Towards Literate Tools for Novice Programmers

Andy Cockburn and *Neville Churcher*

Department of Computer Science
University of Canterbury
Christchurch, New Zealand
andy, neville@cosc.canterbury.ac.nz

Abstract

Literate programming is a powerful technique that helps expert programmers integrate code and documentation in a manner that assists human comprehension. To date, tools for literate programming have had moderately complex syntactic requirements. Consequently, the programmers who stand to gain most from the clarity of the literate technique—novice and learning programmers—have been unable to do so.

This paper describes the potential benefits of literate programming environments within introductory programming courses. Design criteria for such environments are presented, and prototype systems demonstrating the criteria are described. Evidence of student enthusiasm for graphical user interfaces for literate programming is discussed.

1 Introduction

Computer science educators invariably teach the importance of 'top-down design', or 'stepwise refinement', within their introductory programming classes. Students are taught to carefully decompose large abstract problems into smaller problems that reveal successive levels of detail. The importance of documenting the rationale behind the structural decomposition is also espoused, and students are often reminded that their code comments will be assessed. Despite this advice, many novice programmers tackle their assignments head-on with ad-hoc development strategies. Consequently, the code they develop is a 'moving target' which makes dedicating time to documentation risky because the code is likely to change before the final version. If they are done at all, documentation and comments are often written in a cynical manner in order to gain extra marks *after* the program is completed. The result is that students fail to grasp the importance of documentation and comments, and they learn the bad habit of adding comments as an afterthought.

Literate programming [4] is a technique that allows programmers to design, document, and construct their programs in whatever order best aids human understanding. It is an elegant technique that promises to assist novice programmers. Unfortunately, current literate programming tools are designed for expert programmers, and they have crude user interfaces which require moderately complex syntactic understanding. The promise of literate programming is therefore inaccessible to novice programmers.

We are investigating first year programming environments which use graphical user interfaces (GUIs) for literate programming. We believe that these environments can play a pivotal role in reinforcing structured programming techniques, and that they can foster the students' perception of the importance of a symbiosis between documentation and code. For example, we contend that student awareness of program structure can be enhanced through mechanisms that include graphical visualisations of program content and hypertext links

between related components such as variable declarations and uses.

In this paper we describe the motivation for using literate programming as a teaching tool, and we describe our work on developing graphical user interfaces that enhance the usefulness and usability of literate programming techniques. Section 2.1 briefly describes literate programming. It also outlines the relationship between the pedagogical goals of introductory programming classes and the capabilities of literate programming methods. Section 3 details the design criteria that govern our development of literate environments for novice programmers. Section 4 describes our prototype novices' literate programming environment. Section 5 discusses our students' work on improved interfaces for expert literate programmers, and describes the radical change in attitude that our students' underwent when moving from text-based to graphical interfaces for literate programming. Conclusions are presented in Section 6.

2 Literate and Introductory Programming

This section briefly reviews literate programming and the primary educational objectives of introductory programming courses. The potential of literate programming techniques to help achieve some of these educational objectives is discussed.

2.1 Literate programming

The aim of literate programming is to make computer programs easier for humans to comprehend [4]. Using a literate programming tool, programmers can arrange the sequence of programming elements and their accompanying documentation in whatever order best suits human comprehension, 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' correspond to 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. Literate programs can be 'tangled' to produce code that is ready for processing by a compiler or interpreter. Alternatively, the chunks can be 'woven' to produce documentation that includes extensive cross-referencing and indexing of program elements. Literate techniques allow programmers to "tell the story" of their programs clearly and precisely, with their documentation integrated into the program, in a manner that is impossible with standard CASE tools. It has an enthusiastic user community [6] and can be used to construct large or small software systems [5, 2].

Figure 1a shows a portion of a literate program for quick sort, and the resultant woven output. In this example, the text-based literate programming tool `noweb` [8] was used to create and manipulate the literate structures. The `noweb` source document on the left shows some of the syntactic constructs that define the chunking structure of the program and identify program elements such as variables. The corresponding woven portion of the program (figure 1b) is typeset and contains cross-reference information to show the inter-relationship between chunks and their uses, and between variables and their definitions.

The main limitation of current literate programming tools such as `noweb` is their primitive user interface. Literate programmers must learn an abstract syntax which specifies the chunking structure and the cross-referencing within the program. Mistakes produce syntax errors when the program is tangled or woven. For these reasons, Knuth [4] did not advocate the use of literate programming for students or hobbyists.

Modern graphical user interfaces, however, can overcome these problems through their ability to provide "syntactic correctness" [9]. Graphical user interfaces can mask the details of the typographical syntax from the user while maintaining the full range of functionality. `Noweb` programmers, for instance, identify variable declarations within chunks through syntactic structures such as `@ %def i count j temp`. A graphical user interface could allow the user to make the same identification, without the risk of syntactic error, by selecting the variables in a text-widget and clicking a "Variable declaration" button. Graphical user interfaces also allow natural metaphors for chunking structures to be exploited: for instance, allowing chunks to be represented by graphical nodes which can be controlled through direct manipulation.

```
<<*>>=
MODULE quick;
IMPORT IO;
<<Constants, types, and global variables>>
<<The QuickSort Procedure>>
BEGIN
  <<Get some numbers>>
  <<Sort the Numbers>>
  <<Print the numbers>>
END quick.
@ 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 Procedure>>=
PROCEDURE QuickSort(VAR a : ARRAY OF INTEGER; left, right: INTEGER);
<<Local variables>>
BEGIN
  <<Sort and divide until there's nothing left to do>>
END QuickSort;
@ Recursively sort an array {\tt a} of integers. {\tt left} and
{\tt right} denote the leftmost and rightmost elements of the array.

<<Sort and divide until there's nothing left to do>>=
IF right > left THEN
  <<Get set by guessing a cut value and initialising indexes>>
  <<Sort array with respect to cut value>>
  <<Recursively sort array left of cut value>>
  <<Recursively sort array right of cut value>>
END (* if *);
@ 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 thread.

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

<<Sort array with respect to cut value>>=
REPEAT
  <<Find an out of order number from left>>
  <<Find an out of order number from right>>
  <<Swap them>>
UNTIL hi <= lo;
<<Undo Extra Swap>>
```

Figure 1a: Part of a literate program's source.

1a ⟨ * 1a⟩≡
```
MODULE quick;
IMPORT IO;
⟨ Constants, types, and global variables 3a⟩
⟨ The QuickSort Procedure 1b⟩
BEGIN
  ⟨ Get some numbers 2f⟩
  ⟨ Sort the Numbers 2g⟩
  ⟨ Print the numbers 2h⟩
END quick.
```

This program demonstrates the `QuickSort` algorithm. It reads a list of numbers from the standard input, sorts them, and writes the sorted results to standard output.

1b ⟨ The QuickSort Procedure 1b⟩≡ (1a)
```
PROCEDURE QuickSort(VAR a : ARRAY OF INTEGER; left, right: INTEGER);
⟨ Local variables 3b⟩
BEGIN
  ⟨ Sort and divide until there's nothing left to do 1c⟩
END QuickSort;
```

Recursively sort an array `a` of integers. `left` and `right` denote the leftmost and rightmost elements of the array.

1c ⟨ Sort and divide until there's nothing left to do 1c⟩≡ (1b)
```
IF right > left THEN
  ⟨ Get set by guessing a cut value and initialising indexes 1d⟩
  ⟨ Sort array with respect to cut value 1e⟩
  ⟨ Recursively sort array left of cut value 2i⟩
  ⟨ Recursively sort array right of cut value 2e⟩
END (* if *);
```

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 thread.

1d ⟨ Get set by guessing a cut value and initialising indexes 1d⟩≡ (1c)
```
cutval := a[right]; (*arbitrary start for partition*)
lo := left - 1;
hi := right;
```

Uses `cutval` 3b, `hi` 3b, and `lo` 3b.

Arbitrarily pick the rightmost element of the array as the cut value for this pass.

1e ⟨ Sort array with respect to cut value 1e⟩≡ (1c)
```
REPEAT
  ⟨ Find an out of order number from left 1f⟩
  ⟨ Find an out of order number from right 2a⟩
  ⟨ Swap them 2b⟩
UNTIL hi <= lo;
⟨ Undo Extra Swap 2c⟩
```

Uses `hi` 3b and `lo` 3b.

Figure 1b: The corresponding woven documentation.

Figure 1: A `noweb` Literate program.

2.2 Introductory programming

There are normally three primary educational objectives in introductory programming courses. First, educators want their students to learn how to carry out structural decomposition (also called top-down design or step-wise refinement) so that they can break large problems into a series of smaller problems[1]. Second, students must be taught the mechanics of the language so that they can create and manipulate the data-types, control flow mechanisms, and so on, in order to solve the program-level requirements of their problem solution.

Third, lecturers wish to impress upon their students the importance of documenting design issues such as the purpose of procedures and functions, and the algorithms used. Part of the lecturer's aim is to impress on students the practicalities of writing programs for reuse by others, providing an early introduction to the concepts of software engineering. There are, however, several reasons why students may pay little attention to documentation within introductory programming courses.

[1]We include data-structure design and data-abstraction within this process of structural decomposition.

1. Students commonly have a perception that documentation is of secondary importance to creating executable code. This is especially true if the students have prior self-taught programming experience.

2. In introductory programming courses, which are normally large, it is often impractical to mark extensive program documentation.

3. Feedback on assignments often focuses on executable code—partly because it can be automatically evaluated, and partly as a consequence of point 2 above. The absence of feedback on documentation can further exacerbate problems with the students' perception of the low importance of documentation.

4. Code level comments are an impoverished mechanism for program documentation.

5. Tools that assist or encourage program documentation are rarely available in introductory programming courses.

2.3 Literate introductory programming

Wittenberg[11] describes the benefits that can be gained by using literate programming notation when lecturing on program development through step-wise refinement. He observes that the literate notation allows lecturers to leave place-holders for lower level details which can be revealed at the appropriate point in the program's development. The motivation is to generate class notes that reveal the processes of step-wise refinement. Without the literate notation, he argues, students' notes either show complete programs where the details of the refinement process have been added in-line (thus obscuring the refinement process), or they use pointers and arrows to show the relationship between portions of the fragmented code, with a resultant untidy presentation.

By extending Wittenberg's ideas, we believe that literate programming environments can actively support students' understanding of structural decomposition and of the symbiosis between code and documentation. We are developing supportive user interfaces to literate programming environments with the intention of providing these benefits without adding to the cognitive burden of learning the mechanics of programming.

3 Novices' Literate Programming Environments: Design Criteria

The list below describes the design criteria that guide our development of literate programming tools for novice programmers. Note that, to date, we are focusing on the facilities that will be provided by the *interface* to the environments, rather than on the underlying functional architecture. There are several freely available text-based literate tools such as `noweb` which could provide the functional back-end to our systems, but our interests lie in novel interface extensions to these systems.

1. **Foster the symbiosis between code and documentation.** The systems should explicitly support documentation that is tightly coupled with the associated code. By 'explicit support' we mean that the system should, by default, present the user with an interface element (such as a text or graph editor) that supports or displays documentation. This contrasts with 'passive support' such as standard code comments which require pre-emptive action from the user to insert into the program.

The mechanisms for documentation should not be limited to text. Documentation issues such as component coupling or cohesion may be best expressed graphically. Literate environments should therefore support multiple media for recording documentation.

2. **Support visualisations of the literate program structure.** The interface should provide helpful visualisations of the structure of the literate program. These visualisations should be interactive so that the user can use them to navigate to particular portions of the code or documentation.

The window on the left-hand side of figure 2 shows an example structural representation of the quick-sort literate program described in section 2.1.

3. **Non-intrusive support.** Not all students will want to use the literate environment, and not all types of program suit a literate style of development. Introductory programming courses invariably have some students who already have programming experience. The system should not force a literate style on students who do not wish to use it.

4. **Minimal syntactic requirements.** The system should ease learning structured programming techniques. Obviously, it should not add another layer of syntactic requirements to those of the programming language that they are trying to learn. The interface must therefore control the syntactic elements that are required by the underlying literate programming tool. The possibility of syntactic errors in the specification of literate structures can be minimised by providing 'syntactic correctness' in the interface [9]. For instance, syntactic errors brought about by mistyping chunk names can be minimised by allowing chunk names to be selected through a point-and-click interface.

5. **Tools for literate browsing.** The system should include tools that allow the user to browse the program in a variety of formats including the literate format, the woven documentation, and the tangled program. Additionally, the user should be able to expand and contract literate chunks of the program so that they can control the amount of abstraction and detail shown.

The potential range of tools for user support is extremely large. Facilities that we intend to explore include integrated debuggers which assist the student in accessing chunks containing syntactic and run-time errors, and "courseware" utilities which will lead students through development stages in sample solutions.

4 A Prototype Literate Environment for Introductory Programming

With our senior students we have developed several systems which support graphical user interfaces to the literate programming system `noweb`. These systems, discussed in section 5, focus on supporting *expert* programmers. In this section we describe a prototype literate programming environment which demonstrates the type of support that we expect *introductory* literate environments to provide. The prototype's interface is written in Tcl/Tk [7], and its functional back-end is provided by `noweb` [8].

The prototype (figure 2) provides three main mechanisms for viewing and editing the program: a structural overview, a program editor, and a documentation editor. The system maintains user interface "equal opportunity" [10] between the three interface mechanisms to ensure that all program views are mutually consistent: user actions in any one of the windows causes appropriate updates in the other two windows. For instance, adding a new chunk in one window causes the chunk to be displayed in the other two windows.

The structural view (left hand side of figure 2) allows the user to browse and control the overall structure of the program through a graphical representation of the chunking structure. Nodes in the graphical view correspond to 'chunks' in the literate program. The user can control the degree of detail and abstraction in the program and documentation editors by selecting and deselecting chunks in the structural view.

The program editor (middle of figure 2) provides a hypertext text-editor and text-viewer for the program code. Programs can be typed directly into the program editor, with or without literate programming constructs. Its primary hypertext facility is similar to that of a 'folding editor' [3], allowing the user to expand chunks to show their internal details, or contract them so that only their title is shown. Chunk names are shown as underlined blue text when contracted, and as shaded struck-through red text when expanded (in figure 2, only the top-level chunk "QuickSort Program" is expanded). Clicking on a chunk's name toggles between expanded and contracted states. When a chunk is expanded, its representation in the structural view is shaded, and its associated documentation is shown in the documentation editor. The effect of clicking on the chunk 'The QuickSort Procedure' in figure 2 is shown in figure 3 in which the details of the procedure are revealed to the next level of abstraction.

The documentation editor (right-hand side of figures 2 and 3) provides a text and graphics editor for the program documentation. Its hypertext facilities are similar to those of the program editor, allowing the user to expand and contract chunks. A documentation graphic belonging to the documentation chunk of 'The QuickSort Procedure' is shown in figure 3.

The user is free to create the literate structure of the program in whatever way they prefer. This can involve reverse engineering the literate structure, in which the user types (or loads) a complete non-literate program into the program editor, and then breaks it into literate chunks by selecting portions of the code for 'chunking' (using the 'Make Chunk' option from the 'Insert' menu), and finally the student names the resultant chunk. Students could be assigned a reverse-engineering exercise of this type to assess their comprehension of existing programs. Alternatively, the literate capabilities of the environment can be used to support top-down design using step-wise refinement. For example a student could use the structural view to create and name empty chunks which describe the high-level structure of the program, and then successively refine the lower-level details. There is no requirement, however, that programs within the environment be written in a literate style. Non-literate programs are displayed directly in the program editor, and a single top-level chunk representing the entire program is shown in the structural view.

We intend that the environment will assist the assessment and evaluation of programs and their documentation. Assessors will be able to load each student submission into the environment, view the overall structure of the program within the structural viewer, and selectively navigate to critical portions of the solution to view both the code and its associated documentation. Poor structure and absent doc-

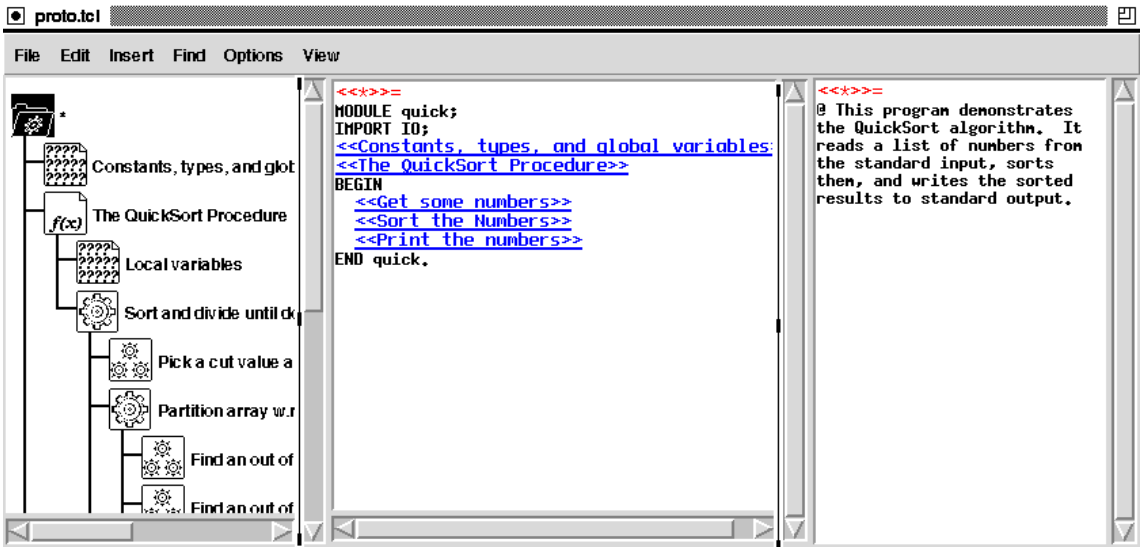

Figure 2: A prototype literate environment: structural overview, program editor, documentation editor.

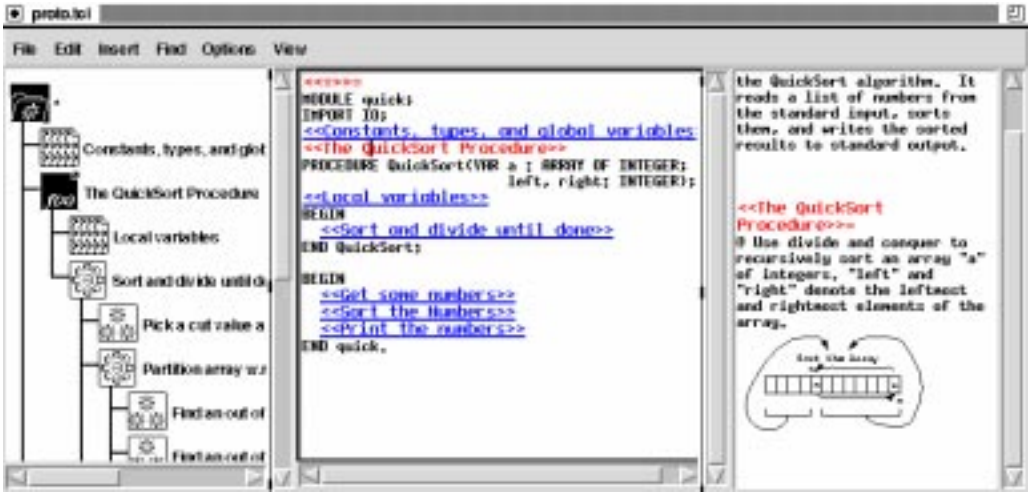

Figure 3: Modified system state having expanded 'The QuickSort Procedure.'

umentation should be readily apparent with minimal browsing.

5 Discussion

This section describes two of the fully-functional GUIs for expert literate programmers developed by our senior undergraduate students. It also discusses our experiences with students' attitudes to literate programming when moving from text-based literate environments to GUI-based literate environments. Finally, the section identifies directions for our further work with novices' literate programming environments.

5.1 Interfaces for expert literate programmers

In our final year undergraduate software-engineering course, teams of three students work throughout the year to extend and improve an existing software system [1]. In 1996 the software system was the literate programming tool `noweb` (see section 2.1). The initial stages of the project involved documenting `noweb`'s internal source code, and extending its facilities in several ways including a language independent pretty-printer. Later in the year students were required to design and construct a graphical

user interface that provided access to all of `noweb`'s facilities for expert programmers and software engineers.

Two of the resultant systems are shown in figure 4. Both of these systems provide a graphical representation of the literate program's structure (right hand window in group B's system, and left hand window in group K's system). There are several interesting features in both of these systems. For instance, group K's system uses semantic icons in the structural view to encode information about each chunk: folder icons denote separate files, cog icons denote functional units, and cross icons denote calls to chunks that are not yet defined, or which contain syntactic errors. Despite these powerful encoding mechanisms, programmers using the system must write standard `noweb` source into the program editor. In contrast, group B's system abstracts some of `noweb`'s syntactic requirements by providing separate text editors for code and documentation, but the overall presentation of the literate program maintains `noweb`'s syntactic mechanisms. Hypertext facilities, and search and browsing mechanisms are poorly or not supported in both of the systems.

Neither of the systems is suitable for use by novice programmers because they both require that the user can manipulate `noweb` syntactic structures. However, they successfully meet the objectives of providing full coverage of `noweb`'s functionality, while adding value through their enhanced interface mechanisms. They also provide some useful pointers for novices' literate programming environments.

5.2 Student Attitudes to Literate Programming

During the 1996 project described above, the class of 40 students underwent a startling change in attitude towards the value and potential of literate programming. Throughout the initial stages of the project, when they were studying `noweb` and extending its text-based capabilities, the general motivation-levels and morale of the project groups was low. Most teams did not believe that literate programming was useful, and they lacked enthusiasm for modifying "unuseful software". They were then required to produce sketched 'storyboards' of their proposed graphical user interfaces, and to describe their storyboards at a meeting with the course lecturers. Almost all of these initial designs were little more than file browsers which contained buttons to execute standard `noweb` command line options such as `noweave -index file.nw > file.tex`. The project groups almost uniformly did not believe that graphical user interfaces could help literate programming.

At their subsequent meeting, the project groups were shown a storyboard similar to figure 2 to reveal how GUIs could enhance and integrate the primary concerns of literate programming: the program's structure, and the symbiosis of code and documentation. During the following months, while the project groups were redesigning and implementing their systems, the general team morale and motivation was extremely high. Several students became almost evangelical in their attitudes towards literate programming! We believe that this is a strong indication of the potential of literate environments for use in education.

5.3 Future Work

The prototype described in the section 4 is still under development, but it serves as a useful 'point system' for demonstrating the potential of literate programming environments for novice programmers, and for shaping the facilities provided by these environments.

We have not yet tested the prototype on our target user-base—students in introductory programming courses. User testing is our primary item of further work. Prior to major user testing, however, we want to develop a thoroughly robust and extensive environment, and there are several directions for our further work on implementation, some of which are described below.

- Integration with programming tools (literate and otherwise). The environment will allow users to process their programs in a variety of ways, including weaving the documentation and tangling, compiling, and executing the program.

  Currently our introductory programming students use the CUTE[2] GUI environment for all their course work. We will extend this environment to include literate programming capabilities, and will integrate facilities such as a literate debugger which will help students access chunks that cause syntactic and run-time errors.

[2] Canterbury University Teaching Environment.

Group B's submission.

Group K's submission.

Figure 4: Functional `noweb` GUI.

- Hypertext facilities. Additional hypertext facilities will be provided in the program editor. These include support for rapid navigation between variable uses and their declarations, and a variety of search facilities.

- Replays of step-wise refinement. We are keen to investigate a 'program-builder' which will allow students to follow, in their own time, the process of step-wise refinement as prepared by the course lecturer. Students will be able to click through critical stages in the development of the program, and the documentation editor will describe these processes. We are excited by the potential of this self-paced and repeatable learning resource.

6 Conclusions

At the 1996 ACM ACSE conference Wittenberg [11] espoused the potential benefits of using a literate programming notation for use by lecturers when *presenting* the processes of step-wise refinement in introductory programming classes. While we agree with his observations, we believe that literate programming can play a much more active role in supporting students who are learning how to program.

In this paper we have described the motivation for providing literate programming support environments in introductory programming courses, and we have described

the primary design criteria for these environments. These criteria are as follows: promote the students' perception of the symbiosis between code and documentation; enhance their awareness of the program's structure using visualisation techniques; provide non-intrusive support that does not demand a literate programming style; do not require that the students learn another syntactic notation for the literate commands; and finally, provide tools that support browsing and searching through the literate program.

To exemplify such support environments, we described a prototype system. We also described our students' experiences in constructing user interfaces for literate programming environments, and noted the radical and positive change in their attitudes towards literate programming as they progressed from text-based literate environments to GUI-based ones. We are encouraged by the keen student enthusiasm for GUI-based environments for literate programming, and look forward to running extensive trials of our system with first-year programming students.

References

[1] N Churcher and A Cockburn. An immersion model for software engineering projects. In *ACM Australasian Computer Science Education Conference '97. Melbourne, Australia. 2–4 July.*, pages 163–169. ACM Press, 1997.

[2] C.W. Fraser and D.R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.

[3] RJ King and YK Leung. Designing a user interface for folding editors to support collaborative work. In G Cockton, SW Draper, and GRS Wier, editors, *People and Computers IX*, pages 369–381. Cambridge University Press, 1994.

[4] DE Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[5] D.E. Knuth. *The Stanford Graphbase: a platform for combinatorial computing*. Addison-Wesley, 1993.

[6] Literate programming library. `http://info.desy.de/user/projects/LitProg.html`, 1997. Deutsches Elektronen-Synchrotron (DESY), Hamburg, Germany.

[7] JK Ousterhout. *An Introduction to Tcl and Tk*. Addison-Wesley, 1993.

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

[9] B Shneiderman. Direct manipulation: A step beyond programming languages (excerpt). In RM Baecker and WAS Buxton, editors, *Readings in Human-Computer Interaction: A Multidisciplinary Approach*, pages 461–467. Morgan Kaufmann, 1987.

[10] H Thimbleby. *User Interface Design*. ACM Press, Addison-Wesley, 1990.

[11] L Wittenberg. Using literate programming notation in introductory programming courses. In *ACM Conference on Computer Science Education, Sydney, June 3–5*, pages 267–272, 1996.