TOWARDS MODERN LITERATE PROGRAMMING

Honours Project Report, 2001

Matthew Smith Dr. Neville Churcher (Supervisor)

ABSTRACT

Literate programming was invented by Donald Knuth as a technique for improved documentation of program understanding. It involves writing code and documentation in a single source document, ordered for comprehension by humans rather than computers. Despite its ability to produce software of higher quality and maintainability, the technique is not widely used. In this report, we present a comprehensive background of literate programming that shows what the methodology is currently capable of. We also isolate the factors that limit its mainstream use, forming a set of requirements for further work.

KEYWORDS: *literate programming, software comprehension, structured documentation*

CONTENTS

Chapter 1

INTRODUCTION

> The fellow who designed it, is working far away;
> The spec's not been updated for many a live-long day.
> The guy who implemented it, is promoted up the line;
> And some of the enhancements didn't match to the design.
> They haven't kept the flowcharts, the manual is a mess;
> And most of what you need to know, you'll simply have to guess.
>
> *David Diamond [11]*

1.1 The software comprehension problem

Maintenance programmers spend approximately half of their time simply trying to understand the function of program code [28]. This factor alone has been estimated as contributing anywhere from 30–90% of the cost of software over its entire life cycle [41]. Reducing this cost is therefore one of the greatest needs of software engineering, yet it remains largely unresolved. It is remarkable how much the above poem, written in 1976, applies today.

The problem centres around a *lack of communication of problem understanding*. It is during implementation that a programmer has formed the most complete mental model of the problem and method of solution. When this is not documented adequately, the understanding can be lost in the code or in the head of the programmer. The result is that maintenance programmers separated both in space and time must recover this understanding by, in most cases, referring directly to the code itself. Inadequately documented software has little chance of being re-used, no matter how efficient the implementation.

The primary cause of inadequate documentation is a common attitude among professional programmers and managers that it is of little use [29]. As shown in Figure 1.1, programming is a highly demanding activity. Modern software architectures often require the use of multiple languages and levels of abstraction at one time. The programmer (centre) must bridge the gap between their own mental representation of a solution and the representations expected by computers (right), in the form of programming languages. They must also communicate their ideas to potential users of the software, who may each have different purposes and needs (left). Because of the belief that this communication reaps no short term benefits, many programmers simply focus on the activity of programming, with documentation merely an afterthought.

Even if a programmer *is* motivated to communicate their understanding, traditional documentation techniques are usually inadequate for the task. Internal documentation is limited to the syntactic order demanded by the compiler and can only be used for low-level, textual explanations. Moreover, source code comments are not suitable for the documentation of understanding that occurs across *sections* of a software system. Higher-level external documentation can capture these aspects, but can become inconsistent with the code, especially when projects and teams evolve from an initial design or implementation.

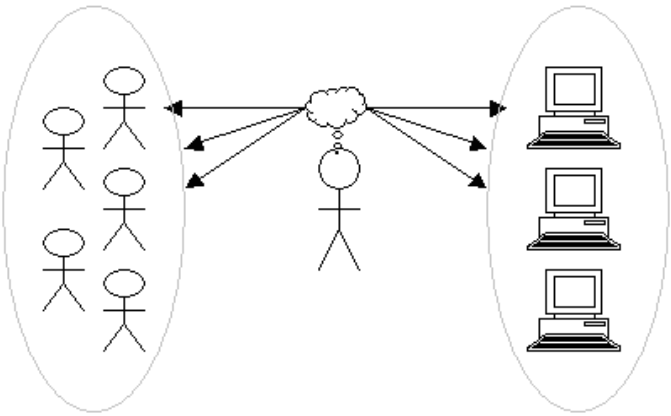


Figure 1.1: The demands on a programmer.

1.2 Motivation and approach

Literate programming [19] is a possible solution to the software comprehension problem described above. The technique involves writing documentation and program code in a single source document, psychologically arranged for comprehension by humans rather than computers. It provides significant incentives for programmers to document their understanding while they code, resulting in programs of higher quality and maintainability.

Despite these benefits, literate programming is typically only used by academics or expert programmers working on personal projects. This problem motivates our research, as we believe the methodology can and should be used to facilitate improved maintenance and re-use of real-world software systems.

In the next chapter, we present a fairly comprehensive background of literate programming, showing what the methodology and its supporting systems are currently capable of. In Chapter 3, we isolate the key problems and limitations of literate programming that contribute to its lack of widespread use. Our work concludes in Chapter 4 with a set of requirements for further work.

Chapter 2

BACKGROUND

> Let us change our traditional attitude to the construction of programs: instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do.
>
> *Donald E. Knuth [19]*

2.1 What is literate programming?

Literate programming was invented by Knuth [19] in the early 1980s as a solution to the software comprehension problem. Dissatisfied at traditional techniques for documenting programs, he made the observation that we should consider programs to be *works of literature*, aimed primarily at human rather than computer consumption. The name "literate programming" comes mainly from this point, but also from Knuth's desire to impose a moral commitment on others to avoid *illiterate* programming.

The methodology has three distinguishing characteristics:

Verisimilitude: Code and documentation are written *together* in the same source document. This integration ensures "active documentation" that evolves and is always consistent with program code. As shown in Figure 2.1, Literate Programming tools either *tangle* this source to produce computer-understandable code, or *weave* it to produce comprehensible documentation for humans. The integration also ensures the literate programmer is given strong incentive to explicate their mental understanding while they code.

Psychological arrangement: A literate program is primarily a communication to humans that should not be limited by the syntactic structure a compiler expects. Code can be decomposed into smaller chunks and explained in whatever order is most appropriate to aid comprehension. The programmer is not limited to a rigid top-down or bottom-up style; they can use whatever hybrid and combination of formality best fits the exposition. Moreover, they should include not only a discussion of the function and purpose of associated code, but also a problem statement, alternatives, background detail that is needed to understand the solution, and directions for potential maintenance users.

Enhanced readability: As a work of literature, a literate program should be presented in a form that enhances the readability and comprehensibility of code. Tools can provide a combination of pretty-printing, cross-referencing and indices to enhance readability. The programmer can use whatever combination of images, tables and text is deemed necessary to enhance the communication of program understanding.

Literate programming gives both an incentive and capability to produce high quality documentation that is inherently readable and comprehensible. It encourages the explicit documentation of program understanding that is *independent from individuals* and *preserved over time*. This

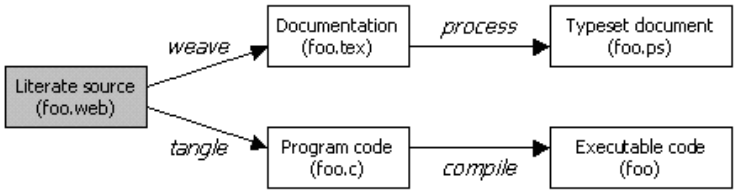


Figure 2.1: The two processes of literate programming. Tangling produces computer code, while weaving produces documentation for human consumption. Brackets show example file names.

benefits future users of the software and in particular should reduce maintenance costs, as less time needs to be spent reverse-engineering the program design and intent from the code. Well-documented software is much more likely to be adapted and re-used.

Practitioners of literate programming typically note one advantageous side-effect of the approach: being forced to clarify a solution to others can often bring forward the discovery of problems. It is well known that the act of explaining something can enhance one's understanding of it. Knuth points out that when programming literately, he no longer takes shortcuts that prove to be mistakes. He also pays more attention to producing quality solutions, because his work will be read by others. Williams [47] calls this "wholistic debugging"—instead of hacking together a solution, and then spending hours in an interactive debugger attempting to find out what is wrong (the technique of many programmers), literate programming encourages you to work this out beforehand. In other words, the role of debugging changes from correcting errors to *improving understanding*.

This is in part a justification for the additional effort required to program literately. There are many claims that the reduction in debugging time and production of more maintainable software outweighs the cost of documenting ones understanding [8, 35, 42, for example]. Unfortunately, as we will discuss in Chapter 3, there is little proof that this applies in real-world projects.

2.2 Components of a literate program

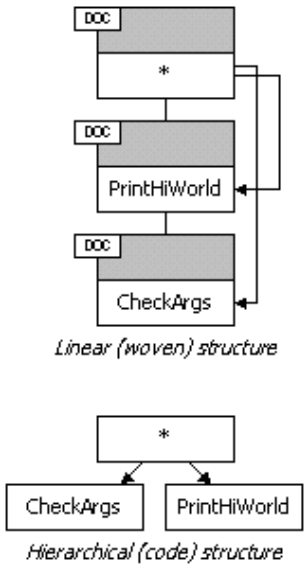
A simple literate program is shown in Figure 2.2(a). The first point to note is that it contains both documentation and code, organised into small sections or *chunks*. Literate programming systems all have differing syntax for distinguishing between the two; in this case, documentation chunks are unnamed and begin with the "@" symbol, while code chunks are named inside angled brackets. The example shown is for the `noweb` system [32], which has a simple set of 5 commands. Others, such as Knuth's original `WEB` system, often contain over 30 or 40 commands are are therefore much more difficult to learn. (`WEB` was introduced in Knuth's original 1984 paper [19], well before the popularisation of the World Wide Web (WWW). He chose the name because a complex system is best understood as a web delicately put together by simple parts and relationships.)

Chunks are the cognitive unit of a literate program. They can be of any size or granularity as deemed necessary by the programmer for best exposition. Furthermore, code chunks are *not* limited to abstractions of the underlying programming language. For example, Figure 2.2(a) contains no functions or sub-routines, yet still splits the code into three separate chunks. It would even be possible (although probably undesirable) to begin a chunk in the middle of a statement or clause. In other words, chunks are a *conceptual abstraction* supported by the literate programming methodology.

All systems have some way of specifying that a code chunk can contain any number of *nested* code chunks. In the example shown, the root code chunk "`*`" contains program text as well as two sub-chunk references to "`CheckArgs`" and "`PrintHiWorld`" (although not all systems allow both text and references within a single code chunk). Furthermore, as per the property of psy-

```
@ Here is a Perl program that simply
prints out |Hello, world!| the number of
times specified in the first argument.

<<*>>=
#!/usr/bin/perl
        <<CheckArgs>>
        <<PrintHiWorld>>

@ Printing involves a simple loop.  Line
breaks are added for clarity.

<<PrintHiWorld>>=
for ($i = 0; $i < $ARGV[0]; $i++) {
        print "Hello, world!\n";
}

@ We \emph{must} make sure, however,
that an argument was specified.

<<CheckArgs>>=
if (@ARGV != 1) {
        die "No argument specified";
}
```

(a) Literate source.



(b) Linear and hierarchical views.

Figure 2.2: A simple literate program in noweb format.

chological arrangement, code chunks can be explained in any order. For example, `CheckArgs` is defined as being before `PrintHiWorld` in the program, yet it is explained last. Similarly, the root chunk could have been moved anywhere; there is no requirement that chunks are referenced before they are defined. The goal is to allow the programmer to adapt the order as needed for clearer exposition of the program.

Unlike code chunks, documentation chunks are not explicitly nested, although formatting language commands, such as LaTeX's `\section` variants, can often be used to group parts of a document. In any case, the *primary ordering* of a literate program depends on the documentation, with code interleaved as necessary. This distinguishes literate programming from highly commented source code, where the primary ordering depends on the code, with documentation interleaved. Moreover, literate programs can include both external and internal documentation, and in a much more readable and extensive fashion than supported by traditional commenting. It is the woven literate document that facilitates greater maintenance and re-use.

The program in Figure 2.2(a) can be viewed as forming two distinct graphs or views, as shown in Figure 2.2(b). The first is the *linear* structure of the documentation. We have shown code and documentation grouped into sections, because in all literate programming systems it is implied that code *is described by* the immediately preceding documentation. Arrows indicate nested code chunk references. The job of the *weave* process, as per Figure 2.1, is to take this linear structure and beautify it for human consumption. As will be described in the following section, this includes typesetting, pretty-printing and indexing of chunks.

The second view in Figure 2.2(b) is the hierarchical structure formed by code chunk nesting. The job of the *tangle* process, also shown in Figure 2.1, is to re-order the linear document structure into this hierarchical form, as expected by the compiler. Tangling programs achieve this by recursively replacing all sub-chunk references with the actual text. For example, the Perl "Hello, world!" program would correctly contain the code of `CheckArg` before `PrintHiWorld`. Although only one level of nesting is present in this simple example, a literate program usually contain much deeper and more complex hierarchies.

2.3 Typical features

Despite their varied input syntaxes, literate programming systems share a common model based on the properties of verisimilitude and psychological arrangement. However, they differ greatly in support for additional features such as pretty-printing and indexing. This results in very obvious visual differences between the woven documents of each system. In this section, we describe some of these features and discuss the varying levels of support in general terms.

2.3.1 Language support

Due to the unique combination of program code and documentation, literate programming systems must provide support for both *document formatting languages* and *programming languages*.

Many of the earlier *language dependent* systems cater for only a single programming language, in order to provide extensive automatic support for pretty-printing, cross-referencing and indexing. The more recent systems sacrifice such features for *language independence*, thus avoiding the need to use a different tool and syntax for each language desired. These are much more applicable to modern software engineering, especially with the increase in development of multi-lingual software systems, such as those delivered over the web with both client and server-side scripting.

In terms of document formatting, most systems cater for TEX and / or LATEX (note how in the example of Figure 2.2(a), a LATEX directive is used within the third documentation chunk). TEX's support for complex mathematics can be of particular use when describing algorithms. More recently, support for the Hyper-Text Mark-up Language (HTML) is commonplace, which provides greater opportunities for multi-media documentation. Some tools provide their own mark-up language and are capable of converting from this to the more recognised forms.

2.3.2 Macros

Many systems provide support for macros similar to C's pre-processor. These were initially included by Knuth in his `WEB` system to overcome limitations of the Pascal compiler. For example, a macro can be used to simulate an array with dynamic bounds, as is commonly done in C systems. Knuth also advocated their use for enhanced readability, although these days higher-level languages tend to be more naturally understandable to humans than, for example, Fortran code.

Macros can be *simple* (text substitutions) or *parametric* (with arguments). Nested code chunks such as `PrintHiWorld` in Figure 2.2(a) are effectively simple macros for the purpose of tangling, which simply replaces chunk references with the actual text. None of the major literate programming systems support *conditional* macros, although both Knuth [19] and Williams [47] suggest a way to simulate their operation using internal comments of the underlying programming language. Knuth also describes how multi-parameter macros can be simulated with repeated single-parameter macros, as justification for limited support in `WEB`.

Other than for code expansion, macros are not essential to literate programming. Extensive use can complicate a literate program to the point where the macros themselves may actually require documentation. Moreover, Avenarius and Oppermann [1] note how macros can help spread programmer's mistakes "silently and effectively". In any case, systems without macro support can always be augmented with the use of an external macro processor if desired.

2.3.3 Pretty-printing

As mentioned in Section 2.3.1, language dependent systems contain the knowledge needed to provide automatic pretty-printing of program code. An extract from a document woven using a variant of Knuth's `WEB` system, `CWEB` [22], is shown in Figure 2.3. `WEB` and `CWEB` apply an extreme level of pretty-printing; they ignores the programmer's line breaks and choice of indentation, basing their decisions on the syntactic rather than lexical features of the code. At the other extreme, language independent systems copy program code verbatim into the woven document, usually displaying it in a typewriter font.

8. Now we scan the remaining arguments and try to open a file, if possible. The file is processed and its statistics are given. We use a **do** ... **while** loop because we should read from the standard input if no file name is given.

```
⟨ Process all the files 8 ⟩ ≡
  argc −−;
  do {
    ⟨ If a file is given, try to open ∗(++argv); continue if unsuccessful 10 ⟩;
    ⟨ Scan file 15 ⟩;
    ⟨ Write statistics for file 17 ⟩;
    ⟨ Close file 11 ⟩;
  } while (−−argc > 0);
```

This code is used in section 5.

9. Here's the code to open the file. A special trick allows us to handle input from *stdin* when no name is given. Recall that the file descriptor to *stdin* is 0; that's what we use as the default initial value.

```
⟨ Variables local to main 6 ⟩ +≡
  int fd = 0;      /∗ file descriptor, initialized to stdin ∗/
```

10.

```
#define READ_ONLY 0      /∗ read access code for system open routine ∗/
⟨ If a file is given, try to open ∗(++argv); continue if unsuccessful 10 ⟩ ≡
  if (file_count > 0 ∧ (fd = open(∗(++argv), READ_ONLY)) < 0) {
    fprintf(stderr, "%s: cannot open file %s\n", prog_name, ∗argv);
    status |= cannot_open_file;
    file_count −−;
    continue;
  }
```

This code is used in section 8.

Figure 2.3: Extract from a woven CWEB document.

There is much disagreement over the extent to which code should be pretty-printed, and whether it is in fact a requirement of a true literate programming system. Empirical studies by Baecker [2] and Oman and Cook [27] have shown that the use of enhanced typography, such as multiple fonts and varied character spacing, can enhance the readability of code by as much as 25%. On the other hand, Ramsey and Marceau [33] conducted a real-world team project using WEB, and found that the single biggest resistance to the literate programming methodology surrounded the loss of control of code appearance.

It would seem that an appropriate balance between the extremes of pretty-printing is to perform only a simple form of syntax highlighting, such as that provided by the Unix `lgrind` tool. Moreover, systems should provide an option to turn off pretty-printing at the request of the user. In their influential book on programming style, Kernighan and Plauger [18] noted that "if code is clear and simple to begin with, formatting details are of secondary importance".

2.3.4 Indexing and cross-referencing

As with pretty-printing, the level of indexing and cross-referencing support varies greatly across literate programming systems. However, there is unanimous agreement that these features are essential, as they can greatly improve the readability and comprehensibility of a literate program. For example, a maintenance programmer making a change to a single code chunk can determine what other chunks may be affected by that change.

Cross-references attempt to capture the *relationships* between chunks. Some of these are defined by the hierarchical structure of code chunk nesting. As shown in Figure 2.3, CWEB puts the section numbers of sub-chunks alongside the name of those chunks (see "<Scan file 15>"), and also lists the parent of a chunk in plain text (see "This code is used in section 5"). Other systems such as noweb use page number references, usually because they do not organise woven

documents into a flat hierarchy of sections in the same manner as `WEB` and `CWEB`. These systems rely on the typesetting language to provide a table of contents, while systems such as `WEB` can provide one themselves.

Because code chunk nesting is an abstraction of literate programming, all systems are capable of providing references such as those described above. However, other types of relationships based on the text *within* code chunks can only be provided with language dependent knowledge. For example, `WEB` and `CWEB` are capable of comprehensive automatic indexing of identifier definition and use. On the other hand, language independent systems must rely on manual intervention, although can use a simple pattern-matching heuristic to find usage of an identifier once it has been defined [32].

These days, paper does not exhibit the accessibility required of software documentation. It is now common for literate programming systems to include support for hypertext browsing, usually via HTML or the Portable Document Format (PDF). All references between chunks and sections can then be implemented as hypertext links, and modern web browsers can provide navigational support such as browsing history and search features.

2.3.5 Processing intelligence

One of the problems caused by literate programming is the extra level of processing before code can be compiled. Configuration management tools such as `make` use dependencies based on the last modification time of files. For this reason, many literate programming systems will not tangle code unless it has changed, to avoid unnecessary compilation by `make`. However, most systems do not support version control, instead relying on external tools such as the Revision Control System (RCS).

As a consequence of the re-ordering of chunks, line numbers in the tangled source do not correspond to line numbers in the literate program. This can make it very difficult to debug literate code when the compiler supplies only tangled line references. To solve this problem, many systems include support for "`#line`" directives that will force supporting compilers to refer directly to line numbers in the literate source. Without this feature, it is tempting to make modifications directly to the tangled code, which nullifies the property of verisimilitude.

The process shown in Figure 2.1 seems to indicate that literate programs can produce only a single tangled output. In reality, this is a limitation only of earlier systems such as `WEB`. Most systems are now capable of tangling multiple files, creating a *forest* of code hierarchies when considering the graph in Figure 2.2(b). The tangled output destinations are usually specified either on the command line or within the literate program itself. Some systems will automatically tangle all code chunks that have no defined parent. Regardless of the mechanism, multiple tangling is essential these days, particularly with systems such as Java that can include a large number of small class files. Multiple tangling can also be used to include subsidiary files within a single literate source, such as test scripts or a Unix `Makefile`.

2.4 Literate programming systems

In this section, we briefly describe the major literate programming systems, focusing on their key characteristics and differences. Table 2.4 correlates the features discussed in the previous section with those supported by each system. No commercial systems exist, and with the exception of `WEB` and CLiP, all of those listed are still actively supported.

2.4.1 The `WEB` family

As previously mentioned, `WEB` was developed by Knuth in 1983 as a proof of the ideas behind literate programming [19]. It is highly coupled to Pascal, and includes comprehensive pretty-printing and indexing support. It is difficult to customise its woven documentation, and tangled source code is partially obfuscated (hence the name "tangling"). Knuth did this to encourage

System	Languages	Formatters	Macros	Pretty	Indexing	Ref
WEB	Pascal	TEX	✔	✔	✔	[19]
CWEB	C, C++	TEX, LATEX	❍	✔	✔	[22]
noweb	Any	LATEX, HTML, troff	✘	❍	❍	[32]
nuweb	Any	LATEX, HTML	✘	✘	✘	[6]
FunnelWeb	Any	LATEX, HTML, FunnelWeb	✔	✘	✘	[47]
CLiP	Any	Any	✘	✘	✘	[45]
Interscript	Any	LATEX, HTML, Interscript	❍	❍	❍	[39]

Table 2.1: A comparison of the major systems. ❍ indicates partial feature support.

programmers to use only the literate program when reading and modifying code, although it complicates interactive debugging. WEB's biggest limitations include a complex syntax and the inability to produce more than one tangled output.

The dependence on Pascal resulted in many similar systems being created for other languages, especially in the late 1980s. These are collectively known as the "WEB family", because they share the same philosophy as WEB, and in most cases the same limitations. Members of the family include systems for Fortran [1], Scheme [23], APL [25] and Matlab [30]. Others are documented in the Literate Programming FAQ [44]. To reduce the time needed to create an alternative WEB, Ramsey invented the "Spider" system [31]. Given a description of the grammar of a language, Spider generates a variant of WEB specific to that language, freeing the programmer from the arduous implementation details of pretty-printing and cross-referencing.

The most well known and widely used WEB variant is CWEB[1], which continues to be developed by Knuth and Levy [22]. CWEB works with C and C++, and does not support parameterised macros due to their provision in the C pre-processor. CWEB's major improvements over WEB include multiple code outputs, #line directives for debugging, and more recently, hypertext support in PDF documents.

As an historical note, a system called FWEB was developed by Krommes [24] in response to CWEB, with an even more complicated command set and support for multiple languages (C, C++, Fortran and Ratfor). Production is now frozen, but it is relevant as the first step towards language independence, as it not only supported multiple languages, but allowed the *current* language in use to change with each section.

2.4.2 noweb and nuweb

noweb was developed in 1989 by Ramsey [32] to overcome the major limitations of the WEB family: overzealous pretty-printing, lack of support for LATEX, singular output, and a complex syntax that is hard to learn and master. Its major benefit is simplicity, with 5 control sequences compared to WEB's 30.. This, along with its language independence, has helped make it the most popular and widely used literate programming system today. The example in Figure 2.2(a) shows that noweb's commands do not dominate the actual documentation and program code, allowing the programmer to focus more on the methodology rather than the tool.

Although language independent, noweb has an extensible pipeline architecture similar to that of Unix shells. This means that *filters* can be created to perform pretty-printing and indexing without requiring that noweb itself be re-compiled. The programmer can use whatever combination of filters they desire. One popular tool for noweb is Pretzel [15], which creates a pretty-printing filter based on the formal description of a language (similar to Spider). noweb integrates well with make, has support for #line directives, and can tangle multiple files.

One problem with noweb is a lack of speed and portability caused by the pipeline architecture. This inspired Briggs to develop nuweb [6], a very similar but much faster system. nuweb is

[1]This should not be confused with cweb, an earlier system by Thimbleby [42] that supported C and troff but is no longer developed.

contained in a single monolithic C program designed for maximum portability. However, it provides no support for pretty-printing or automatic indexing. With the increased execution speed of modern desktop computers, and a Windows port of noweb now available, nuweb's benefits have been eroded somewhat. In addition, noweb is able to parse nuweb files.

2.4.3 FunnelWeb

Also motivated by WEB's many limitations, Williams began work on the FunnelWeb system in 1986 and released it to the public in 1992 [47]. It is noweb's main competitor among language independent systems, but suffers from a more complex WEB-like syntax and extensive use of macros for all functionality (chunks are actually called "macros", and there is little distinction between these and macros used for other purposes). Like nuweb, it is implemented in highly portable and efficient C code, but can not be extended without re-programming the system itself.

A major distinguishing feature of FunnelWeb is its attempt to achieve *typesetter independence*. It provides an abstract typesetting language, allowing the user to mark-up text independent of LaTeX and HTML. The system can then convert between these representations as necessary. Unfortunately, the language provided is too simplistic for many purposes, and does not include support for images or other multimedia. The result is that FunnelWeb users will often revert to directly entering LaTeX or HTML commands in the literate source. However, the *idea* is a powerful one and will be revisited later in this report.

2.4.4 CLiP

CLiP (Code from Literate Program) is a different type of literate programming system, developed in 1992 by van Ammers and Kramer [45]. It takes the typesetter independence concept of FunnelWeb a step further, and allows the programmer to write documentation and code in *any system* of their choosing. This can include WYSIWYG editors, provided they are capable of saving output in text-only form. CLiP provides a language independent tangler for processing such text, which searches for special directives in program comments that identify chunks of code. However, CLiP has no weaving process, and if the programmer wants their code pretty-printed, cross-referenced or indexed, they must do it themselves.

This extreme style of literate programming has never been popular, because the role of weaving is given to the programmer, who already has a big enough mental load in the task of programming literately. Although the CLiP system itself is typesetter independent, the programmer must actually have *increased* knowledge of typesetting compared with the more traditional systems, because CLiP provides no assistance. This technique is only really useful if the programmer demands, and is willing to specify, *exact* control over the format of woven documentation.

2.4.5 Interscript

Interscript [39] is a relatively new language independent system that includes its own typesetting language, as well as extensibility via an object-oriented design with abstract classes. For example, tangling and weaving processors for different languages or typesetting systems can be plugged in with little effort. It also includes an embedded scripting language, Python, which advanced programmers can use to control and possibly generate output (providing a form of *run-time extensibility*). Interscript control commands are actually calls to embedded Python modules.

Although powerful, Interscript is at this stage too complex and underdeveloped to be a serious contender to the more established systems such as noweb and FunnelWeb.

2.5 Uses and examples

There are many published examples of literate programs. One of Knuth's original claims was that the literate programming methodology would work regardless of the size of the program,

because it allows step-wise refinement of the problem solution. As evidence, he has published the full source for TEX [20] and METAFONT [21] as literate books written in the WEB system. Fraser and Hanson showed that the methodology is not limited to Knuth alone by publishing a large retargetable C compiler written in noweb [13].

In terms of smaller examples, a series of nine articles appeared in the *Communications of the ACM* from 1987 to 1990 [10]. The intention was to publish small literate programs submitted by various authors, and have them reviewed independently as works of literature. The column ceased because of a lack of agreement on the literate programming methodology, and a lack of inter-operability between tools. In his final assessment, the moderator Van Wyk stated: "A fair conclusion from my mail would be that one must write one's own system before one can write a literate program, and that makes me wonder how widespread literate programming is or will ever become" [48]. We will return to this point in the following chapter, as it highlights a major limitation with literate programming: a lack of *real-world usage* by "ordinary" programmers and teams who have not themselves designed the tools.

In terms of academic use, literate programs are good candidates for *teaching* programming, because they allow for clear explanation of the intent and purpose of code. They can be used to guide students through the process of learning a new language by explaining the major constructs and features, with interleaved code chunks as examples.

There have been several experiments actually requiring students to *use* the literate programming methodology. The justification, as noted by Soloway [40], is that learning to program involves not only learning to build computer solutions, but also to *construct explanations*. Thimbleby [42] reported that in student projects written as literate programs, the quality of documentation was as good as the code, and the two were more clearly integrated and homogeneous. Shum and Cook [37] compared submissions from sixteen students of varying skill levels, in two assignments coded with and without a literate programming system. Results showed that the *information content* of those submissions was much improved; they included more description of the purpose of documented code than was evident in the more traditional programs. Bossomaier and Johnson [5] found similar results in a second-year programming project that required submission in WEB format. They also noted that literate programming is a good teaching aid at intermediate level, because it encourages practical experience in a mark-up language such as LATEX.

There have been many attempts to combine the ideas of literate programming with formal modelling and specification [12, 17, 36, for example]. The two ideas fit naturally together, as when writing a program, one may want to justify its implementation, verify how it conforms to the design, or explain a formal notation or specification in psychologically correct order. Typically, TEX is used as the formatter for such purposes due to its complex support for mathematics.

2.6 Alternative documentation techniques

In this section, we briefly compare and contrast some of the major alternatives to literate programming, which can be classified into either *single* or *multiple* source methods.

2.6.1 Single source methods

Single source methods share the principle of verisimilitude with literate programming, because they include documentation and code together in the same file. We have already described how traditional source code commenting, although benefiting from this property, is unsuitable for high-level or lengthy explanations. Moreover, the use of literate programming can reduce the need for internal comments, because named code chunks (such as `PrintHiWorld` in Figure 2.2(a)) often describe their content in sufficient detail. In other words, chunk nesting can be viewed as a form of *pseudo-code*, which states the main purpose of a chunk without requiring all details to be specified until necessary.

Interface documentation techniques such as Javadoc [14] are *not* literate programming systems. They do not support the concept of *psychological arrangement* as described in Section 2.1,

and are targeted only at certain users who wish to call but not modify program code. They are also limited to the abstractions supported by the underlying programming language, such as classes and methods. However, the popularisation of Javadoc has been useful in increasing awareness among programmers on the need to document code for enhanced re-usability.

There are some "lightweight" approaches to literate programming which, like Javadoc, retain the ordering of source files [38, 43, for example]. However, they are able to extract and typeset more complex documentation from within program comments. Although these techniques exhibit verisimilitude, and are capable of producing pretty-printed and cross-referenced, they are again limited by a lack of support for psychological arrangement (and are therefore not true literate programming tools).

Because literate programming seems to hinge on this concept, it is worth further justification. Most modern programming languages are quite flexible in allowing methods or functions to be arranged in any order. Although a highly skilled programmer can use such re-ordering within a single file, it can not be done *between* files in a multi-file system. Furthermore, the chunk abstraction of literate programming allows arbitrary divisions at any point in program text. This can be simulated to some extent by creating a new function or method, but such decisions should be the result of *design choices* only [42]. There is also a need to explain program understanding in *threads* that represent the chronological order in which code is written, and this is rarely related to the hierarchy of a design [29]. Programmers who document solutions after implementation often miss pointing out the understanding that caused them to program in a certain order, when this is usually the best way to explain the code [19, 42].

2.6.2 Multiple source methods

Multiple source methods differ in that the documentation and code are maintained in separate files. Traditional external documentation is the most obvious example. Although such documents can exhibit psychological arrangement, they do not support verisimilitude. For example, many documenters will manually copy source code into a typesetting system or WYSIWYG editor if they wish to explain it, or perhaps refer to "lines x–y" of a program file. As soon as the code changes, the two documents become inconsistent.

The alternative approach is to have an automatic tool manage quite strictly the relationships between documentation and code. These usually insert special directives inside source code comments that identify chunks and links with documentation. Such a system can simulate verisimilitude to a certain extent, provided that the documentation and code files are not changed outside of a controlling tool. One such system is "elucidative programming", a recent development by Nørmark [26]. Although capable of producing high quality documentation linked in with code, it does not provide the same *incentives* to document program understanding, because documentation and code are written in separate windows. In our experience, without true verisimilitude this typically results in a lot of time spent coding before any documentation actually gets written. However, such systems do have the advantage that there is no specific tangling process, because source code can be compiled directly.

2.7 Summary

In this chapter, we have presented the fundamental concepts and ideas behind literate programming, as well as the major supporting systems and their features. We have also justified the approach against alternative techniques, none of which can provide the same level of support for psychological arrangement and verisimilitude. When combined, these concepts provide powerful incentives for programmers to document their understanding, which can facilitate greater re-use and less costly maintenance.

Our description to this point has focused on what literate programming is currently capable of. In the next chapter, we instead focus on its major *problems and limitations*, as a basis for further work on enhancing it to support the demands of modern programming.

Chapter 3

PROBLEMS AND LIMITATIONS

> The thing that will prevent literate programming from becoming a mainstream method is that it requires thought and discipline. The mainstream is established by people who want fast results while using roughly the same methods that everyone else seems to be using, and literate programming is never going to have that kind of appeal. This doesn't take away from its usefulness as an approach.
>
> *Patrick T. J. McPhee [44]*

Despite the many potential benefits of literate programming for producing high quality programs and documentation, it is not a mainstream methodology. In this chapter, we present a critical analysis of literate programming to answer the first goal of our research: why is literate programming not widely applied? We clearly distinguish the problems and limitations identified here from the background of the previous chapter, because they indicate a *set of requirements* for future work in the area.

3.1 Issues of philosophy

In this section, we briefly look at some higher level issues that are not necessarily problems with the methodology, but problems with the applications of, and attitudes towards it.

3.1.1 Programs as works of literature

One of Knuth's original goals was to encourage programmers to want to publish their work. He imagined that one day, our libraries would be full of beautifully typeset literate programs, as well as books of the more traditional kind. His publication of TEX in `WEB` form is one example [21]. He thought that we should publish our solutions to stand as the "quintessential definition" for an addressed problem [19].

One definition of literature is: "writings having excellence of form and expression and expressing ideas of permanent or universal interest". While it is true that literate programming should promote clear exposition and style in documentation, "permanent or universal interest" does not really apply to the needs of real-world programmers. First, publication on paper does not suit programs that are constantly evolving, which is the case for most software and especially modern business applications. Second, if we all published our solutions, we would have a situation something like the World Wide Web (WWW) where it is difficult to find quality material amongst vast spaces of information. Third, most commercial software is not intended to be re-used or read universally, but only by future maintenance users within an organisation.

Most of us do not want to spend a pleasant evening in a comfy chair reading a good program [3]. Nor do we want to publish our work for such grandiose purposes as envisaged by Knuth. For real-world programmers, literate programming should be about encouraging the *documentation*

of understanding and developing empathy with future users of their software. We should write enough quality documentation for this purpose, but no more, as the goal is still primarily to produce good *software*. Towards a more pragmatic view, it might perhaps be time for the name to change to something more like "explicative" or "expositive" programming.

3.1.2 Lack of quantitative evidence

As mentioned in Section 2.1, most evidence that literate programming can reduce debugging time and long term maintenance costs is based only on *observations* by individual programmers, who have often themselves designed the tools. The results from experiments with students mentioned in Section 2.5 can not be extrapolated to real-world programming, because students are motivated by different factors and in particular a desire to do what it takes to receive good marks.

Possibly the only published real-world evidence is by Ramsey and Marceau [33], who used `WEB` on a three-year team project under typical management demands. They observed that programmers found it easy to extend and modify the work of others and that new programmers could learn about the system much more quickly in order to make a change. There were also no quality complaints or requests from project managers to re-write code, as was common in previous projects where programs would often deviate from the original specification.

Such evidence would seem to suggest that it is possible for literate programming to be put to good use in mainstream software engineering. However, there is a distinct need for *quantitative evidence* that can prove the methodology reduces overall costs. Unfortunately, this is something of a paradox, because most real-world managers will be reluctant to try the methodology until it has been proven to work; yet it can not be proven until real-world managers are willing to try it.

3.1.3 The attitude towards documentation

Literate programming requires a major change in attitude among programmers. As discussed in Section 1.1, documentation is usually considered secondary to getting functional code, and there is a common view that it is of little actual use. This is largely because existing techniques result in poor documentation that force maintenance programmers to simply read the code. However, as mentioned in Section 2.1, the literate programming methodology can actually enhance the quality of program code while at the same time producing quality documentation.

Quantitative evidence and more widespread use of literate programming may help change this attitude, as has happened to some extent with the popularisation of Javadoc. However, the biggest problem is a lack of early education on the benefits of literate programming. If it is to succeed, the methodology must be taught early in software engineering courses and programmers must be encouraged to change their attitude towards documentation. This suggests the need for appropriate tools and guidelines to reduce the difficulty of programming literately, which we will discuss in Section 3.3.

3.2 Fixed system models

In the previous chapter, we described the principles behind literate programming, and the characteristics of the major systems. Unfortunately, all of those systems support a very similar *fixed representation model* based on Knuth's original ideas. This model limits their applicability to real-world programming, because it restricts the input and output methods that are available.

3.2.1 Restricted input methods

Lack of inter-operability

Input methods refer to the way in which a literate program is specified and supplied to a literate programming system. One of the biggest problems here is a lack of inter-operability between

systems, because they each have a different syntax and command set. For example, `noweb` uses angled brackets to define and nest code chunks, and the "@" symbol to define documentation chunks (as was shown in Figure 2.2(a)). Other systems use different commands, even though the essential idea is the same: to (a) somehow *distinguish between* documentation and code chunks, and (b) allow the *nesting* of code chunks for the purpose of psychological arrangement.

The implications is that if one wants to use a different system, then one must learn a different (and potentially complicated) syntax. Furthermore, most systems can not read input files in the formats of other systems. This makes it hard to work with literate programs written by others, which may be particularly desirable in team environments or when organisations merge.

The ideal would be to agree on a *standardised syntax* for literate program specification. This would allow systems to focus more on the *processing* of literate programs, and would also allow programs to be written once but then given to whatever tool has a desired feature. Furthermore, with a standard and well defined syntax authors could concentrate on their actual task, which is to write high quality documentation and code. This is the content that exists *between* the syntactical mark-up and should be the focus of literate programming.

This idea can also be extended to the typesetter-dependent mark-up that occurs within documentation chunks. As described in Section 2.4, both FunnelWeb and Interscript provide their own custom mark-up language which does not tie the literate programmer to a certain output form until processing time. However, such a language needs to be standardised if it is actually going to be used by a wide variety of people. For maximum flexibility, it also needs to encourage the use of *logical* rather than *physical* mark-up. One promising candidate is the DocBook standard [46].

Monolithic input files

The process shown in Figure 2.1 indicates that literate programs are written in a *single source document*, such as `foo.web`, and then processed by weaving and tangling into various outputs. This is a big limitation of the existing model, because it demands that all code and documentation must be contained within a *monolithic* input file. While this may be feasible for small programs, it does not scale well to larger systems or those that may be edited by teams. Furthermore, it does not allow chunks to be *re-used* from external files. This is interesting because one of the benefits of literate programming is that it encourages software re-use via better documentation, and yet literate programs themselves can not be re-used.

Note that in some systems, it may be possible to *include* input files using a special command of the system itself, or a directive of the typesetting language (such as LaTeX's `\include`). This can allow a literate source to be split into multiple files for the purpose of scalability—for example, putting each chapter of a literate document into a separate file. However, this is not the type of re-use we are referring to, because the multiple files are still combined into one for the purpose of processing. The desired type is where *arbitrary chunks* from other files can be cross-referenced or re-used, without actually pulling the full verbatim text of the files they are defined in into the input stream.

This limitation can be further understood by considering a likely scenario in modern software engineering. Imagine that a literate program has been written for a reasonably large implementation. It may be desirable to also write an "overview" document for maintenance users that guides them through only key parts of the code, or perhaps a "usage" document that describes only the interface of the system. Existing literate programming systems are not capable of supporting such multiple *themes*, unless the code is re-typed or copied into the extra documents. This is highly undesirable in terms of verisimilitude. Similar scenarios may arise when the same code requires explanation by multiple authors, potentially in different psychological orders. It may also be desirable, in terms of the software life cycle, to maintain analysis and design documents separately from the literate program, but cross-link them together to maintain consistency or verify conformance to a specification.

Inflexible documentation chunks

As was shown in Figure 2.2(b), literate programs contain two main views: the hierarchical structure of nested code chunks, and the linear structure of documentation. The nesting of code chunks is flexible and supports the property of psychological arrangement. However, documentation chunks do not exhibit the same power. In all literate programming systems, it is implied that documentation describes the *immediately following* code chunk, and that there is a linear series of such two-part sections.

The problem with this linear structure is that it becomes difficult to write "overview" level documentation chunks. For example, one might want to describe the purpose and intent of the three sections shown in Figure 2.3 at a "high level", before going into the implementation details. It is possible to do this by specifying a preceding documentation chunk with no code chunk (which might be section 7 in the example). However, because the woven documentation is a linear structure, this does not capture the explicit relationship that the "high-level" chunk *describes* the other three sections. In other words, explicit *documentation nesting* is desired in the input format.

Supporting a more flexible view of documentation chunks could also provide the basis for documentation re-use. For example, one could define a common explanation for a certain design pattern, name it, and then re-use that explanation wherever it is implemented in the code. Furthermore, cross-references between documentation chunks currently rely on the features of the embedded typesetting system. It may be desirable to support cross-referencing as an abstraction of the literate programming model itself; for example, "see documentation chunk x for an alternative explanation of this code".

Lack of input format extensibility

Current literate programming systems are fixed in their input formats and do not allow the programmer to adapt them in any way. For example, it might be useful to allow chunks to have an *author* or *version number* associated with them, to support team use or revision control at a chunk level. Such features may want to be added on demand, rather than fixed into the literate programming system itself.

3.2.2 Restricted output methods

Fixed weaving

The outputs of a literate programming system are the result of the tangling and weaving processes shown in Figure 2.1. We have already described how multiple tangling is now supported by most systems, thus allowing multiple program files to be output from the same literate source. However, current systems do not support *multiple weaving* as part of their model. For example, a modern programmer may wish to output both a one-page overview of their literate program as well as a full description. This also requires a more flexible means for inputting documentation chunks into nested hierarchies, as described in the previous section.

Lack of differentiated outputs

The structure of a literate program is determined fully by the author based on the linear ordering of documentation, and nesting of code chunks. Tangling and weaving processes in current systems respect this order and output the appropriate documentation and code files in their entirety.

However, users of a literate program may have very different needs. For example, maintenance users may want to view the full woven document or may want only the subset of it that is relevant to their task. Actual users of the code may only want to see the documentation for the program interface (such as functions and their arguments). Managers may want to request statistics such as the ratio of documentation to code or some other measure of software quality. Testers may want to extract only certain code chunks that have not already been marked as "tested".

All of this suggests a need for more ad-hoc querying of literate programs, allowing arbitrary tangling and weaving from and between any chunks. It also requires that the varying needs of future users have been thought of by the programmer and included in the documentation.

3.3 Authoring problems

Literate programming is more demanding than ordinary programming, because it suggests the author must be good at communicating with both humans and computers at the same time. In this section, we briefly discuss two major problems that contribute to its limited widespread use: a lack of tool support, and a lack of guidelines to help with the authoring process.

3.3.1 Lack of tool support

Literate programming suffers from the problem that it requires users to apply four languages at one time: the language of a literate programming system, a typesetting language, a programming language, and English (usually). This can actually be worse when programming a multi-lingual software system, or documenting for the multiple purposes and needs of future users. Tools therefore have a significant role to play in reducing the mental load of the programmer.

Some of the conveniences tools can provide include:

- hiding the syntax of the literate programming system behind an abstraction-oriented user interface.
- hiding the syntax of the typesetting language via WYSIWYG views (including the display of images and other media).
- *on-line* views of the normal conveniences given only to a reader of a literate program, such as tables of contents, cross-references and indexes.
- *interactive* views of tangled program files that allow editing both in the code (for debugging purposes) as well as the literate program.
- automation for the tangling and weaving processes.
- integration with external tools such as version control software for team use.
- the normal conveniences of a program source editor, such as syntax highlighting and bracket matching.

Although there has been a fair amount of research into design criteria and prototypes for such tools [4, 7, 9, 35, for example], there is a distinct lack of *actual* fully implemented, well supported, and widely applicable tools. For example, there are quite a few `emacs` modes for literate programming systems documented in the FAQ [44]. These are only useful for a small minority, and `emacs` can not be considered a mainstream program editor. Full systems are needed that are oriented towards the literate programming methodology, because of its unique combination of both typeset documentation and program code. These systems need to work on modern platforms (such as Microsoft Windows), and need to integrate with modern development environments or source code editors. Until such systems exist, literate programming will never have widespread appeal because it will be passed off as a methodology for academics, experts, or Unix gurus.

It should be noted that there is one promising recent development called “Leo”, a graphical editing environment that runs on Windows and Macintosh [34]. Leo can read `noweb` and `CWEB` files, as well as its own proprietary format. It can also syntax highlight certain programming languages and provide automatic support for tangling and weaving. It does have many limitations, however, such as only displaying the content of one chunk at a time. This gives *focus* but little *context*. However, the system is promising in that it is an attempt at a more widely applicable and easy to use tool, which can help reduce the mental load of the user.

3.3.2 Lack of guidelines

Authoring of literate programs is not easy; in addition to the need to understand how to design programs and communicate with humans, one must also design the literate program itself. This requires splitting code into appropriately sized chunks for best explanation to humans. Often, novice users will break up their code based only on the abstractions of the underlying programming language, and this is not always the best way to document understanding (as discussed in Section 2.6.1). Hamer [16] points out that they may also abandon good program design in favour of "wholesale chunk usage".

The problem is a distinct lack of guidelines on how to write a good literate program, or indeed agreement on what constitutes a good literate program. A "style guide" for literate programming is needed that can help authors, particularly novices, avoid common pitfalls and deal with everyday tasks. Such a guide might also include "patterns for literate programming", because it is likely that good literate programs share common features in the same way that good designs do.

Chapter 4

TOWARDS MODERN LITERATE PROGRAMMING

In summary, we have identified several key requirements for mainstream adoption of literate programming:

- there needs to be improved education on the methodology, and a change in attitude among programmers towards developing empathy with future users of their software.
- there needs to be greater inter-operability and standardisation among literate programming systems and their input formats.
- the model supported by literate programming systems needs to be less restrictive, including support for multiple and more flexible inputs, multiple re-use, and multiple outputs for the varying needs and purposes of different software users.
- there needs to be powerful, widely applicable, and highly usable tools to help reduce the mental load on the literate programmer.
- clear guidelines are needed to help literate programmers with applying the methodology.

These requirements provide a solid basis for further work in the area.

BIBLIOGRAPHY

[1] A. Avenarius and S. Oppermann. FWEB: A literate programming system for Fortran8x. *ACM SIGPLAN Notices*, 25(1):52–58, January 1990.

[2] R. Baecker. Enhancing program readability and comprehensibility with tools for program visualization. In *Proceedings of the Tenth International Conference on Software Engineering*, pages 11–15, Raffles City, Singapore, April 1987.

[3] J. Bentley and D. Knuth. Programming pearls: Literate programming. *Communications of the ACM*, 29(5):364–369, May 1986.

[4] J. M. Bishop and K. M. Gregson. Literate programming and the LIPED environment. *Structured Programming*, 13(1):23–34, 1992.

[5] T. Bossomaier and C. Johnson. The role of literate computing in undergraduate curricula. Technical Report TR-CS-94-01, Department of Computer Science, Australian National University, ACT, Australia, 1994.

[6] P. Briggs. Nuweb version 0.92: A simple literate programming tool. `http://nuweb.sourceforge.net/`, Department of Computer Science, Rice University, Houston, TX, USA, 1993.

[7] M. E. Brown and B. Childs. An interactive environment for literate programming. *Structured Programming*, 11(1):11–25, 1990.

[8] B. Childs. Literate programming, a practitioner's view. *Tugboat*, 13(3):261–268, October 1992.

[9] A. Cockburn and N. Churcher. Towards literate tools for novice programmers. In *ACSE'97: Second Australasian Computer Science Education Conference*, pages 107–116, Melbourne, Australia, 2–4 July 1997.

[10] P. J. Denning. Announcing literate programming. *Communications of the ACM*, 30(7):593, July 1987.

[11] D. Diamond. For a friend assigned to a maintenance group. *Datamation*, page 134, June 1976.

[12] M. Ferguson. A "formal" promela model for CDMA-RLP based on IS707-2. Available from `http://www.cosc.canterbury.ac.nz/~mike/`, December 2000.

[13] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, Redwood City, CA, USA, 1995.

[14] L. Friendly. The design of distributed hyperlinked programming documentation. In *IWHD'95: International Workshop on Hypermedia Design*, Montpellier, France, 1–2 June 1995.

[15] F. Gartner. The Pretzel homepage. `http://www.informatik.tu-darmstadt.de/BS/Gaertner/pretzel/`, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany, 1999.

[16] J. Hamer. Literate programming: A software engineering perspective. In *Proceedings of the Software Education Conference (SRIG-ET '94)*, pages 282–288, University of Otago, Dunedin, New Zealand, 22–25 November 1994. IEEE Computer Society Press.

[17] C. W. Johnson. Literate specifications. *IEEE Software Engineering Journal*, 11(4):225–237, July 1996.

[18] B. W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, New York, 1974.

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

[20] D. E. Knuth. METAFONT*: The Program*. Addison-Wesley, Reading, MA, USA, 1986.

[21] D. E. Knuth. *TEX: The Program*. Addison-Wesley, Reading, MA, USA, 1986.

[22] D. E. Knuth and S. Levy. The CWEB system of structured documentation. Available from `http://www-cs-faculty.stanford.edu/~knuth/cweb.html`, January 2001.

[23] D. Kobler and D. Hernandez. StoL—literate programming in SCHEME. Technical Report FKI-157-91, Technical University of München, München, Germany, 1991.

[24] J. A. Krommes. FWEB: A WEB system of structured documentation for multiple languages. Available from `http://w3.pppl.gov/~krommes/fweb.html`, 1998.

[25] P. Naeve, B. Strohmeier, and P. Wolf. APL programming without tears—it is time for a change. *APL Quote Quad*, 24(1):185–189, August 1993.

[26] K. Nørmark. Requirements for an elucidative programming environment. In *Proceedings of The International Workshop on Program Comprehension (IWPC'2000)*, Limerick, Ireland, 10–11 June 2000.

[27] P. W. Oman and C. R. Cook. Typographic style is more than cosmetic. *Communications of the ACM*, 33(5):506–520, May 1990.

[28] G. Parikh and N. Zvegintov. *Tutorial on Software Maintenance*. IEEE / Computer Society Press, Silver Spring, MD, USA, 1983.

[29] D. Parnas and P. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, February 1986.

[30] M. Potse. MWEB user manual. Available from `ftp://ctan.tug.org/tex-archive/matlabweb/`, May 2000.

[31] N. Ramsey. The spidery WEB system of structured documentation. Technical Report CS-TR-226-89, Department of Computer Science, Princeton University, Princeton, NJ, USA, August 1989.

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

[33] N. Ramsey and C. Marceau. Literate programming on a team project. *Software—Practice & Experience*, 21(7):677–683, July 1991.

[34] E. K. Ream. Leo user's manual. `http://personalpages.tds.net/~edream/`, July 2001.

[35] T. Reenskaug and A. L. Skaar. An environment for literate smalltalk programming. *ACM SIGPLAN Notices*, 24(10):337–345, October 1989.

[36] T. C. Ruys and E. Brinksma. Experience with literate programming in the modelling and validation of systems. In *TACAS'98: Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 393–408, Lisbon, Portugal, 31 March–3 April 1998.

[37] S. Shum and C. Cook. Using literate programming to teach good programming practices. In *SIGCSE'94: Twenty-fifth Technical Symposium on Computer Science Education*, pages 66–70, Phoenix, AZ, USA, 10–11 March 1994.

[38] V. Simonis. ProgDoc—a program documentation system. Available from `http://www.progdoc.org/`, May 2001.

[39] J. Skaller. Interscript homepage. `http://interscript.sourceforge.net/`, June 2001.

[40] E. Soloway. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, September 1986.

[41] T. A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5): 494–497, 1984.

[42] H. Thimbleby. Experiences of 'literate programming' using cweb (a variant of Knuth's `WEB`. *The Computer Journal*, 29(3):201–211, June 1986.

[43] H. Thimbleby. Explaining programs reliably. Working paper, `http://www.cs.mdx.ac.uk/harold/srf/javatex.pdf`, School of Computer Science, Middlesex University, London, England, UK, 2000.

[44] D. B. Thompson. The literate programming FAQ. Available from `http://shelob.ce.ttu.edu/daves/lpfaq/faq.html`, March 2000.

[45] E. W. van Ammers and M. R. Kramer. The CLiP style of literate programming. Available from `ftp://sun01.info.wau.nl/clip`, Computer Science Department, Wageningen Agricultural University, Wageningen, The Netherlands, February 1993.

[46] N. Walsh and L. Muellner. *DocBook: The Definitive Guide*. O'Reilly, October 1999.

[47] R. Williams. The FunnelWeb literate programming tool. `http://www.ross.net/funnelweb/index.shtml`, University of Adelaide, SA, Australia, 1999.

[48] C. J. V. Wyk. Literate programming: An assessment. *Communications of the ACM*, 33(3): 361–365, March 1990.