the number of pixels the scrollbar must move to cause each line of text to disappear (scrolling down) or appear (scrolling up) at the top of the window. Consequently, if the user drags the scrollbar at a constant velocity, then the rate at which text scrolls through the window depends on whether the lines at the top of the window are elided or not. When the lines are elided the text appears to scroll slowly, and when they are expanded the text appears to scroll quickly. A constant scroll speed through a document that is part elided and part expanded, therefore causes jerky text motion, even though lines are leaving or entering the top of the window at a constant rate. Several participants commented on the 'jerky' scrolling, and it is reasonable to suspect that this affected their performance in Task Four. Recent research has investigated ways of overcoming this effect, as discussed in Section 6.2.

Finally, it is likely that the theoretical costs and benefits of elision interfaces outlined in Section 3.1 are applicable in a wide range of application areas beyond program navigation. The successive display of increasingly detailed information within structured documents seems to be particularly attractive for browsing large documents on small displays, such as personal digital assistants (Buyukkokten et al. 2000).

6.1 Experimental Further Work

Although the results are promising, there are many limitations in the study. We plan to address some of these in our further work.

The experiment investigated the efficiency of simple code navigation tasks within single files. These tasks are useful for initial investigation into the efficiency of elision, but they are caricatures of real programming activities that raise concerns about ecological validity. These concerns can only be removed through field trials of elision mechanisms in commercial systems such as Together^{TM}, shown in Figure 1. There are many other concerns about the experimental tasks. For example, all tasks started from fully contracted states and finished on finding information. They therefore ignored the costs of managing the display so that information is elided when no longer necessary. This burden of 'cleaning up' the display may be unacceptable for many users, for whom the code view will degenerate to a fully expanded state, removing the advantages of elision. Further evaluation is necessary to determine how users adapt to the presence of elision facilities.

Also, in focusing on task efficiency, we have not investigated the impact that elision has on program comprehension. We wish to conduct studies, similar to those reported in Section 2, to determine the impact (if any) of elision on program comprehension. This issue is particularly important given the indication from Hornbaek & Frokjaer (2001) that fisheye text editing resulted in inferior comprehension than their overview+detail editor.

Like all behavioural experiments, there are concerns of generality and validity in our experiment. These include the generality concern of using a participant pool of postgraduate students rather than professional programmers, and the validity concern of constraining the experiment to only the tools available within the text-editor. In particular, most current commercial programming environments allow navigation shortcuts through a graphical depiction of program contents (similar to the element on the left of Figure 2), yet this feature was disabled in our evaluation to focus the tasks on navigation within the editor window. Further experiments are necessary to determine whether the theoretical benefits of total elision in the graphical overview exceed the costs of changing focus and moving the pointer to the side window and back. A related issue awaiting evaluation is a comparison between total elision (where text is removed completely from the display) and illegible elision. Finally, having determined that illegible elision can improve navigation efficiency, we wish to more closely analyse the levels of the theoretical costs and benefits. By doing so, we hope to better understand the appropriate granularity of elision: whether classes, methods, or smaller components such as block statements are suitable for elision.

Despite these concerns and limitation, we believe the experiment has been successful in establishing a starting point for empirical evaluation of elision in program navigation, and we believe the study is relevant because these features are appearing in commercial software without a firm understanding of its use.

The problem of 'jerky' scrolling velocity, caused by the presence of text of various sizes, was described earlier in this section. Several researchers are examining mechanisms to increase the efficiency and smoothness of moving between abstract and detailed views of information, including text. These motion smoothing techniques may provide suitable alternatives to the explicit and discrete elision method used in Jaba.

The problems of undesired motion in distortion-oriented systems are well known. Bederson (2000) described the targeting problem with his fisheye menu system, in which menu items are increasingly elided with distance away from the cursor location in the menu. To ease the problem, fisheye menus allowed the user to approximately locate the target item before 'locking' the fisheye by moving the cursor rightwards to disable distortion (and side-effect movement) while precise target acquisition was completed. Although effective, this technique introduces a second mode of menu use that is comparatively cumbersome for an otherwise simple menu selection, and similar techniques are unlikely to be applicable in text editors.

Research on the problems of selecting expanding targets has led to proposals for improved 'speed-coupled' scrolling mechanisms that could be applied in elision-like interfaces. McGuffin & Balakrishnan (2002) examined the time to select targets in a one-dimensional movement tasks where target width increased as the cursor approached. They found that selection time for expanding targets is accurately modelled by Fitts' Law, and that selection time depends on the final size of the target rather than the initial size. In apparent contradiction, an evaluation of two dimensional target acquisition by Gutwin (2002) disagreed with the findings of McGuffin & Balakrishnan. Gutwin's results show that performance in selecting targets is detrimentally affected by expansion, with performance becoming worse as the level of expansion increases. The disagreement is explained by the different expansion implementations. McGuffin & Balakrishnan used a simple in-place expansion, where the target centre did not move, whereas Gutwin used the well known Sarkar & Brown (1992) graphical fisheye layout algorithm that causes target motion. With the Sarkar & Brown fisheye, targets move towards the cursor as they expand, with the rate of target displacement being greatest at the point the cursor enters the target. This movement causes 'hunting effects' where the user's cursor overshoots the target, requiring a change in cursor direction, and the risk of repeated overshooting. To overcome this problem, Gutwin described 'speed-coupled flattening' in which the level of distortion is inversely proportional to the cursor velocity. Initially the cursor is stationary, and the display is maximally distorted around the cursor location. When the user accelerates to the ballistic phase of cursor movement the distortion is modified to zero, flattening the display. Finally, as the user decelerates the cursor in final target acquisition, the distortion begins to apply, and as the cursor stops over the target, the maximal level of distortion is re-applied around the new cursor location. Gutwin's experiments showed that speed-coupled flattening improves the speed and accuracy of target acquisition over the traditional Sarkar & Brown fisheye.

These results are relevant to our evaluation of elision interfaces because they are applicable to large text documents in improving the efficiency of scrolling, and promise to overcome the problems of 'jerky' scrolling. Igarashi & Hinckley (2000) describe the concept of 'speed-dependent automatic zooming' for browsing large documents. With this technique visual flow of text up/down the screen during scrolling is maintained at a constant rate. When scrolling rapidly the document is zoomed out, so that more information is scrolled per unit time[2]. A preliminary evaluation of the technique indicated that it enhances the efficiency of browsing large documents. We are interested to explore how these techniques can be usefully deployed in support of program navigation.

7 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.

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

The evaluation reported in this paper compared the efficiency 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 showed that users were able to complete navigation tasks more quickly with the eliding interfaces, particularly when working with larger source code

[2] Movies and applets demonstrating the technique are available at Takeo Igarashi's website `http://www.mtl.t.u-tokyo.ac.jp/~takeo/research/autozoom/autozoom.htm`

files. Although several participants' commented that they preferred 'legible' elision, performance was better when using illegible elision. The primary cause of inefficiency with the legible elision interface was that it encouraged users to read text that was much smaller than normal. This lead to slower reading speeds and comments of 'squinting at the text'. For these reasons, legible elision seems inadvisable, while illegible elision appears promising.

The results and observations support the theoretical costs and benefits of eliding interfaces described in the paper: the costs being the cognitive and motor tasks of configuring the level of elision, and the benefits being the reduced search and scrolling demands. Further work will attempt to refine our understanding of the levels of these costs and benefits.

Acknowledgements

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

References

Baecker, R. (1988), Enhancing Program Readability and Comprehensibility with Tools for Program Visualization, *in* 'Proceedings of the 10th International Conference on Software Engineering, Singapore', pp. 356–366.

Baecker, R. & Marcus, A. (1990), *Human Factors and Typography for More Readable Programs*, Addison-Wesley.

Bederson, B. (2000), Fisheye Menus, *in* 'Proceedings of the 2000 ACM Conference on User Interface Software and Technology, San Diego, California.', pp. 217–225. 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.'. *citeseer.nj.nec.com/buyukkokten00seeing.html.

Card, S., Moran, T. & Newell, A. (1983), *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates.

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

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

Fitts, P. (1954), 'The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement.', **47**, 381–391.

Friendly, L. (1995), The Design of Distributed Hyperlinked Programming Documentation, *in* 'Proceedings of the International Workshop on Hypermedia Design, Montpellier, France, 1-2 June', Springer, pp. 151–173.

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

Gellenbeck, E. & Cook, C. (1991), Does Signaling Help Professional Programmers Read and Understand Computer Programs?, *in* 'Empirical Studies of Programmers: Fourth Workshop', Ablex Publishing.

Griswold, W., Chen, M., Bowdidge, R., Cabaniss, J., Nguyen, V. & Morgenthaler, J. (1997), 'Tool Support for Planning the Restructuring of Data Abstractions in Large Systems', *IEEE Transactions on Software Engineering* **24**(7), 534–558.

Gutwin, C. (2002), Improving Focus Targeting in Interactive Fisheye Views, *in* 'Proceedings of CHI'2002 Conference on Human Factors in Computing Systems Minneapolis, Minnesota, 20–25 April', pp. 267–274.

Hansen, W. (1984), User Engineering Principles for Interactive Systems, *in* D. Barstow, H. Shrobe & E. Sandewall, eds, 'Interactive Programming Environments', McGraw-Hill, pp. 288–299.

Hinckley, K., Cutrell, E., Bathiche, S. & Muss, T. (2002), Quantitative Analysis of Scrolling Techniques, *in* 'Proceedings of CHI'2002 Conference on Human Factors in Computing Systems Minneapolis, Minnesota, 20–25 April', pp. 65–72.

Hornbaek, K. & Frokjaer, E. (2001), Reading of Electronic Documents: The Usability of Linear, Fisheye, and Overview+Detail Interfaces, *in* 'Proceedings of CHI'2001 Conference on Human Factors in Computing Systems Seattle, Washington, March 31–April 6', pp. 293–300.

Igarashi, T. & Hinckley, K. (2000), Speed-dependent Automatic Zooming for Browsing Large Documents, *in* 'Proceedings of the 2000 ACM Conference on User Interface Software and Technology, San Diego, California.', ACM Press, pp. 139–148.

Knuth, D. (1984), 'Literate programming', *The Computer Journal* **27**(2), 97–111.

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

Lamping, J., Rao, R. & Pirolli, P. (1995), A Focus+Context Technique Based on Hyperbolic Geometry for Visualising Large Hierarchies, *in* 'Proceedings of CHI'95 Conference on Human Factors in Computing Systems Denver, May 7–11', pp. 401–408.

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

MacKenzie, I. (1991), Fitts' Law as a Performance Model in Human-Computer Interaction, PhD thesis, University of Toronto: Toronto, Ontario, Canada.

McGuffin, M. & Balakrishnan, R. (2002), Acquisition of Expanding Targets, *in* 'Proceedings of CHI'2002 Conference on Human Factors in Computing Systems Minneapolis, Minnesota, 20–25 April', pp. 57–64.

Miara, R., Musselman, J., Navarro, J. & Shneiderman, B. (1983), 'Program Indentation and Comprehensibility', *Communications of the ACM* **26**(11), 861–867.

Oman, P. & Cook, C. (1990), 'Typographic Style is More Than Cosmetic', *Communications of the ACM* **33**(5), 506–520.

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.

Shum, S. & Cook, C. (1993), 'AOPS: An Abstraction-Oriented Programming System for Literate Programming', *Software Engineering Journal* **8**(May), 113–120.

Shum, S. & Cook, C. (1994), Using Literate Programming to Teach Good Programming Practices, *in* 'Proceedings of SIGCSE'94: Twenty-fifth Technical Symposium on Computer Science Education. Phoenix, Arizona. 10–11 March.', pp. 66–70.

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

Soloway, E. (1986), 'Learning to Program = Learning to Construct Mechanisms and Explanations', *Communications of the ACM* **29**(9), 850–858.

Spence, R. (1999), 'A Framework for Navigation', *International Journal of Human-Computer Studies* **51**, 919–945.

Tapp, R. & Kazman, R. (1994), Determining the Usefulness of Colour and Fonts in a Programming Task, *in* 'Proceedings of the 3rd Workshop on Program Comprehension Washington, D.C.', pp. 154–161.
*citeseer.nj.nec.com/tapp94determining.html

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.

Thimbleby, H. (1986), ‘Experiences of ‘Literate Programming’ using cweb (a variant of Knuth’s WEB)’, *The Computer Journal* **29**(3), 200–211.