INFSOF4201

ELSEVIER

Information and Software Technology 00 (2001) 000–000

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Supporting tailorable program visualisation through literate programming and fisheye views

Andy Cockburn*

Department of Computer Science, University of Canterbury, Christchurch, New Zealand

Received 11 April 2000; revised 23 July 2001; accepted 14 August 2001

Abstract

This paper describes the 'Jaba' program editor and browser that allows users to tailor the level of abstraction at which they visualise, browse, edit and document object-oriented programs. Its design draws on concepts from literate programming, holophrasting displays, fisheye visualisation and hypertext to allow programmers to rapidly move between abstract and detailed views of Java classes. The paper focuses on the motivation for, and user interface issues surrounding, the integration of these facilities in Jaba. Limitations in the current tools and theories for programming support are identified, and modifications are proposed and demonstrated. Examples include overcoming the static post-hoc documentation support provided by Javadoc, and normalising Furnas's 'degree of interest' fisheye visualisation formula to avoid excessive suppression of program segments. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Literate programming; Documentation; Fisheye visualisation; Hypertext; Programming environments; Java

1. Introduction

Computer programming is a demanding activity. Programmers work within complex information spaces at many different levels of abstraction. For example, modifying the internal structure of a method requires a detailed view of its contents, but invoking a method needs only an abstract view of its method signature to determine the number, type and order of parameters. Fig. 1 shows the problem in an object-oriented program. It shows a program line inside class *X*, and the possible points of reference that the programmer may wish to view in association with the line. The figure also shows the limited display extent of a 'typical' editor window into class *X*. To ease these problems modern programming environments include powerful searching and marking capabilities, and many support context-sensitive editing features such as pop-up menus that let the programmer select available methods from object reference variables. Despite these enhancements, each editor window is essentially a 'flat' representation of program text that displays the programmer's focal point of interest and whatever neighbouring text fits into the window extent; scrolling, searching and marking must be used to move between related program segments that lie outside the display extent of the window.

This paper describes 'Jaba', a hypertext system that supports programmers in visualising, browsing, editing and documenting object-oriented programs. By integrating concepts from 'literate programming' [15,14], 'holophrasting displays' [25], 'fisheye views' [8], and hypertext [6], Jaba allows programmers to tailor the level of program detail displayed across an arbitrary number of program regions. It automatically divides the program into 'chunks' that encapsulate syntactic program units, and users can add further chunks to capture the cognitive units that they perceive in their programs. Literate programming techniques support a strong connection between program code and its associated documentation. Holophrasting schemes allow the user to show or hide program regions, and fisheye views are used to tailor the level of detail shown at, and around, the user's focal point in the program. The aim is to enhance the user's ability to focus on, and navigate through, the salient program details without the distraction of display-space clutter from superfluous information.

The structure of the paper is as follows. Section 2 provides background reviews of literate programming, holophrasting interfaces and fisheye views. The javadoc system, which produces HTML documentation from Java classes is included in the review to motivate enhancements in systems such as Jaba. Readers who are familiar with these techniques may wish to move directly to Section 3 which describes the Jaba system. Section 4 provides the rationale behind the major design decisions, and Section 5 critically

* Tel.: +64-3-364-2987/7768; fax: +64-3-364-2569.
E-mail address: andy@cosc.canterbury.ac.nz (A. Cockburn).

PII: S0950-5849(01)00182-3

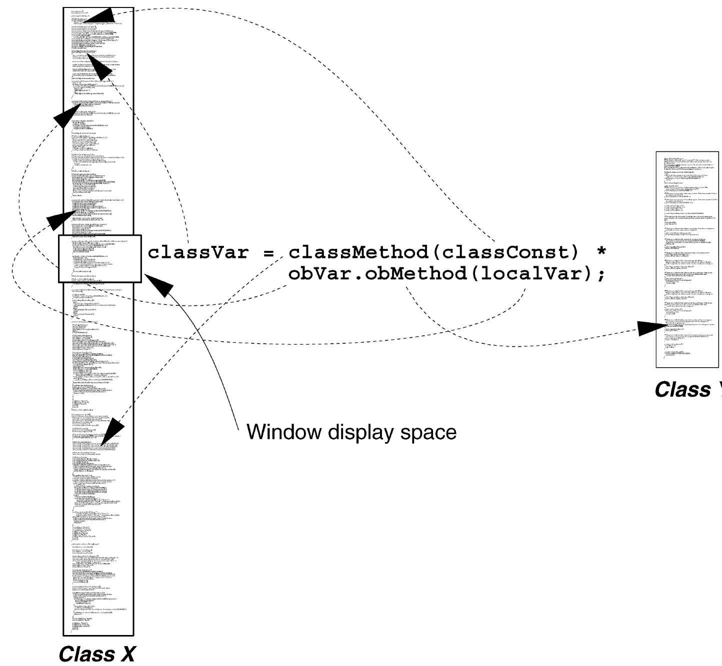


Fig. 1. Interconnections arising from one program line. Window display space over laid in class *X*.

assesses Jaba's capabilities and discusses further work. Interactive programming environments that demonstrate related capabilities to those of Jaba are presented in Section 6. Section 7 summarises and concludes the paper.

2. Background: programming and levels of detail

2.1. Literate programming

Literate programming [14,15] is an elegant technique that allows programmers to design, document, and construct their programs in whatever order best aids human understanding. Using a literate programming tool, users can arrange programming elements and their accompanying documentation in whatever order they choose, rather than having the order of exposition dictated by the requirements of the language's compiler or interpreter. The resultant literate program consists of 'chunks' of code and documentation in which the chunks represent cognitive units in the program. These cognitive chunks need not correspond to the programming language's syntactic constructs. For example, a cognitive chunk for a looping construct may contain a set of variable assignments that establish pre- and post-conditions in addition to the syntactic elements of the loop. Defined chunks can be used by zero or more other chunks.

Literate programs can be 'tangled' to produce code that is ready for processing by a compiler or interpreter, or they can be 'woven' to produce documentation that includes extensive cross-referencing and indexing of program elements. Literate techniques allow programmers to describe their programs clearly and precisely, with their documentation integrated into the program, in a manner that is impossible with standard CASE tools. Fig. 2 shows the mark-up of a java class 'QuickDemo' that implements the quick sort algorithm using the literate programming tool noweb [21]. Chunk definitions are denoted by the construct $\ll$Chunk Name$\gg =$, and chunk uses by $\ll$Chunk Name$\gg$. The 'root' chunk is identified by the chunk-name $*$. The root chunk in Fig. 2 'uses' four chunks ('Import Packages', 'Static variable declarations', 'The QuickSort method' and 'The main program'), which are each defined later in the literate program. Chunks may be defined in any order. Documentation chunks begin with the @ symbol.

The text-based mark-up of literate programs adds a layer of syntax on-top of the programming language syntax. Mistakes in the specification of the chunking structure cause syntax errors when the literate program is 'tangled' or 'woven'. For this reason Knuth did not advocate the use of literate programming for students or hobbyists. Graphical user interfaces, however, can overcome these problems by providing 'syntactic correctness' [24] — when the user requests modification to the chunking structure, the program can assure that the correct syntactic modifications are made to the underlying program. Such a graphical user interface to literate programming for novice programmers is described in [5].

2.2. Holophrasting program displays

Holophrasting interfaces [3,25] aim to improve visualisation of textual information spaces by providing contextual overviews that allow users to suppress or 'elide' the display of regions of text.

Holophrasting systems extract structural information

```
<<*>>=
  <<Import packages>>
  public class QuickDemo {
    <<Static variable declarations>>
    <<The QuickSort method>>
    <<The main program>>
  }
@ This program demonstrates the {\tt QuickSort} algorithm.  It reads
a list of numbers from the standard input, sorts them, and writes
the sorted results to standard output.

<<The QuickSort method>>=
  public void qsort (int[] data, int left, int right) {
    int cutval, temp, lo, hi;
    <<Sort and divide until divided down to nothing!>>
  }
@ Recursively sort an array {\tt data} of integers. {\tt left} and {\tt
right} denote the leftmost and rightmost elements of the array.
cutval is the value around which the array is sorted in each pass
through the array

<<Sort and divide until divided down to nothing!>>=
  if (right > left) {
    <<Get set by guessing a cut value and initialising indexes>>
    <<Sort array with respect to cut value>>
    <<Recursively sort left and right sub-arrays>>
  }
@ When we make a recursive call where the right and left indexes are
the same, then we've divided down to nothing and we're done with this
recursive call.

<<Get set by guessing a cut value and initialising indexes>>=
  cutval = data[right];
  lo = left -1;
  hi = right;
@ Arbitrarily pick the rightmost element of the array as the cut value
for this pass.
```

Fig. 2. Literate mark-up of a segment of the QuickDemo class using noweb [21].

from the document source. Document markup tags such as section and subsection headings can be used to determine structure. A variety of schemes has been proposed for extracting structure from computer programs. These include using the grammatical rules of derivation for the language, and the use of program blocks such as the sequence of statements between opening and closing braces in C. Holophrasting systems are reviewed in Section 6.

The document regions to be suppressed may be under direct user control, or may be automatically configured as the user moves their cursor through the document. A variety of interface mechanisms can be used to reveal that text has been suppressed-the most common is to display an ellipsis ('…'). Fig. 3 shows 20 lines of the QuickDemo class in a normal view (unholophrasted) and in a holophrasted view which uses ellipsis to represent suppressed text: line numbers are shown on the left of the program text. Note that the holophrasted display reveals the entire extent of the class (first line to the last line).

2.3. Fisheye visualisations

Furnas [8] introduced fisheye views as a way of allowing users to simultaneously view the details of their focal point of interest in an information space while also displaying the surrounding contextual information. Fisheye views have become a popular research topic and many systems have extended the research, particularly in graphical information spaces [17,18]. When applied to text, fisheye views are a powerful holophrasting technique in which the display

```
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       }
```

Unholophrasted

```
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 }
```

Holophrasted

Fig. 3. Twenty lines of the QuickDemo class in normal and holophrasted views.

contents are automatically adapted in an attempt to match the user's interest in regions in the document.

A simple 'degree of interest' (DOI) formula is used to calculate the user's 'interest' in all of the data-points in the information space (Eq. (1)). The two factors used in this calculation are the user's a priori interest in the data, and the distance that the data lies from the user's current focal point.

$$\mathrm{DOI}_{\mathrm{fisheye}}(x|. = y) = \mathrm{API}(x) - \mathrm{Distance}(x, y) \quad (1)$$

$\mathrm{DOI}_{\mathrm{fisheye}}(x|. = y)$ returns the user's interest in the information at point x, given that their current focus of attention is directed at point y. API(x) returns the user's a priori interest in data point x — it is a measure of the semantic importance of the information. In a map, for instance, it is reasonable to expect that cities would have a higher a priori interest than towns. In computer programs, API values decrease with the nesting depth of program elements. *Distance*(x, y) is a measure of the distance between points x and y — in hierarchical data structures such as computer programs, distance may be measured in terms of path distance between nodes, rather than as an absolute measure.

If the calculated DOI measure for data-point x falls below a threshold k, then the information at that point is suppressed or 'elided' (not displayed). In our experience, it is necessary to normalise the values returned by the DOI formula: this issue is further discussed in Section 3.5.

Furnas describes several example systems, including a visualisation mechanism for C programs. In this system, program details around the user's focus of interest are displayed in full, while only the 'landmark' program segments are displayed further from the user's location, producing program views similar to the holophrasted view shown in Fig. 3. Ellipses and non-contiguous line-numbers are used to indicate that lines in the text have been suppressed. Furnas provides preliminary empirical evidence that fisheye techniques can assist in searching hierarchical information.

Recent work on fisheye visualisations has greatly extended the original work, particularly in graphical displays of networks (for example, Refs. [23,16]). Fisheye visualisation techniques now offer many capabilities that

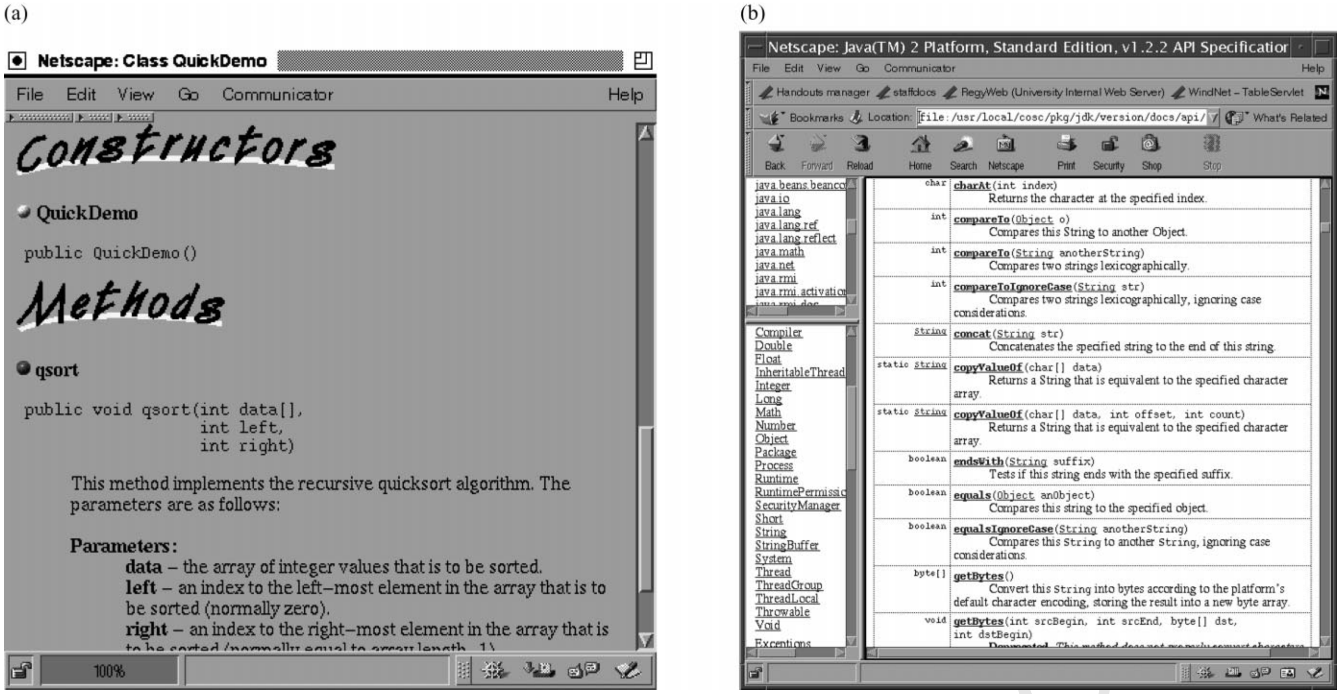


Fig. 4. Javadoc documentation. (a) Java 1 javadoc of the QuickDemo class; (b) Java 2 framed javadoc documentation of the class java.lang.String.

could be used to enhance the C program visualisation system originally proposed by Furnas. Multiple focal points [22] would allow programmers to selectively reveal the details of several points within program files, such as an editing point and a secondary reference point. Another possibility is to enrich the display mechanisms used to denote suppressed lines of text. Techniques such as scalable fonts would reveal much more information about the suppressed information while consuming minimal amounts of screen real-estate. Systems demonstrating text-based fish-eyes are reviewed in Section 6.

One potential problem with fisheye view techniques arises from the DOI formula's calculation of the user's degree of interest. The formula implements a heuristic assessment of the user's *likely* degree of interest, and it will sometimes incorrectly suppress desired information or display information that is unnecessary for the user's task. Thus, the formula will make it difficult for programmers to explicitly select portions of the text that should be displayed regardless of the user's movement within the program. The equivalent of 'manual overrides', or holophrasting, in the interface could be used to ensure that regions in the program stay displayed regardless of their calculated degree of interest.

2.4. Javadoc documentation

One of the major claims of the object-oriented programming paradigm is that it encourages and supports code reuse. In Java, code comprehension and reuse is greatly enhanced by the availability of javadoc[1] [7] documentation. The javadoc tool generates HTML documentation by parsing the contents of class files, and extracting information about methods, data-fields and any specially formatted comments. All of the Java API (application programmer's interface) can be reviewed with a web-browser through javadoc's consistent and easily comprehensible format. Fig. 4(a) shows a the javadoc generated for the QuickDemo class constructor and its qsort method.

Sun's Java2 javadoc produces framed HTML (Fig. 4(b)). The frames ease navigating between high-level packages, but the code-level documentation remains similar to version 1.1. The Continuous Zoom interface [11] used a graphical fisheye technique to ease navigation between package level views of the Java API and the javadoc documentation pages.

There are several opportunities for enhancing the support javadoc offers. First, javadoc produces static documentation that is separate from the actual code. Code modifications can therefore render the documentation redundant or incorrect. A dynamic version of javadoc could automatically ensure consistency between the documentation and the program code. Second, javadoc is a post-hoc documentation strategy that requires that the class have been developed into a syntactically correct (and presumably complete) class specification. An extension to javadoc could offer dynamically generated documentation even for partially complete classes. Third, javadoc offers only a single level of abstraction for investigating the class: it reveals method signatures, the names and types of class data-fields, and any specially formatted comments that the programmer has written at the

[1] http://java.sun.com/products/jdk/javadoc/.

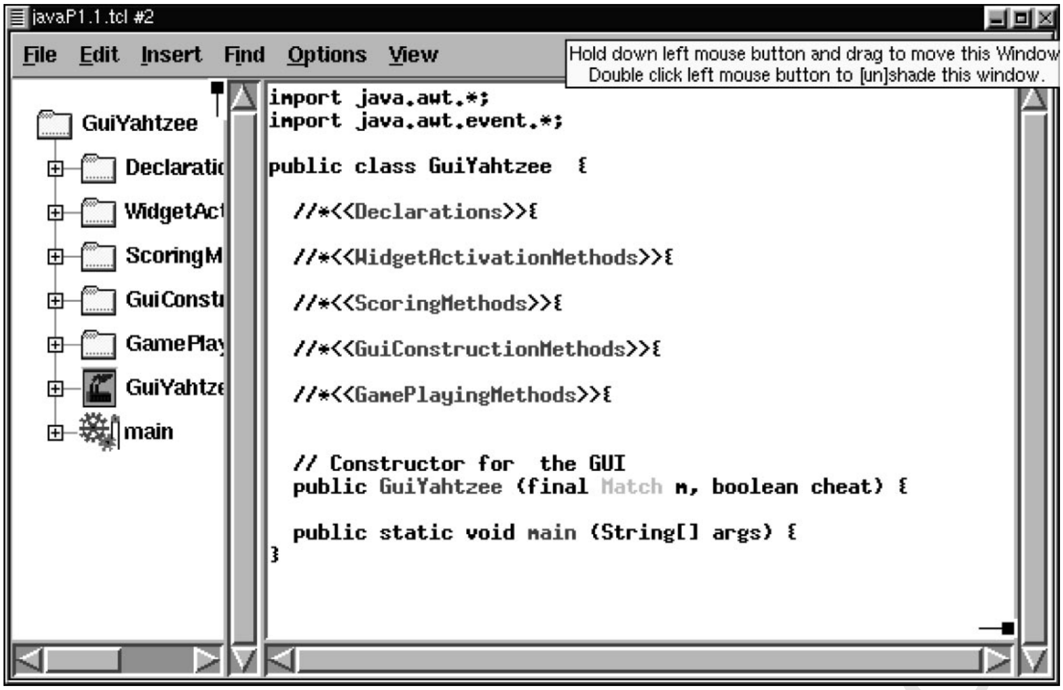


Fig. 5. Jaba's main window, showing the 'main' class of the Yahtzee program.

top-level in the class (formatted comments inside methods are ignored). An extension to javadoc could allow programmers to investigate the internal details of classes, for instance checking the details of the algorithm contained within a method. The Jaba system, described in Section 3, attempts to exploit each of these opportunities.

3. Jaba: system description

Jaba is an experimental tool for editing and browsing Java programs. Although its implementation is tightly bound to the Java programming language, most of the interface properties demonstrated by the tool could be adapted to a wide set of object-oriented programming languages.

3.1. Jaba overview

Fig. 5 shows a typical Jaba window, which contains two sub-windows: a graphical tree representation of the class structure (left), and a hypertextual text editor/viewer (right). An HTML text-viewer for displaying javadoc documentation is also available (bottom of Fig. 6). The graphical tree and javadoc windows can be hidden through checkboxes under the 'View' menu.

When a class is loaded into a Jaba window it is displayed at the most abstract level (as in Fig. 5). Only top-level chunks are shown, and none of the inner-details of those chunks is revealed. The GuiYahtzee.java class displayed in Fig. 5 contains over 400 program lines, but the entire extent of the class (first line to last line) is shown in the text-editor window. Semantic information about chunk-types is displayed in the graphical tree. Jaba supports five different types of chunks (Table 1), each of which has its own iconic representation in the graphical tree.

Users reveal successive levels of inner detail within chunks by clicking on the plus icons in the tree representation or by clicking the hypertext links in the text viewer/editor. When a chunk is expanded, the text it contains is shaded grey for two seconds to help the user perceive the extent of the newly displayed information contained in the chunk. The hypertext links associated with contracted chunks are coloured red and expanded links are coloured blue (all colours are configurable). Chunks are contracted by clicking on the link or by clicking the corresponding minus icon in the tree viewer. Several interface features are intended to assist programmers in navigating through the program. For instance, clicking on the name of a chunk in the graphical tree causes the text display to immediately scroll to the associated chunk. Other interface mechanisms that assist navigation are described in Section 3.4.

The top-half of Fig. 6 shows the system state after expanding two levels of inner detail. First, the user clicked the ScoringMethods link, which encapsulated two Java methods clickScoreCell and attach-listener). This caused abstracted representations of these methods to be displayed, showing only their signatures. The user subsequently clicked on the clickScoreCell hypertext link, revealing the inner-details of that method's code.

Jaba parses classes prior to displaying them. All five types of abstractions are detected, the types of all object variables are stored, and method invocations are detected, as are connections with super-classes such as overriding methods

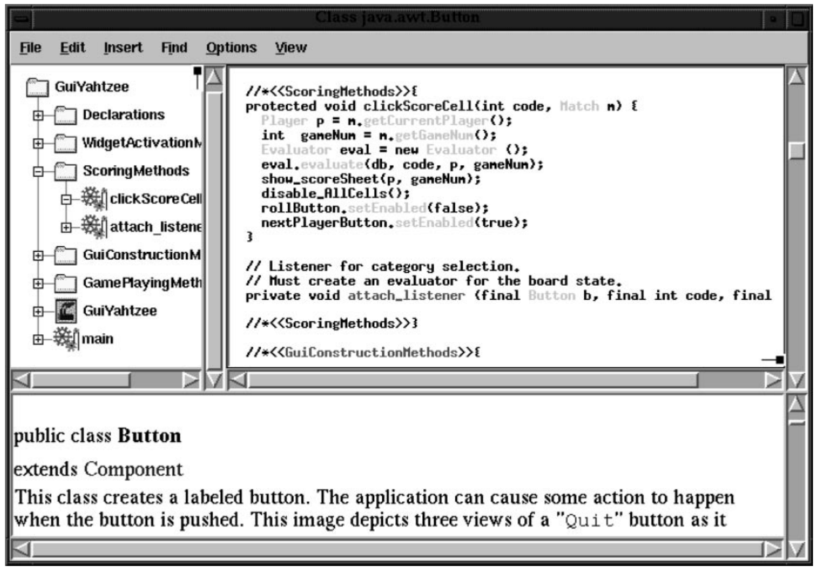


Fig. 6. Expanding abstractions, and inspecting object details.

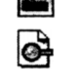
and invocations of super constructors. The text of every method invocation has a hypertext link attached to it (coloured green) allowing easy inspection of the associated method details. The declaration of every object variable is similarly linked to the associated class. When these links are clicked, if a class file of the object type is available on the user's class path,[2] then the class details are displayed in a new Jaba window. Otherwise, if javadoc documentation of the class is available then it is displayed in the HTML viewer at the bottom of the window and in an external web-browser if the appropriate options are set (using the Options menu). For example, the bottom half of Fig. 6 shows the javadoc documentation for the Button class. This was displayed when the user clicked the Button hypertext link associated with the declaration of the first parameter in the attach_listener method. When the user clicks on method invocation links, Jaba or javadoc immediately scroll to display the appropriate method description.

3.2. Tailoring the representation of context

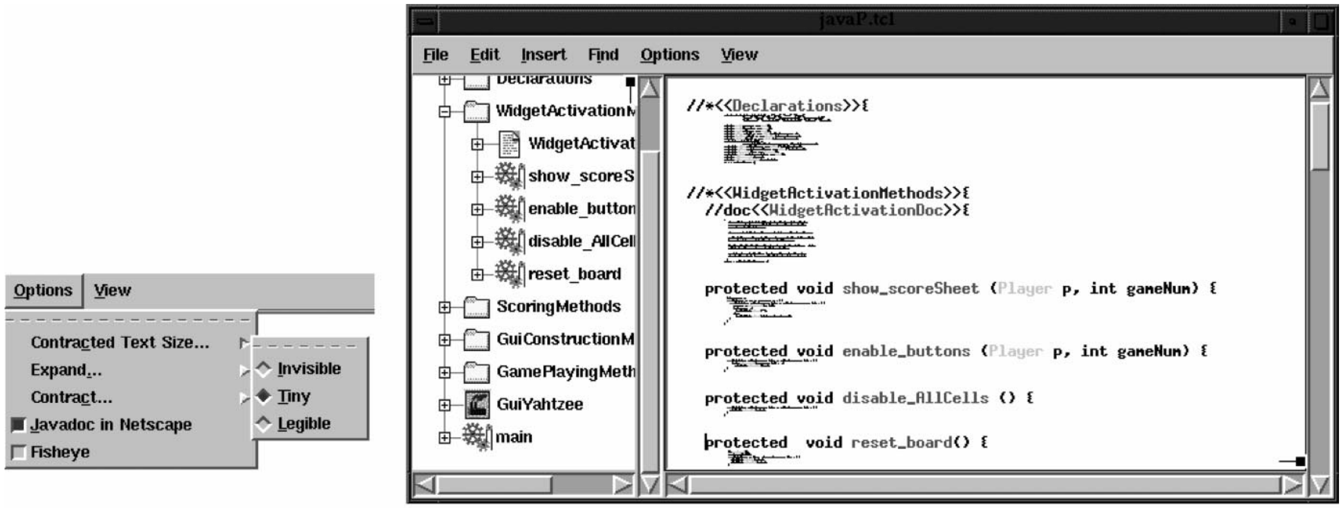
Figs. 5 and 6 show no contextual information about the contents of unexpanded chunks-all chunks in Fig. 5 are unexpanded, and in Fig. 6 chunk attach-listener in the text-edit window is unexpanded. This is similar to the approach described by Furnas (Section 2.3) in which suppressed information is completely hidden from the user.

Jaba allows users to tailor the representation of the abstracted information by selecting one of three text-sizes for the suppressed text (Fig. 7(a)). The 'Invisible' option completely suppresses the abstracted details (as shown in Figs. 5 and 6). The 'Tiny' option, shown in Fig. 7(b), provides limited contextual information about the suppressed information contained within a chunk. Although the text is not legible, the tiny option provides indications of the amount of suppressed information, its overall structure (apparent from indentation and from the number of red or blue portions which represent further abstractions), and limited information about the contents-blocks of green, for instance, reveal many declarations. The extreme miniaturisation of the 'tiny' font assures that minimal screen real-estate is dedicated to contextual information. The 'Legible' option renders suppressed text in a very small, but just legible, font. This option is a trade-off between the detailed views provided by expanding chunks and the broad views that are enabled by hiding and miniaturising suppressed chunks.

Table 1
Five chunk types supported by Jaba and their iconic representation

Chunk type	Icon	Comment
Generic abstraction		User-defined generic chunks. Used, for example, to group a set of related methods
Documentation		User-defined documentation chunks
Methods		Jaba automatically detects methods and stores their contents as chunks that can be contracted and expanded
Constructors		Constructor methods are automatically detected
Statement blocks		Jaba automatically detects statement blocks contained in loops and conditionals

[2] The Java class-path determines where to search for source-code associated with java classes.



(a) (b)

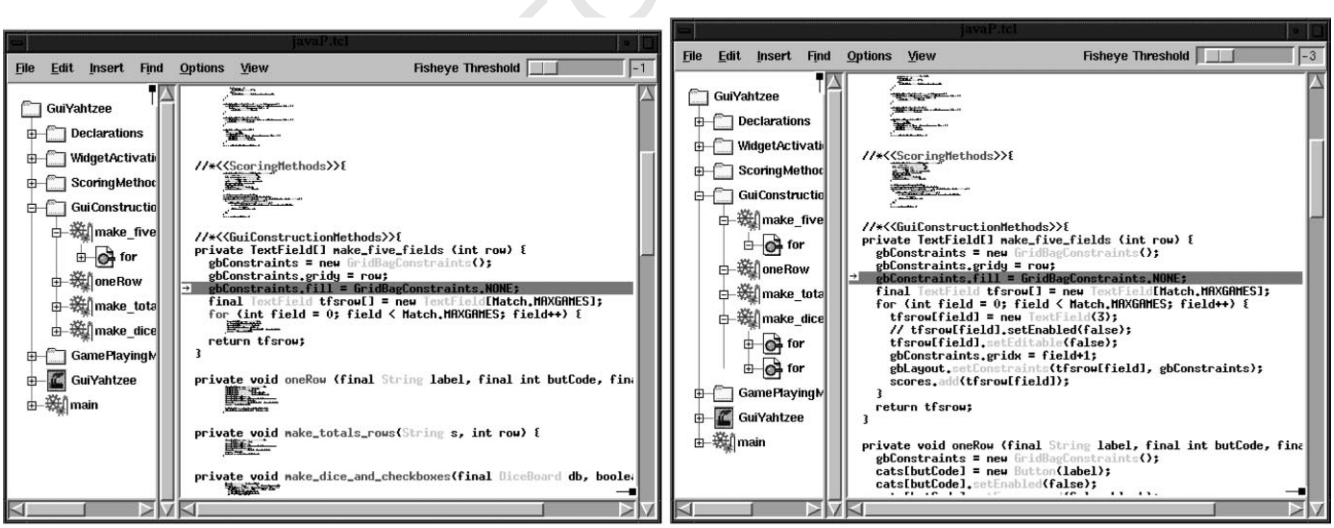
Fig. 7. Selecting and displaying 'tiny' text for abstracted program details; (a) Selecting the 'tiny' size; (b) 'Tiny' text revealing the context of abstracted chunks.

3.3. Creating chunks

Section 2.1 noted that traditional interfaces to the mark-up of literate programs introduce a second layer of syntax on top of the programming language. This raises the possibility of syntax errors in the mark-up of the literate structure.

Part of Jaba's chunking structure is automatically extracted from the program code, without the need for any additional mark-up-methods, inheritance, loops and conditionals, for instance, are all automatically extracted, as are the hypertext links to other classes and their methods. When the user chooses to explicitly create new abstractions, the new mark-up is embedded within Java comments. Although the user can enter the mark-up for new chunking structure by typing it directly, the normal way to do so is through menu options.

To convert an already existing section of code or documentation into a chunk, the user first selects the region to be chunked and then selects the 'Chunk the selection' option from the 'Edit' menu. To create a new chunk before its contents have been written, the user selects 'Add chunk' from the 'Insert menu'. In either case, a pop-up dialogue box prompts the user for a chunk name and type. The type can be either 'Abstraction'-for generic abstractions such as a grouping of related methods-or 'Documentation'. The appropriate syntactically correct comments are then added to the text to mark-up the new chunking structure. There are no limits to the nesting depth of the chunk structure.



(a) (b)

Fig. 8. Fisheye selection of suppressed regions. (a) Threshold value −1; (b) Threshold value −3.

Table 2
A program segment with API, Distance and DOI values for focal points *A* and *B*

Focus	Code	API	*A* Dist	*A* DOI	*B* Dist	*B* DOI
	IF (x < y) THEN	−1	1	−2	3	−4
	FOR i = 1 TO 10 DO	−2	1	−3	4	−6
	x = x + 1;	−3	2	−5	5	−8
	y = y * 2;	−3	2	−5	5	−8
	z[I] = I;	−3	2	−5	5	−8
	END;	−2	1	−3	4	−6
⩾ A	m = z[I];	−2	0	−2	4	−6
	ELSE	−1	2	−3	2	−3
	FOR x = 1 TO 10 DO	−2	3	−5	1	−3
⩾ B	z[x] = x;	−3	4	−7	0	−3
	END;	−2	3	−5	2	−4
	END;	−1	2	−3	3	−4

3.4. Shortcuts for exploring abstractions

Several system capabilities are intended to assist users in rapidly attaining the 'right level' of abstraction in their visualisation of classes. A variety of short-cuts, accessed under the 'Options' menu (Fig. 7(a)), allow users to expand or contract specific chunk types within the class. Through this menu, users can selectively contract or expand all chunks of each semantic type (generic abstraction, documentation, methods, constructors, or statement blocks), or they can choose to contract or expand all chunks regardless of type. Expanding all chunks provides a standard 'flat' text-editor with hyperlinking to the objects referred to in the class.

The system also remembers prior levels of abstraction within any chunk, allowing users to quickly refer back to previously inspected program regions. For example, if the user expands five levels of detail within chunk *X*, they can contract all of that detail by clicking the top-level link to *X*. When the user next expands *X* it will automatically display the five levels of detail that it previously showed.

3.5. Automatic DOI display configuration

Jaba includes a 'fisheye' option (bottom of the options menu Fig. 7(a)) which automatically selects which chunks are suppressed and which are displayed. Selecting the fisheye option adds two elements to Jaba's interface (Fig. 8(a) and (b)): a 'fisheye threshold' slider widget appears in the top-right of the window, and a focal point identifier/selector is added to the text-editor.

The focal point identifier/selector is shown as a small arrow in the left-margin of the text-editor. The program line pointed to by the focal point arrow is highlighted. The focal point is relocated by vertically dragging the arrow, and when the arrow is released, the DOI formula (Section 2.3) is used to calculate whether each chunk in the program is displayed or suppressed. The threshold slider controls the *k* threshold value for determining the lowest DOI value to be displayed. Modifying the threshold value also causes the DOI formula to be called, with consequent changes to the suppression and display of program chunks. In Fig. 8(b) the user has decreased the threshold value from −1 (Fig. 8(a)) to −3, causing the for loop inside method make-five-fields to be expanded. The graphical overview window in Fig. 8(b) shows that methods oneRow, make-tota… and make-dice have also been expanded by decreasing the threshold value.

In implementing Furnas's DOI formula, we found it necessary to normalise the DOI values to ensure that at least one chunk has a DOI value of −1. Consider a programmer moving from focal point *A* to focal point *B* in the program shown in Table 2. The API and distance values for focal points *A* and *B* are shown in Fig. 9 — the paired values in parentheses identify the distance of each abbreviated program line from focal points *A* and *B* respectively. The un-normalised DOI values are shown in the table. Assuming that the user initially focuses on Point *A* with a threshold value of −2, all program lines will be suppressed except for the focal line 'm = z[i];' and the conditional statement that provides its context 'IF (x < y) THEN'. When the user moves to focal point *B*, *all* program lines, even the focal point, will be suppressed because the highest DOI value (−3) is lower than the threshold. Normalising the DOI values assures that focal information is displayed, consequently saving the user from having to continually modify the threshold value.

In Jaba, the DOI formula's automatic selection of chunks for suppression does not affect the user's ability to explicitly tailor the level of detail in the display through the hypertext links or graphical overview.

3.6. Other capabilities

3.6.1. Superclasses and method overriding

Jaba automatically provides hyperlinking to super classes, super constructors and overridden methods. Text referring to super classes and super-constructors is coloured green for consistency with links to other object classes and their methods (Section 3.1). Overriding methods are linked

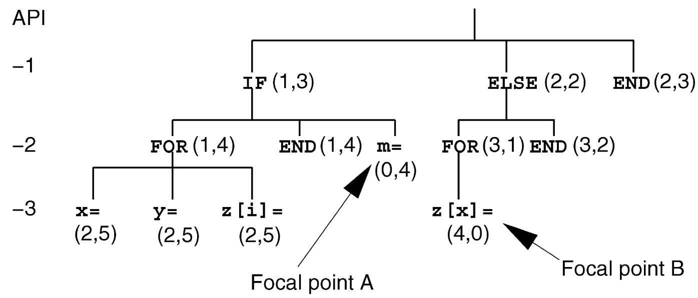


Fig. 9. API and Distance values for each program line (abbreviated) with focal points *A* and *B* in the program segment shown in Table 2. Values in parentheses show the distances from focal points *A* and *B* respectively.

with the method that they override by a small up-arrow icon which is displayed in the text immediately after the name of the method.

3.6.2. Dynamic parsing of text additions

As the user types new program lines into the text editor, the lines are automatically parsed and the necessary hyperlinks are added. Currently each line is parsed only when the newline key is pressed. This will often be too late to help programmers who want to use the object's methods within the current line. Ideally, Jaba would allow users to dynamically select methods or data-fields from menus associated with each object variable in a similar manner to that supported by systems like Jbuilder[3], VisualCafé,[4] and Visual J++[5]

Dynamically updating Jaba's state to reflect the programmer's modifications to the program is a major technical challenge. The program is almost certain to become temporarily syntactically incorrect during the programmer's editing changes: for instance, missing close-braces for methods, loops, conditionals, etc. One solution would be to use a syntax-directed editor, such as the Cornell Program Synthesizer [26], to ensure that the program is constantly syntactically correct. We strongly suspect that programmers would resist such functionality because of the constrained workflow that it imposes on program exposition. Instead, Jaba supports two-levels of parsing. As each new line of program code is typed, it is scanned against regular expressions to determine its form. If potential hypertext links are detected, then they are added when the return key is pressed. The chunking structure of the program is not dynamically updated. Instead, the complete class is reparsed only when the programmer explicitly requests that the class be 'rescanned' (selected from the file menu). Jaba does not currently offer debugging assistance when the chunking mark-up of the program, or the program itself, contains syntax errors.

[3] JBuilder is a registered trademark of Borland International Inc.
[4] Visual Café is a trademark of Semantec.
[5] Visual J++ is a trademark of Microsoft Corporation.

3.6.3. Linkages with the java compiler and virtual machine

Jaba is linked with the Java compiler and virtual machine. If the class displayed in a Jaba window contains a main method, then the 'Compile and Run' menu option under the 'File' menu is active. Selecting this option compiles all of the classes necessary to run the class, and runs the program in the Java virtual machine.

4. Design considerations

This section discusses the major design considerations that shaped the design and implementation of Jaba. By making Jaba's design rationale explicit, we aim to aid the reusability and repeatability of the work on Jaba. The design considerations, discussed in Sections 4.1–4.3, are separated into three categories that address the following questions:

1. How should program abstractions (or chunks) be formed?
2. How should the user interface support tailoring levels of program detail?
3. What additional program interlinking capabilities are required?

4.1. Forming abstractions in the program

In order to allow the user to tailor the level of program detail, systems such as Jaba must form a structural representation of program content. There are many possible approaches to extracting this structural information. Literate programming systems such as noweb (Section 2.1), for example, require that the structural information is explicitly specified by user-defined textual markup in the program source. Other systems automatically extract structural information using knowledge of the syntactic rules of the programming language (Section 6). Jaba uses a hybrid of these approaches, automatically detecting 'natural' abstractions in the program code (such as methods, loops and conditionals), while also permitting the user to explicitly add their own abstractions.

A more complex issue is how to apply literate programming chunking concepts within an object-oriented programming environment. Knuth described literate programming as an alternative to top-down or bottom-up design, allowing programs to be expressed and read in a ‘psychologically correct order’. Object-oriented programming, in contrast, focuses on reuse of well-encapsulated individual object descriptions; small program units then tie the objects together into programs. Object encapsulation makes the notion of a ‘psychologically correct order’ a weak one in object-oriented programming.

In designing Jaba, we decided to limit chunk encapsulation mechanisms to the text contained within individual Java class files. The primary motivation for this decision stems from concerns about the programmer’s familiarity with the class representation provided by the system. Each Jaba text-editing window is limited to the same text extent as the flat text-editors that programmers would normally use to edit Java classes, consequently once all chunks are expanded each window provides a ‘standard’ flat text representation of the class contents. If Jaba’s editing windows could include text segments from more than one class file (through a sophisticated implementation of literate chunking structure) then there would be a potentially confusing inconsistency between the contents of the window display and the user’s knowledge of what ‘should’ be contained in Java class files.

4.2. Interface for tailoring levels of detail

There were two major considerations in designing the interface for tailoring the level of program detail revealed. First, where and how to reveal the details of expanded chunks, and second, what events should trigger chunk expansion and contraction.

Østerbye’s [20] hypertext system for object-oriented literate programming in Smalltalk displayed each expanded chunk in a new window. Jaba, in contrast, displays the content of each newly expanded chunk in-line within the text-editor window in a manner that is similar to folding editors. Three factors motivated this decision. First, creating a new window for each new chunk is likely to raise a substantial user-interface overhead in window management. In the worst case, n classes each with m chunks will result in $n \times m$ windows. With in-line expansion, the maximum number of windows is equal to the number of class files. Second, an in-line representation of the class file is likely to be more familiar to programmers than the fragmented view provided by multiple windows because when all chunks are expanded in-line the window provides a standard ‘flat’ text editor. Third, in-line expansion maintains the context of each node within its surrounding information space. Chunks may be co-located within the class file for specific reasons-in-line expansion maintains this co-location but separate windows would not. Finally, Jaba’s interactive graphical representation of chunking structure is intended to aid perception of the structural relationship between chunks in the class file.

In determining what events should trigger chunk expansion and contraction, we strongly favoured explicit user control over the level of detail revealed. Implicit schemes-such as the automatic suppression of chunks when users relocate their focus in Furnas’s original fisheye view system-will sometimes incorrectly suppress chunks that the user wishes to see (the DOI formula is a heuristic assessment of *likely* interest). In Jaba, the main mechanism for control over the level of detail is through explicit selection of chunk names, either in the graphical overview or in the text-editor. Even when the fisheye view mechanism is activated (Section 3.5), explicit user selection of chunk names overrides the level of detail provided by the DOI formula. A further enhancement, not yet implemented in Jaba, would be to allow the user to lock certain chunks so that they cannot be expanded or contracted by the DOI formula.

4.3. Additional hypertext interlinking

Section 4.1 discussed the design rationale for choosing to limit Jaba to intra- rather than inter-class chunk structures. A consequence of this decision is that inter-class relationships must be managed through other mechanisms. Jaba parses all object variable declarations and instantiations, and these are linked to appropriate classes. Clicking the hypertext link associated with the class name causes either the class to be displayed in a Jaba window or the javadoc for the class to be displayed in the javadoc window (and/or web browser according to the set options). Method invocations from object variables are also linked to the associated classes, but access to data-fields from object variables are not. The rationale behind this decision was a trade-off between the utility of linking to data-fields and the display clutter of adding more links to the class display. Further enhancements to the object interlinking, not yet supported by Jaba, include dynamic selection of object methods from pop-up menus associated with object variables in a manner similar to that provided by commercial systems such as JBuilder, VisualCafé, and Visual J++.

5. Discussion and further work

Table 3 provides a summary of Jaba’s interface and functionality across eleven categories of system properties that we believe are desirable. These properties provide a distillation of our experiences in designing, implementing and using Jaba, combined with recommendations extracted from related work. Only properties 10 and 11, clarified in the table, have not been introduced in preceding sections of the paper. Summary information for Visual J++ is included in the table to help clarify Jaba’s primary differences from a current commercial system.

Commercial systems such as JBuilder, VisualCafé and Visual J++ support some of the features offered by Jaba, and they offer other capabilities that Jaba does not yet support (see Table 3). In particular, Jaba’s text-editing

Table 3
Summarising Jaba's capabilities, and comparing them to a standard commercial software development tool (Microsoft's Visual J++)

	Property	Jaba	J++	Comment on Jaba's support
1	Integrated environment for editing and browsing	√	√	Supports 'abstracted' browsing of documentation as well as editing details (unlike Javadoc which only supports abstracted browsing)
2	Automatic extraction of semantic 'abstractions'			
	• Methods	√	√	
	• Loops and conditionals	√	✗	
	• User-defined chunks	√	✗	Users can group related units of code into 'chunks', and they can define documentation chunks. Chunks can be nested
3	Light-weight creation of abstractions	√	✗	Existing text can be chunked by selecting it, making a menu-selection and naming the chunk
				Chunks can be created in advance of text by menu-selection and naming
				User-defined abstractions can be classified as 'generic' or 'documentation'
4	Easy transition between levels of abstraction	√	√	Hypertext links in text window expand and contract chunks
				Plus/minus icons in graphical tree window expand and contract chunks
				Shortcuts to expand/contract all chunks of specific types
				Shortcuts to previously visited levels of abstraction
				Option for automatic detail configuration through fisheye view
5	In-line expansion of abstracted details	√	✗	Extent of expanded region denoted by temporary shading
6	Support interactive visualisations of the object structure	√	√	Dynamic configuration of graphical tree to reflect program display
				Navigational shortcuts through graphical tree
				Icons provide semantic information about chunk types
7	Contextual information about suppressed code	√	✗	Tailorable font-size for abstracted text: invisible, tiny and legible
				Tailorable representation of context (extent, structure, and contents) of suppressed code
8	Context-sensitive hypertext linking between classes	√	√ ✗	Hyperlinks dynamically computed for super classes, super constructors, over-ridden methods, object variable declarations and instantiations, and method invocations
				Automatic display of associated method in Jaba window or in javadoc on following a method invocation link
				J++, VisualCafé, etc, provide method name completion (which Jaba does not), but do not support hypertext navigation to the class
		✗		Jaba's identification of over-riding methods currently limited to one-level of inheritance
9	Integration with existing tools	√	√	Integrated with java tools (javadoc, compiler, and virtual machine)
				Integrated with Netscape (through Netscape's remote control capabilities, see http://home.netscape.com/newsref/std/x-remote.hfor display of javadoc
		✗		Jaba is not readily adaptable to other languages. Although its techniques are adaptable, it has not been written in a language-independent manner
10	Non-intrusive support	√	√	Fully expanded views provide a generic 'flat' text editor that does not require users to adopt the abstraction and chunking features
11	Enhanced presentation of source text	✗	√	Only minimal adoption of program display principles such as those of [1,2]
				Semantic information currently captured by Jaba could allow improved presentation in future work

capabilities are rudimentary, and it is unlikely that commercial programmers would be willing to exchange their proprietary software development environments for Jaba. Commercial development, however, was not a design goal. Rather, Jaba *explores* new interface paradigms for visualising, browsing, editing and documenting object-oriented programs, and it demonstrates powerful capabilities for working with programs at configurable levels of detail. Such capabilities are not yet available in commercial packages.

To date, Jaba has only been informally evaluated. It has been used in lectures to assist teaching introductory Java programming to first year undergraduate students (approximately 600 students in the class), and for teaching Graphical User Interface programming with Java Swing to second year students (approximately 150 students in the class). As a teaching aid, the ability to expand and contract program segments is particularly useful for directing students' attention to salient program segments. The hypertext linking to other classes and to Javadoc is also extremely useful for demonstrating inter-class relationships.

Eight post-graduate students have also used Jaba in a single thirty minute unstructured session for browsing the class files associated with a traffic-light simulation program (thirteen classes, and a total of approximately 3000 lines of code. All of the students were familiar with the traffic-light simulation program. Comments from the students were positive, with three of the eight describing the interface as 'cool'. More substantive comments focussed on the positive aspects of being able to 'block out the irrelevant stuff' and 'focus on what you are interested in'. Several of the students stated that they thought they would be able to find needed program sections more quickly using Jaba than their normal program editing tools.

One problem reported by two of the students arose when using the 'Legible' setting for suppressed program segments. With this setting, it is possible (but uncomfortable) to read the program text. The students reported that they had to 'squint' at the text to read it, prior to realising that they could expand the chunk by clicking on the appropriate chunk. None of the students reported this problem when using the invisible or 'tiny' settings for suppressed code.

There are many potential directions for further work. Jaba's text-editing environment could be improved to bring it closer to commercial systems, and more work on its typographical display of programs (property 11) would improve program visualisation. Another area for further work on the system would be to allow user-defined chunk types to be created (beyond the five identified in Section 3.1). This would enable a wide range of new capabilities: in particular, it could be used to support different documentation perspectives on the same code chunk, such as 'exposition' and 'rationale' perspectives [20].

The major focus of further work will be on evaluation. The most important question to be addressed is the following:

Do the interface and cognitive overheads of defining and configuring levels of abstraction outweigh the quantitative and qualitative benefits?

Quantitative benefits can be explored in a similar manner to Furnas's [8] investigation of search times and error rates in 'flat' text displays versus fisheye displays. Upcoming evaluations will measure search times and error rates in finding lines of program code that cause Java compilation errors.

6. Related systems

The Tioga editor within the Cedar programming environment [27] stored documents in a tree structure of nodes. This allowed users to successively reveal levels of document details, or all levels up to a certain depth. Although users were able to expand and contract *global* levels of details, it appears that they were unable to selectively inspect inner levels of detail along a specific branch. This limitation would prohibit the simultaneous visualisation of the details of two distant regions in the document. It is not clear from the paper how Tioga's abstraction capabilities were applied to program code.

Several systems have applied holophrasting techniques (Section 2.2) to programming languages. The contraction and expansion of text within programs can be based on program constructs such as statement blocks and procedure definitions, or on the formal properties of the programming language's grammar. BNF syntactic rules specifying the allowable derivations from 'non-terminal' symbols to lower-level non-terminals and to 'terminals' can be used to store the program as a hierarchy of specialisation. In the EMILY system [10], for instance, users constructed, modified, and visualised program text through BNF-based holophrasts. The primary difficulty with grammar-based holophrast abstractions is that they require programmers to work through the formal levels of language. Programmers must therefore have a thorough knowledge of the language's grammar, and they cannot make 'shortcuts' through the levels of syntactic decomposition-for instance, even if the programmer *knows* that she wants an if statement she must still navigate through the syntactic rules that expand the grammar's non-terminals into an if statement.

None of the systems reviewed above, nor the fisheye program visualisations presented in Section 2.3, provide contextual information about the contents of abstracted units when they are suppressed. Holophrasting, folding editors (such as Tioga) and Furnas's program fisheye views totally elide [2] suppressed text, replacing it with ellipses. They therefore offer no indication of the extent, contents and structure of the suppressed program fragments. Smith, Barnard and Macleod [25] described a variant holophrasting text suppression technique called 'compaction' in which line-breaks are removed to display several lines of code on the same line. The tailorable views of suppressed details offered by Jaba are, to our knowledge, the first investigation into the use of scalable fonts in support of contextual awareness in programming environments.

Several researchers have investigated fisheye-based text visualisation. Keahey and Marley [13] performed an experiment with a variation of fisheye text to determine its effectiveness in helping users search through structured text. The results indicate that users preferred fisheye views for certain searching tasks, but that none preferred it for reading. Their implementation of the fisheye scheme was unusual in that text suppression was achieved by decreasing (to negative values) the text's inter-line gap. This caused dense text that wrote over the top of neighbouring lines.[6]

Scalable fonts can be reduced to one pixel per line of text, but even at this severe level of miniaturisation it will be impossible to display large text files within a single display space without scrolling. The Information Mural [12], however, demonstrates a variety of display scaling techniques including text display schemes that require less than one pixel per line.

Within synchronous groupware research, several prototype fisheye text systems have been developed to experiment with new ways of allowing users to stay aware of each others' actions in shared text-based information spaces [9,28]. These prototypes do not support programming, nor do they provide any support for moving between levels of abstraction. They do, however, support tailorable levels of text magnification in a similar manner to Jaba.

Together/J[7] [19] is an extensive commercial Java and C++ software development environment. It integrates many of the capabilities of UML object modelling [4] including package and class diagrams into its support for Java programming. Modifications within Together/J's text editor are immediately reflected in the corresponding class-diagram editors, and vice-versa. Equivalent capabilities could (and should) be supported by Jaba. The levels of abstraction supported by Together/J's class diagram editors are equivalent to those of javadoc-package and class. Users are unable to create new abstractions that correspond to their own cognitive units in the program, nor can users successively reveal inner levels of detail within the abstractions supported by the system.

7. Summary

The Jaba system presented in this paper demonstrates a novel and synergistic integration of four user-interface techniques that have been proposed to assist programmers: literate programming, holophrasting displays, fisheye visualisation techniques and hypertext. Literate programming supports programmers in dividing their programs into cognitive 'chunks' that are linked to other chunks. Holophrasting displays allow programmers to tailor the level of detail revealed in an information space by suppressing portions of the text. Fisheye techniques provide sophisticated visualisations of suppressed text that offer a trade-off between the provision of contextual information and use of display space. The integration of these techniques in Jaba enables programmers to configure their displays to reveal only the program details that are salient to their task while suppressing superfluous 'clutter'. Contextual information on the extent, structure and contents of the suppressed program text can be displayed through customisable miniaturised renderings of the text. Extensive automatically generated hypertext links facilitate rapid navigation between different levels of detail and between interlinked object classes and their contents.

To date Jaba is a proof of concept system that has been used to edit and modify several small Java programs. Although implementation details such as the absence of integrated debugging support and it's relatively crude text-editor preclude Jaba's viability in commercial software development, there are no reasons why the abstraction, visualisation and hypertext techniques demonstrated by the system should not scale-up successfully. The results of preliminary informal evaluations of the system are encouraging.

[6] Jaba assigns different font-sizes for the levels of magnification it supports, therefore text is not over-written.

[7] Together/J is a registered trademark of Object International, Inc.

Acknowledgements

This research is supported by New Zealand Royal Society 'Marsden' grant. Many thanks to Saul Greenberg at Calgary, Gerhard Fischer and the L3D research group at Boulder, Kai-Uwe Loser at Dortmund, and Warwick Irwin and Matthew Smith at Canterbury for comments on this work.

References

[1] R. Baecker, A. Marcus, Design principles for the enhanced presentation of computer program source text. Human Factors in Computing Systems II, Proceedings of the CHI'86, pp. 51–58, 1986.

[2] R.M. Baecker, A. Marcus, Human Factors and Typography for More Readable Programs, Addison-Wesley, Reading, MA, 1990.

[3] D.R. Barstow, H.E. Shrobe, E. Sandewall (Eds.), Interactive Programming Environments McGraw-Hill, New York, 1984.

[4] G. Booch, I. Jacobson, J. Rumbaugh, The Unified Modeling Language User Guide, Addison-Wesley, Reading, MA, 1998.

[5] A. Cockburn, N. Churcher, Towards Literate Tools for Novice Programmers, ACM Australasian Computer Science Education Conference'97, Melbourne, Australia, 2–4 July, ACM Press, 1997 pp. 107–116.

[6] J. Conklin, Hypertext: an introduction and survey, IEEE Computer 20 (9) (1987) 17–41.

[7] L. Friendly, The design of distributed hyperlinked programming documentation, Proceedings of the International Workshop on Hypermedia Design, Montpellier, France, 1–2 June, pp. 151–173, Springer, 1995.

[8] G.W. Furnas, Generalized Fisheye Views, Human Factors in Computing Systems III Proceedings of the CHI'86 Conference, North Holland/ACM, Amsterdam, 1986 pp. 16–23.

[9] S. Greenberg, C. Gutwin, A. Cockburn, Awareness Through Fisheye Views in Relaxed-WYSIWIS Groupware, Proceedings of Graphics Interface Conference, 21–24 May, Toronto, Canada, Morgan-Kaufmann, Los Altos, CA, 1996 pp. 28–38.

[10] W.J. Hansen, User engineering principles for interactive systems, in: D.R. Barstow, H.E. Shrobe, E. Sandewall (Eds.), Interactive Programming Environments, McGraw-Hill, New York, 1984, pp. 288–299.

[11] M. Heinrichs, Evaluating a Focus + Context Zoom Interface in Complement with Hypertext as a Program Understanding Tool, 1998. MSc Thesis. Computer Science, Simon Fraser University, Vancouver. http://www.cs.sfu.ca/~heinrica/personal/CZoom/.

[12] D.F. Jerding, J.T. Stasko, The information mural: a technique for displaying and navigating large information spaces, IEEE Transactions on Visualization and Computer Graphics 4 (3) (1998) 257–271.

[13] T.K. Keahey, J. Marley, Viewing text With non-linear magnification: an experimental study. Technical Report, Computer Science, 215 Lindley Hall, Indiana University, 1996.

[14] D. Knuth, Literate programming stanford California: center for the study of language and information. CSLI Lecture Notes no. 27, 1992.

[15] D.E. Knuth, Literate programming, The Computer Journal 27 (2) (1984) 97–111.

[16] J. Lamping, R. Rao, P. Pirolli, A focus + context technique based on hyperbolic geometry for visualising large hierarchies, Proceedings of CHI'95 Conference on Human Factors in Computing Systems Denver, 7–11 May, pp. 401–408, 1995.

[17] Y.K. Leung, M. Apperley, A review and taxonomy of distortion-oriented presentation techniques, ACM Transactions on Computer Human Interaction 1 (2) (1994) 126–160.

[18] S. Mukherjea, Y. Hara, Focus + Context Views of World-Wide Web Nodes, Proceedings of the ACM Hypertext '97, University of Southampton, UK, 6–11 April, ACM Press, 1997 pp. 187–196.

[19] Object International, Inc. Together/ J Product family.http://www.togetherj.com/, 1999.

[20] K. Østerbye, Literate smalltalk programming using hypertext, IEEE Transactions on Software Engineering 21 (2) (1995) 138–145.

[21] N. Ramsey, Literate programming simplified, IEEE Software 11 (5) (1994) 97–105.

[22] M. Sarkar, M.H. Brown, Graphical Fisheye Views of Graphs, Proceedings of CHI'92 Conference on Human Factors in Computing Systems Monterey, 3–7 May, Addison-Wesley, Reading, MA, 1992 pp. 83–91.

[23] D. Schaffer, Z. Zuo, S. Greenberg, L. Bartram, J. Dill, S. Dubs, M. Roseman, Navigating hierarchically clustered networks through fisheye and full-zoom methods, ACM Transactions on Computer Human Interaction 3 (2) (1996) 162–188.

[24] B. Shneiderman, Direct manipulation: a step beyond programming languages (excerpt), in: R.M. Baecker, W.A.S. Buxton (Eds.), Readings in Human Computer Interaction: A Multidisciplinary Approach, Morgan Kaufmann, Los Altos, CA, 1987, pp. 461–467.

[25] S.R. Smith, D.T. Barnard, I.A. Macleod, Holophrasted displays in an interactive environment, International Journal of Man–Machine Studies 20 (4) (1984) 343–355.

[26] T. Teitelbaum, The cornell program synthesizer: a syntax-directed programming environment, Communications of the ACM 24 (9) (1981) 563–573.

[27] W. Teitelman, A tour through cedar, IEEE Transactions on Software Engineering 11 (3) (1985) 285–302.

[28] P. Weir, A. Cockburn, Distortion-Oriented Workspace Awareness in DOME, People and Computers XII (Proceedings of the 1998 British Computer Society Conference on Human–Computer Interaction), Sheffield Hallam University, Sheffield, Springer, Berlin, 1998 pp. 239–252.