Explaining code for publication

Harold Thimbleby*

UCLIC, University College London Interaction Centre, London.

SUMMARY

The computer science literature discusses code and algorithms extensively, but not always reliably. Tool support can help ensure integrity between code and explanation so that published papers are more reliable.

A versatile, light-weight tool to support explaining code for publication is justified, described and compared with alternatives. The tool works with Java, C and similar languages, and provides support for publishing explanations of real code in LaTeX, XML, HTML, *etc.*

KEY WORDS: Documentation, Java, Javadoc, LaTeX, Literate programming, Loom, Publishing code, Warp, XML.

Introduction

A basic view of how science is done is that scientists do research; they keep records in laboratory books; they then submit selected material and reasoning to journals, which brings their key discoveries and ideas to the attention of the rest of the scientific community. With computers, of course, keeping and managing laboratory books and publications can be partly automated: editing, typesetting systems and other authoring tools are widely used.

Despite the possibilities of computer support, the quality of published program code in scientific publications is often low. Numerous algorithms and code extracts published in refereed journals and books contain errors, even to the point of not being compilable as published. Some algorithms are published in pseudo-code, but the informality risks errors being overlooked or new errors being introduced when the pseudo-code is translated to real programming languages.

*Correspondence to: H. Thimbleby, UCLIC, University College London Interaction Centre, 26, Bedford Way, London, WC1H 0AP. Email: `h.thimbleby@ucl.ac.uk`

This paper surveys the relevant issues in scientific dissemination as they apply to computer science, and the role and relevance of tools to support the explanatory process. Part 1 of this paper, then, not only establishes the rational for tool requirements but should also encourage computer scientists to take code explanation tools seriously.

Just as there is a continuum from laboratory notebooks to polished published results, there is a continuum from explaining entire programs, as required for instance for internal documentation, to explaining specific code and algorithms for journal publication. Literate programming and related tools address the laboratory notebook end of code explanation, though they can still be useful for publication. Part 2 reviews such tools.

The main practical contribution of this paper then follows. Part 3 is a discussion of the requirements for publication tools and a comparison of two such tools. A new tool, warp, is introduced and explained, and compared with an existing (but little known) alternative, loom.

The final part of this paper is a complete stand-alone short paper presenting a new variant of insert sort, which used the tool warp to ensure the published code is accurate and reliable. Part 4 is separate from the rest of this paper so that the value of warp can be assessed without reference to the discussion of Parts 1 to 3 interfering — one of the points this paper makes is that publication tools must be transparent to readers who do not have in mind the rationale for using them. (One of the main reasons why literate programming is unsuitable for general publication purposes is that its conventions need explaining to readers.)

The ideas expressed in this paper are not limited to explaining code reliably, though this is the main area of concern: the approach has applications in any areas where publications include formal text or pictures, in fields as diverse as chemistry to theorem proving, and even in papers where the formal text is as simple as just requiring correct arithmetic. Further applications include generating user manuals or diagrams automatically from data defined in programs.

Part 1

Publishing computer science

Science, understood most generally, is concerned with finding explanations of the universe and its phenomena,[2] and to do so it is guided by principles, such as Occam's Razor, and a conviction that reality is fixed. As a matter of fact, we cannot ensure reality conforms to our explanations;[3] but in computer science, at least, we have an advantage. In computer science we design and build the objects we explain, and we can change them — and perhaps we explain them again. We can in principle ensure that our explanations are correct. We can do so by changing the explanations, changing the objects, or by establishing and maintaining relations between objects and explanations so they remain consistent.

There is good and bad science. Good science is distinguished by being reproducible or, as John Ziman puts it, *consensible*:[4] its claims should be expressed clearly and precisely so that others can either give their assent or find well-founded objections. Others should be able to build directly on published work, with a minimum of guesswork or interpretation.

Computer scientists write programs and explain programs — whether to document them for other programmers, to explain them in the computer science literature, or to write manuals, or help for users. As Abelson and the Sussmans put it in their classic *Structure and Interpretation* *of Computer Programs*,[5] "programs must be written for people to read, and only incidentally for machines to execute." Alan Perlis (who wrote the preface for their book) wrote much earlier in 1966, of a "firm conviction that fluency in writing algorithms for one another and reading those written is a fundamental property of a professional ... "[6]

Publishing code reliably means that people can use the code directly and benefit from its correctness; it also means that people can independently check its correctness, efficiency, portability and so on. Informal means of publishing code, particularly using pseudo-code, are inadequate; errors are spread in the literature, work must be unnecessarily duplicated, and when an error is detected one does not know whether this is caused by a fault in the original work, a fault in authoring the paper, a fault in its printing, or a fault in understanding and implementing its ideas. Fortunately, as this paper shows, better approaches are possible and the process can be automated, and hence ensure, to much higher standards, the reliability of published code.

Dunham,[7] writing before the world wide web provided alternatives to conventional journal publication, argued that programs should be published in full: full publication of program code enables reproducible experiments; and the exposure of full publication encourages people to write better programs. Conversely, without publication there can be little or no use of the algorithms, because nobody knows what they are. Furthermore, when algorithms are stated vaguely or in informal language it disguises the difficulty of their real implementation. We agree with his sentiments, but not that programs should necessarily be published in full: part of the contribution of a scientific paper is the selectivity of the author in chosing the key pieces of code to publish. Indeed, a clear fragment of code, as might be presented in a paper, is not necessarily one that is optimal or works nicely in a properly self-contained way under a particular operating system, with error recovery, user interface, on-line help and so on.

Surprisingly the whole enterprise of reliably publishing code in the computer science literature has received scant attention, with perhaps only two notable exceptions. Attention to the issues was focused during the early debates on whether the programming language Algol 60 was appropriate to publish algorithms;[6] and a renaissance of attention arose with literate programming, which was popular over a decade ago. We will look more closely at literate programming later in this paper.

Publishing code

Programming and explaining are activities that involve human intervention, typically using word processors. Since explanations of algorithms or programs often refer to code, parts of the documentation or explanation will be similar if not exactly the same as parts of the program. Thus there are opportunities for computer support to help explain code more reliably.

Programs are usually written in plain ASCII text, but documentation usually has a special form. If a WYSIWYG word processor is used, program code needs editing to fix the font, alignment, point size and so on; if a mark-up language (like XML, HTML or LaTeX) is used, then various program symbols need converting to the mark-up language's conventions so that they can appear properly. For example, the programming language symbol '`&`' has to be edited to '`&`' for HTML or to '`\&`' for LaTeX — and of course this mark-up is no longer valid in programs, so accurate conversion to documentation cannot readily be confirmed with a compiler. Even JavaScript, a programming language *designed* to be embedded in HTML files, does not help: JavaScript cannot be printed as HTML, nor is HTML-formatted JavaScript code valid JavaScript that can be run.

The program Expect[9] can be used to check whether examples generated by running programs remain correct while the programs being explained are modified. Don Libes, the author of Expect, notes that writing about a program forces the author to reexamine implementation decisions: and he rewrote parts of Expect when he realised that parts of his explanation were overly complex for no good reason.[9] There are even examples where "programs for humans" have also been improved by trying to explain them clearly: for example, shop signs explaining discount rules were tested and improved, but greater improvements were made to readability by changing the rules so they could be more easily explained.[10] With proper tool support improvements to the code can be reflected in the documentation without further effort on behalf of the author. However, without tool support, an author might be tempted to avoid improving code because to do so would mean not just improving the code but also revising the documentation further — a tedious process.

For writing journal papers, which typically only use very small fragments of code, the overhead of learning or building tools may seem out of proportion to the advantages. When one starts to write a paper — say, for a conference deadline — the salient goal is usually to submit on schedule: building tools to automate a comparatively small part of the process is an unwelcome diversion. Typically, then, a fragment of program code is cut-and-pasted from a working program into the journal paper (if the code fragment is small, it may even be retyped *in situ*). The code is then edited carefully to conform to the documentation system's requirements (and to the typographical requirements, particularly line length and indentation of the journal or conference proceedings). This is rarely the end of the matter, as the resulting explanation will be read and revised, no doubt repeatedly.

Inevitably some changes will be made, say, to improve the style of the program for the explanation. For example, the documentation might read better if a name is changed, or some comment is added or altered ... this is the thin end of the wedge, and a subtle threat to integrity. The first may be a trivial change, so it perhaps isn't worth going to the trouble of making the corresponding change in the actual program. As time goes by, some changes will not be made consistently and some not at all. Eventually the explanation and original code will diverge to a point where there is no simple way to reconcile the differences.

The papers about Quine are an illustration of this problem. Quine is a system to generate accurate documentation, and the papers were written in Quine itself to guarantee their accuracy. Unfortunately the publishers edited the original paper[11] to make it look better for their journal, but their edits made it incorrect. The errors introduced were sufficient to justify republication,[12] and as the corrected paper noted, if one wants to explain programs reliably, the *entire* process should be automated, not just the author's part!

In comparison, using pseudo-code seems like an attractive option for explaining code, but even that approach is not without problems. Take the Chinese Postman Problem,[13] which is closely related to the Travelling Salesman Problem, and has many applications. Many references to it describe the algorithm in a mixture of English and mathematical steps: for instance, the algorithm sketched in reference 14 cannot be made to work, if at all, without very close reading of the rest of the paper, plus expertise in network flow algorithms. Some publications (e.g., references 15 and 16) describe the algorithm entirely in English prose, so at least there are no unexpected sources of confusion, but it is unhelpful for people who want correct, executable algorithms. Some (e.g., 17) just sketch the main steps; indeed its presentation is, in its own words, "rather informal." Some (e.g., 18) provide a mixture of mathematics and English, and provide code for most of the relevant routines, but the code is difficult to translate into complete and correct algorithms (e.g., because what may look like a simple variable is in fact a non-trivial dynamically bound expression). The definitive review of the state of the art in the field,[19] which has nineteen leading contributors, gives no formal algorithms, and in one place says of certain efficient algorithms, "According to our knowledge these very refined implementations were never materialized into machine-executable codes . . . " One wonders whether they would still consider them "very refined implementations" if they tried running them!

In short, pseudo-code has become the established mode of publication for the Chinese Postman Problem, and even the very extensive literature on it is not an effective starting point for getting a reliable algorithm. The algorithms that have been published can at best be called *quasi-algorithms*, for they clearly fail the test of definiteness required in computing science.[20] Skiena's comprehensive *Algorithm Design Manual*[16] concludes that you have to do it yourself.

Another example is Porter's stemming algorithm,[†] which was originally published using a non-programmable notation,[21] which led to a proliferation of incorrect implementations. As Porter himself says, "Three problems seem to compound: one is a misunderstanding of the meaning of the original algorithm, another is bugs in the encodings, and a third is the almost irresistible urge of programmers to add improvements. [. . .] Researchers frequently pick up faulty versions of the stemmer and report that they have applied 'Porter stemming,' with the result that their experiments are not quite repeatable. Researchers who work on stemming will sometimes give incorrect examples of the behaviour of the Porter stemmer in their published works."[22]

Rather than present clearly explained algorithms, it is easier to say that a program is available (e.g., on the web); yet as Porter says, to extract an algorithmic description from source code of a complete program is hard. The original author saves much effort and work by not preparing the algorithm for clear explanation (and hence publishes faster). But which details of their program are essential? Which are accidental bits imposed, say, by their implementation environment? Indeed, this was the problem with Porter's classic paper: providing complete source code was not an adequate explanation of the algorithm. It takes considerable work to polish an algorithm for clear explanation. If it takes considerable work to polish an algorithm, we need tools to make that work as easy and as reliable as possible.

I myself have published papers in this journal and elsewhere that fail to reach the standards I would now advocate. Taking just two recent publications of my own in this journal: I published an algorithm in an unrunnable pseudo-code,[23] and, in another paper, I used manual cut-and-paste plus reformatting to present algorithms in Pascal.[24] To date my experience (publishing

[†]Stemming is finding canonical forms of words; for instance, programmable→program.

92 algorithm-related papers to date) is probably representative of the standards to which many authors work. In almost every case my approach to preparing papers was relaxed, and it was not required to be otherwise. Referees for ACM, BCS, IEEE and other peer reviewed journals were happy with my sloppy standards. In my case the few exceptions were either written in systems I myself wrote (e.g., Quine[12]) or were papers (e.g., reference 25) written in *Mathematica*, a system that combines word processing and programming, and hence permits the paper and the program to be more-or-less the same thing (*Mathematica* is reviewed below).

Even in a recent paper in *ACM Transactions on Computer Human Interaction*,[26] which presents the entire code relevant to the paper in an appendix, we formatted the code by hand (manually marking it up in LaTeX) to make it conform to the journal's requirements. There is no guarantee that the appendix is correct, other than that we proof read it conscientiously! As an author I made the code available on a web page, which at least avoids the risks of rekeying for readers who wish to recover the code. The journal did not require this.

Comparisons with other fields

Is this relaxed attitude to publishing programs actually a problem?

Because scientists are human and fallible, science has developed a collection of ethical principles, including the requirement that data reported should be authentic and reliable, that it should be obtained in ways that are described fully enough to be replicatable, and that claims made should be clear enough to be testable by others. Different disciplines apply these ideas in different ways: for example, the *Proceedings of the National Academy of Sciences* requires authors to make unique material (specifically including computer programs) available to non-commercial researchers.

In the field of cognitive psychology, John Anderson and Christian Lebiere promote a "no-magic doctrine." Their view is that the dissemination of theories in psychology traditionally counted on a sympathetic audience to understand how the theories should be applied in practice.[27] They go on to cite some of their own past work that exploited this sympathetic culture that accepted loose descriptions of research in publications. But their current programme of research now adheres to the no-magic doctrine. The doctrine is broken down into six tenets, not all of which concern us here, but which for instance require that there should be detailed and precise accounting of data. It is no coincidence that their no-magic doctrine is enabled by the use of computer simulation in psychology: the formal nature of computers allows experimental situations to be precisely specified and shared with the research community. They take advantage of that fact to improve the quality of their published work.

The sciences that have reacted to the problems of reliability of published explanations are older and more established than computer science. Of all fields, mathematics has the longest history of publishing precise reasoning, and its notion of proof lies perhaps closest to our notion of algorithm. As in computer science, where there are some programs that are only described by their results, and others that can be concisely explained in a short paper, mathematics has some proofs that are too long to be understood and others where proofs are short and elegant. Fermat's Last Theorem and the Four Colour Theorem, to say nothing of proofs of computer programs themselves, are examples that highlight the differing opinions on what constitutes a reliable explanation of a proof for mathematicians.[28]

In mathematics, then, and in other areas (such as weather forecasting), there are complex computer systems that in principle cannot be explained concisely but which are, nevertheless, useful contributions to science: we do not need to — and should not — impose the requirements for explaining algorithms reliably on any and every publication. On the other hand, when an algorithm *seems* to be presented as explicit code, every step should have been taken to ensure the explanation is as reliable as it appears. Publication in any discipline ideally exposes to the community an explanation that should be able to be reliably assessed for what it apparently stands for.

Unfortunately, in computer science itself it has become routine to describe programming ideas without publishing, let alone depositing, the relevant code, programs or underlying algorithms for community access.

We must now ask why.

Partly, low publishing standards have become accepted because of the commercial value of programs — it has become inappropriate for commercial reasons to publish code.[29] Partly, because programs are large. (Neither of these issues inhibits biologists from making data available to the research community.) Partly, because it is easier to describe in unqualified terms a program that, perhaps, does not quite work in the general ways implied. Partly, because debugging is practically a permanent state of affairs, and adequate version control is a nightmare — particularly since the relation between code and its explanation in a publication is implicit and not addressed by automatic version control tools.[‡]

It has become routine not to publish accurate code partly because computer science has developed a culture where radical honesty[30] is not valued and not expected. It is actually *extremely* difficult to get programs working correctly, and why bother when the essential idea seems perfectly clear? The existing literature is now training researchers what the minimal levels expected are; it is not helping improve standards, and it is not helping introduce new approaches or new tools to increase the reliability of the future literature.

Partly it is because computer science is in a hurry. In the 1960s, George Forsythe argued that the publication of algorithms constitutes an important part of scholarship, but he warned that the process of writing, polishing, refereeing, editing and publishing can hardly be undertaken in less than two years.[31] Even if we can work a little faster today, we still require the programming notations we use to have a stable life of around five years for the final publication to have much value — but, with few exceptions, the only stable programming languages are the ones that are not widely implemented or widely used!

A great deal of the published literature in computer science is about the commerical applications of systems, software and hardware reviews and such like. Computer science and business are so closely connected that the methods of science are often confused with the methods of business, where disclosure without limitation would be unprofitable. For example, a review of commercial Chinese Postman solutions[32] says that vendors are "tight lipped" about their algorithms. From the commercial point of view, then, there is no interest in and even a resistance to clear exposition of algorithms.

[‡]Les Carr has an interesting experimental system for converting change files into unfolding explanations (which he uses for lectures); see `www.ecs.soton.ac.uk/~lac/annann.html`

Computer science must be unique in the continual triumph of hope over reality. Software is unreliable. We tend to emphasise hope in tomorrow's solutions, as exemplified by the exponential growth of almost every performance figure, yet we forget that every new solution bought represents a failed, obsolete solution that is unfixable because its workings are unknown and hidden: Moore's Law is a business, not a technical law.[33] As commercial software warranties show, customers are made responsible for fixing problems that should never occur, or which should be the manufacturers' responsibility.[34] Undoubtedly, concealing and obscuring program code, regardless of its scientific hazards, is good business: it is more profitable to sell upgrades than admit or fix problems.

As professionals we are so inured to this continual progress that it seems perfectly reasonable to publish code and shortly afterwards say that it does not work because the compilers have changed, or because we have upgraded the operating system, or because the work was on a computer that has now been replaced. Or perhaps we didn't make a backup before we lost it? In the established scientific disciplines any of this sort of behaviour would lead to outrage.[35]

If program code is not made available, how can it be tested? How can it be refereed? How can the community develop the ideas without starting again? How can the discipline advance with confidence? In particular, if the publication itself does not provide an adequate description of the algorithm, where are future workers to find it?

These problems are ironic because the computer itself can provide a stronger test of validity than any that is available to other scientists in their work;[36] furthermore the testing can be automated, for instance using framework testing[37] — so there is no long-term burden on the author, and there is everything to gain.

Positive aspects of computer science publishing

Such a negative list of problems needs balancing with some undoubted successes. Certainly there are some specialist areas, such as the ACM's publication of numerical algorithms where these problems have been recognised and addressed, but the sentiment is by no means universal.

Open source software (distributing program source code openly, allowing the code to be criticised and fixed by others) has resulted in some unusually reliable programs;[38] the internet is encouraging authors to make their programs downloadable; Java applets allow authors to publish documents that *include* working programs (though the source code still may not be available).

In computer science we can make the tools to make the necessary scientific assurances easier to achieve. For example, there is a web site of collected algorithms (`http://www.acm.org/calgo/`) so the source code of some algorithms associated with refereed publications in ACM journals can be down loaded. CALGO specialises rather on mathematical algorithms, and one wonders why algorithms more generally are not made available, for instance for the numerous programs discussed in the *Communications of the ACM* or in *Software—Practice & Experience.*

As noted above, being able to access complete source code, useful as it is, solves a different problem than assuring that the published explanation of code (and examples of runs, if any) is accurate.

Not just programs, but data too

One reason incorrect implementations of the Porter stemming algorithm proliferated is that the original paper on the algorithm[21] did not provide test data. Thus people read the pseudo-code description of the algorithm in the paper, and they implemented what they thought was meant. Neither they nor the people they distributed their implementations to had any way of checking their work against what Porter intended. Porter's current work is now supported with a database of test cases to avoid this problem.

Even when a paper explains an algorithm lucidly, there is a danger that a reader's transcription error will lead to mistakes. In particular, even though a well-defined programming language may be used in an explanation, some readers of the paper may need to translate the program into a language that is accessible to them. If it is plausible that errors will creep in, then there need to be mechanisms to reduce the likelihood that errors of any sort remain undetected. Test criteria and, typically, test data should be made available (or programs that generate test data should be made available).

It is of course possible that the original author of a program has made a mistake in the choice of test data, and thus their claims about their algorithm are faulty. Distributing the test data is one way to help detect this sort of mistake.

Some fields of computer science, notably information retrieval and machine learning, have made substantial test databases available. There are also web sites of XML data, and programs to check XML standards conformance. There are many other examples, but it is by no means standard practice. One wonders why the need for publishing test data (whether for pure science or for profit) is not more widely recognised. For example, Java compilers would be more reliable — and Java programs more portable — if they could be and were routinely validated against standard tests.[39]

The ethics of publication in computing science

When the community relies on unreliable publications, problems arise. In computer science this is at least tedious; in medicine, for instance, lives are obviously put at risk. Medicine has therefore developed a strong sense of appropriate ethical behaviour to be applied throughout the publication process. The medical community has also detected numerous cases of fraud and unethical publishing behaviour. It would be very strange if computer scientists were exempt from the sorts of temptation to which medical researchers succumb. Yet the ACM Code of Ethics and Professional Conduct[40] makes no mention of ethical behaviour directly relevant to maintaining high standards in scientific publication, though "the honest computer professional will not make false or deceptive claims about a system" (Code §1.3). Few computer professionals, however, will be conscious of the connection between this exhortation and their standards of scientific publication.

There are many refereed computing science papers that "envision" work that is clearly not yet working. Some imaginative papers do not even make their envisionment status clear. Such papers make it hard for subsequent work to progress, as referees tend to reject later papers that appear to repeat, test or develop prior work, even if it was envisionment, as if the scientific copyright to the idea was held by prior imagination rather than by actual fact.

Taking conceptual ideas to something that *really* works uncovers important additional details and qualifications that would not have been known but for the concrete effort.[41] Yet to work to these standards takes time, and risks others jumping in with outline papers that take precedence — and then the developed work cannot (under current practice) be published.

There is certainly scope for further ethical research to develop and apply the traditions of reliable science and reliable publication to computer science — for instance to be clear about the dangers of fudging, trimming and cooking published code. See Resnik for a general review[42] — especially if these terms are unfamiliar! However, this paper is not the place to pursue in any greater depth a discussion of the processes and ethics of publishing; instead we turn to pragmatic ideas to help avoid errors in the first place.

Part 2

'Laboratory book' tools for explaining code
Literate programming and related work

Literate programming was introduced by Knuth[41,49,53,54] to combine programs and their documentation to create readable programs. Numerous programs have now been prepared using literate programming: sizable examples include Knuth's own outstanding books (e.g., on graph theory algorithms[55]). In literate programming, there is no duplication of either code or documentation: there is only one shared copy. Code and documentation are interleaved in a file, and as they are adjacent in the same file it is *very* much easier to keep them consistent. Reducing the obstacles for editing both together, and increasing pride in the polished results, has an invigorating effect on programming, as well as on dissemination.

Conventional literate programming systems support the internal documentation for entire programs. When literate programs are processed, cross referencing, tables of contents, and indexes are generated automatically. Literate programming breaks code up into separate 'modules'; this makes the code easier to structure, but introduces various conventions the documentation reader must understand. The overall result, as well as a compilable program, is essentially a book (with automatically-generated cross references, contents, indices, *etc.*) providing unusually good internal documentation.

In the past, few people wrote literate programs using tools they had not written themselves — because if you built your own tool, you understood its behaviour and building your own system was easier than fathoming out the workings of someone else's tools. Furthermore, most tools make assumptions, and circumventing the imposed approach to fit in with a particular project may be harder than starting from scratch. Today, only a few literate programming tools have survived the test of time.

Despite its advantages, after almost twenty years the use of literate programming for publishing code in the mainstream literature is now negligible. In whatever ways people may be using literate programming internally in software development projects (in lab notebook type uses), it is evidently not addressing the needs of the broader research community for publication. Probably the main reason for literate programming failing to survive in the literature is that it imposes its own styles and conventions (e.g., modules), which adds significantly to the 'noise' of a paper and makes it hard to conform to journal style requirements. Ramsey's paper in this journal,[52] for example, required a section to explain the notation being used.

Nevertheless literate programming is certainly a 'good thing' and numerous variations and alternative approaches have been developed to achieve some or all of its advantages. The following are examples, and illustrate the diversity of useful approaches:

Autoduck[56] generates documentation from program source code, generating Microsoft Rich Text Format (RTF) text, which is presentable in WYSIWYG applications such as Microsoft Word. If programs are to be released to customers, technical authors can add examples and other documentation to the source code in order to generate programmer documentation. Autoduck can be used to generate online hypertext help for programs. Autoduck supports much more extensive documentation than Javadoc (*q.v.*) does, but it is much more complex than Javadoc; its limitations for writing about programs are similar.

Doc[57] allows TEX code to be documented in such a way that the code can be used directly (because doc's documentation is simply written in standard comments), but if the documentation is wanted a 'driver' file is used to obtain it. Since documentation makes files larger, and hence slower in an interpreted system like TEX, the partner system **docstrip** can be used to remove documentation.

Haskell[58] is a programming language with two styles: in the standard one, comments are conventional (i.e., text on lines following a `--` code, corresponding to Java's `//`, is ignored); in the converse, literate, style comments and program are 'swapped,' so the documentation itself is the default, and lines of progam are 'uncommented' (by `>` codes).

Javadoc[59,60] generates program interface documentation for Java: it is therefore more specialised than literate programming, though the result is intended to be browsable hypertext rather than printed. Javadoc uses an extended Java comment and is invisible to Java programming tools. It is not sensible to modify the generated documentation, and, again, the approach is not suitable for writing *about* programs.

LiSP2TEX[61] is one of several systems that places program code in the documentation. A tool reads the code, evaluates it, and merges the results into the documentation. This approach assumes that the source documentation fully defines the program.

Mathematica[62] is a combined word processor and powerful symbolic mathematics package. There are *Mathematica* journals that take *Mathematica* articles and publish them directly. Although documentation and code can be interleaved, it must be in the right order for *Mathematica*, and it is not easy to omit code (such as initialisation) that one may not want to write about. There are literate papers written in *Mathematica*,[63] and examples using concealed code, just publishing the explanation and output.[64]

Noweb[50,51] is a simplified literate programming tool, following closely in the style of Knuth's approach, but which aims to reduce the learning and effort hurdles to using literate programming. Noweb can convert a literate program into a conventional compilable program, by putting the noweb explanation into ordinary comments: this makes the resulting program more accessible, but unconstructively encourages programmers to edit a copy of, not the actual explanation. Noweb has been used for publishing programming books, such as Dave Hanson's book on C programming,[65] and it has been used (by its author) to publish papers in *Software—Practice and Experience*.[52]

Quine[11,12] is a logic language designed to generate manuals. The language includes an explanation mode, where text between the symbols { and } is copied directly and logic expressions and their evaluations are formatted in HTML unambiguously. The published papers were generated in HTML using the system to explain itself.

Soap[66] is mentioned here because it is a *really* old system: some people have been concerned with these issues since at least 1977! Soap provides a converse to literate programming: it could read documentation to extract compilable source code — and therefore helps check that what documentation says is accurate.

A wider review and details of the current state-of-the-art in literate programming, as well as tools to download, can be found at `www.literateprogramming.com`.

Part 3

Publication tools for explaining code
Warp and loom

Published papers are edited by hand, and often go through a long and arduous life-cyle; papers are often revised and, in some cases, have to conform to changing typographical requirements as they are resubmitted to different journals. Equally, the program code on which a paper is based has to conform to specific programming language, compiler and system requirements to work at all.

We now summarise the requirements for tools for supporting the reliable explanation of code for publication:

1. As a paper is revised, it is very easy for the text of the actual program to diverge from the text of the paper: program compilers and word processors have quite different criteria, and the author has quite different goals when working in each context. Under these circumstances, to ensure published papers are reliable, it makes sense to share common text automatically as much as possible. The key idea is to make it very easy to keep together and maintain what are normally separate — and sometimes independent! — documents: shared text should only be edited in one place. This is basically standard good software engineering practice: don't duplicate information unnecessarily, otherwise discrepancies soon accummulate. The requirement is to establish and automatically maintain a reliable relationship between parts of two or more files: the explanation will include or refer to fragments of program text, and the program will include (as substrings or files, or in some other way) fragments that are to be included in the documentation.

2. The shared texts (i.e., the program code appearing in the explanation) could in principle reside anywhere: in separate files, in program files, or in documentation files. Management will be easier if the number of files that require human attention is reduced. Since program development tools are usually sophisticated and interactive, and encourage the author to edit program text, all shared text should originate and reside in the program source file, where it can be edited and used without restriction to support all normal programming activities (e.g., testing), with no overhead.
3. The program source files and the documentation source files must be or remain in their standard formats; that is, they must be able to be edited and manipulated directly using existing development/word processing tools without affecting the integrity of the shared material.
4. The tool must be light-weight and so easy to use that it does not encourage any manual touching-up — even when starting to write a paper, when there is 'hardly anything to do' and it looks seductively easy to do it all by hand! In contrast, a heavy-weight system would tempt authors to do it all by hand because to do so would seem easy in comparison to getting up to speed with the tool.
5. The tool must scale, and not introduce fifteen or more of its own problems with larger projects. It should work equally well with small programs and small papers, as with large programs and long papers. In particular, the tool should be useful throughout the lifecycle of the document, as it goes from the earliest stages of drafting to final publication.
6. Whereas literate programming supports the documentation of an entire program, publication requires the author to be more selective (except for the very shortest programs). The tool must permit *any* fragments of code to be explained, and in any order that suits the explanation.
7. The approach itself must not need any special explanation when used in papers or other documents — it should simply embed fragments of code as required without imposing any notation or conventions on the reader of the paper. (Literate programming systems are excluded by this requirement.)
8. The tool must work with real code written in real programming languages. Ideally the approach should be language independent.
9. The tool must make all necessary translations between the programming language and the text processing system (e.g., if the programming language is Java and the text processing system is LaTeX, then '{' has to be converted to '\{', *etc.*). Since there are numerous programming languages and numerous text processing systems, the tool must be extensible — and it must be extensible in a well-defined way.
10. The tool must have a "commensurate level of complexity": sufficient to solve the core problems, but not having so much generality that the complexity of parameterising and using it is a hurdle in itself. (The complexity of literate programming systems is a significant hurdle limiting their widespread use.[43]) Commensurate complexity has been widely promoted through the slogan, "make the simple easy and the complex possible."

Warp: An approach to explaining code reliably

Warp is a tool for supporting the reliable publication of code that satisfies the requirements listed above.[§] Warp works with C, C++, Java, *etc*, but we will explain it here using Java.

Warp is run from the command line, operating on a Java file:

```
warp prog.java
```

This basic use of warp outputs the file `prog.java` marked up in XML, which is a widely-used, general-purpose international standard markup language.[44] For instance warp converts the Java `"A string"` to `<string>A string</string>` and `/* comment & stuff */` to `<block-comment> comment & stuff </block-comment>`. Thus warp allows any program to be considered directly as XML.

The XML version of the program generated by warp can now be processed with any of the wide range of the available XML technologies. There are tools such as XSLT[45] that can convert XML to other formats (for instance, LaTeX or HTML).

As warp has been explained so far, the XML it produces does not help find the specific parts of a program that may be needed in an explanation. Warp therefore additionally allows a programmer to introduce arbitrary XML into programs, and then the relevant parts of the program can be more easily picked out by XSLT or other tools. Here's how it is done ...

Java allows comments:

```
// ordinary Java comment on one line

/* or block comments
   over several lines
 */
```

which simply get ignored by the Java compiler (and sometimes by the programmer). Warp would mark these up in XML as described above, *except* if the comment is XML it gets copied directly. So a Java comment containing XML like `/* <highlight> */` is translated to just `<highlight>` — the comment markers themselves disappear. Such XML might be used to highlight parts of a Java program until a comment like `/* </highlight> */` which, matching the earlier `<highlight>` tag, would end the highlighting. Since this extra XML is introduced into the program as standard Java comment it is completely ignored by the Java compiler and has no effect on the meaning of the program. With this approach, warp ensures programs correspond directly to XML in a straight forward way, yet programmers can very easily introduce additional XML for any purpose.

To make it easy to identify sections of code that may be needed for an explanation, typically XML comments are written like this:

[§]The current version of warp does not permit programming language comment conventions to be changed. Warp would need trivial modifications to its lexer to work with, say, Prolog or LISP instead.

```
// <warp id="helloworld">
System.out.println("Hello World!");
// </warp>
```

Once warp processes this Java to XML, it is very easy for tools like XSLT to extract any specific code fragments as required for explanations. However, since standard XML tools (e.g., XSLT, XPath) are dauntingly complex, warp itself can do the necessary processing and, in fact, warp need not be used to generate any XML at all. Typically, then, warp is used to extract the relevant code fragments directly itself, which it converts to any of various formats as required, including LaTeX, ASCII or HTML, as well as XML.

For example, to pick out the Java section above and convert it to LaTeX, warp could be used as follows:

```
warp id=helloworld -latex prog.java > helloworld.tex
```

Or to convert it to HTML:

```
warp id=helloworld -html prog.java > helloworld.html
```

The `id=helloworld` is a simple pattern that (in these examples) matches the attribute of the `<warp id="helloworld">` tag, and causes warp to extract all text which is that tag's content, that is, up to the matching `</warp>` end tag. In turn, the `-html` flag makes warp mark up that text to HTML. In the second example above, the formatted output of warp is saved in a file, `helloworld.html`, for further processing. This trivial example might have been made more interesting by including more markup in the Java program, for instance:

```
// <warp id="helloworld"> <h1>Hello World Example</h1>
// <p>This is now a fully self-contained example!</p>
System.out.println("Hello World!");
// </warp>
```

The final feature of warp extracts just the XML attributes that have been introduced in comments that match a pattern. If warp was run in this mode on the file containing the example above, it would output `helloworld`, and thus identify the code fragment or fragments the programmer intended to be explained. Note that in this example the attribute value (`helloworld`) is also a suitable file name, so it is quite easy to write a script that extracts every code section and puts each in an appropriate file, or processes them in other ways.

In summary, warp is trivial to use for most of its intended applications, but the full power and generality of XML is available for authors who wish to do complex operations. Although warp helps explain code, it has many other uses since XML is so versatile. Here are some possibilities, all of which are easily supported:

- A Java program can be converted to XML using warp. Arbitrary transformations can be performed on the XML, for instance embedding data structures generated by other programs. XSLT can then translate the warped XML *back* to Java, where it can be edited and debugged as usual. If the data structures later require updating, warp can be used to translate the program back to XML, and then the XML tools can be used to update the data, and so on. Thus we have the advantage of normal Java development, and automatic insertion of correct data — a powerful generalisation of techniques used in some Rapid Application Development tools.

- A Java program may have a complex data structure, such as a graph. Warp can extract the data structure to an XML file, and the Java data can then be processed to display it graphically or to check it has desired properties.
- Programmers are often unable to finish programs in one sitting, and often have to create stubs to be completed later. Often comments get littered with notes, to remind programmers to finish a piece of code in a particular way. Warp can be used to identify stubs and other notes-to-programmers, which can then be extracted easily as reminders any time in the future. For example, writing `/* <note>Must add error code here</note> */` can be picked out along with all other notes, and hence help the programmers to work more reliably.
- The opposite of explaining a program reliably might be setting an exam paper on programming. Warp can be used to generate reliable exam questions: omitting code from working programs that the student is expected to supply; or it can highlight code from working programs that the student is expected to explain; and so on. In all cases, warp helps ensure the question is accurate, and based on a compiled, working program.

Finally, the current version of warp is intentionally simple; it does not do parsing, prettyprinting and many other functions that might be desirable for other purposes than for which it is intended.

Using warp

This paper is followed by a complete example to show the results of using warp in a realistic context, written and formatted like a short paper to *Software—Practice & Experience* would have been. The example was processed directly by warp (and nothing else) without using any XML tools at all, and the resulting paper was submitted to the publishers in LaTeX, so they have had very little cause to modify it whether deliberately or accidentally.

One should note that the code published in the short paper does not constitute the entire code of a complete program: for example, Java's use of `import` statements is irrelevant for the purposes of the paper, and therefore none appear in it; likewise, the actual program provides a test harness (that can, amongst other things chose between two different implementations) but none of it appears in the paper. Nevertheless, the code shown in the paper is the exact code and it has (along with its `import` and other necessary support statements) been checked by a compiler, and has been tested and run successfully.

Warp is used from the command line, and a file or files (presumed to be C, C++ or Java, *etc*) are converted to XML or other specified format (ASCII, LaTeX, . . .), as follows:

`warp -`*`format files`* ...

Convert *`files`* to the specified format (currently implemented: `-ascii`, `-html`, `-latex`, `-xhtml` and `-xml`); the default format is XML. Case is not significant, so purists can write `-XML` or `-LaTeX` if they wish.

Any XML comments in the files are copied directly by warp, but with the comment symbols stripped (unless the -all option is used: see below). An “XML comment” is any comment text (within /* ... */, or after // ...) starting with < (perhaps preceded by blanks), ending with > (perhaps followed by blanks), and containing lexically correct XML. This is a more relaxed restriction than being properly well-formed XML, since it is unnecessary to match start and end tags within the comment (though if XML’s CDATA or <!-- tags are used these would have to be contained entirely within the program comment).

When generating LaTeX, warp ‘left shifts’ code to remove any uniform indentation, though a LaTeX macro is provided so the original indentation can be restored if desired. Thus methods and nested blocks, and so on, can be explained without their original nesting looking inappropriate in the context of the final document.

warp -*format* *pattern* *files* ...

If a pattern is specified, warp extracts, converts and concatenates all matching XML sections of *files*. This is warp’s normal mode of use.

warp *pattern* *files* ...

List all matches of the pattern in *files*. The output is formatted as lines of text in the same syntax as a warp pattern, and can therefore be reused in another run of warp (e.g., in a shell script).

warp -*format* -all *files* ...

Convert *files* to the specified format, ignoring the special status of XML comments, and hence *including* comments containing XML. (The flag -all simply stops warp recognising XML in comments.) The main purpose of -all is to help authors review complete code within the relevant document processing system.

warp -help

Provide summary help of warp (including some details we do not cover in this paper, including the use of XML DTDs, and a description of patterns).

Comparing loom and warp

Janet Incerpi and Robert Sedgewick’s Loom[46,47] was used originally for Sedgewick’s book *Algorithms*[8] Loom has no published description (there is only an example of its use[46]): a detailed comparison of loom and warp is therefore justified here.

Loom extends comments by matching forms /* include *name* */ to /* end *name* */. This defines named sections of code, and the effect is very similar to warp’s XML approach, which would be to use /* <include stuff="*name*"> */ to /* </include> */.

A loom documentation file then includes the named sections using %include commands:

%include *file section*

or

`%include` *file section* | *command*

which (when processed by loom) are replaced by the named section, which loom locates in the named file. If a command is specified, the named section of code is filtered through it (the command is a Unix filter, and could be used possibly even to compile and run the section of code to insert its output).

This is similar to warp, except warp writes the named section to a file and the documentation system reads the file. Loom can transform the program text inside the documentation, by using the command filter; in contrast, warp processes the file outside of the documentation — but this allows the documentation to use a standard include, rather than a special syntax as loom requires. Warp can be used directly with proprietary tools such as Microsoft Word (which can include files), whereas loom would need to be modified to parse the Word format (and it would create two versions of the documentation file, one with and one without the code included).

Apart from the syntactical details, the main difference to using warp and loom is that loom must process all files whereas warp only processes the program. If the author completely forgets to run loom, then the formatted documentation will contain no code; if the author forgets to run warp, however, the code files will not be created and there will be errors reported by the formatter (or an older version of the code will be used if the files are still available).

Warp and loom are compared in more detail in Table I.

Neither loom nor warp will be the last word in tools for explaining code. Fortunately both of them are small, simple programs: developing and evaluating new tools in this area will make a worthwhile and interesting project.

Confidence

The point of using an approach like loom or warp is to increase confidence in published results; moreover sufficient integrity has to be achieved with reasonable cost. We now discuss issues affecting confidence a little more closely.

The opposite approach to using markup would be to use a WYSIWYG mechanism, such as publish-and-subscribe:[48] but in all commercial implementations, the design goal of "ease of use" makes it *far too* easy to make changes in one place and not the other. Small edits can lead to unnoticed and unknown consequences, particularly as a WYSIWYG system leaves no mark up trail to show explicitly that changes have been made.

When writing internal documentation, as opposed to explaining code, an issue is to ensure that *all* of the program is documented, or at least that nothing relevant is omitted from the documentation. On their own, loom and warp, but unlike a literate programming system, cannot make any such guarantees. However, for writing about programs, it does not matter — indeed it is helpful — when bits are concealed.

Any process that allows explanation to take full advantage of a typesetting system is open to abuse. One might insert hand-crafted code that looks like it has been warped, but has none of the warranties doing it automatically achieves. Warp itself permits anything that XML permits, which is a lot; however, when used with its internal filters, what the author can do is more restricted (e.g., code cannot be deleted without going to unreasonable effort).

Table I. Comparison of warp and loom.

warp	loom
Standard program code with comments.	Standard program code with comments.
Standard documentation file.	Special `%include` command in documentation.
Requires no external programs for basic use.	Requires special filters for use with most text formatters (such as LaTeX).
Code can specify typographical or other features for documentation.	Code cannot specify any typographical features, but may be arbitrarily transformed (by filters) in documentation.
Designed to concatenate code together from several parts of a program.	Does not concatenate code.
Must be run before documentation is formatted if code has been updated.	Must be run when documentation is formatted.
Main advantages	
Documentation can be given to publishers without any special instructions.	
Both code and documentation files remain standard source files that need no special processing.	Only code remains standard.
Documentation can make full use of any multi-file structuring facilities of formatter.	
Uses XML. Code can be processed in any way if desired.	Ability to filter code using Unix tools, and the filter command is embedded in the documentation.
Forgetting to generate files will get error messages from the documentation system.	
Main disadvantages	
	Forgetting to run loom will generate no diagnostics (and in LaTeX, loom's `%include` syntax is a comment, so generates nothing).
	Documentation cannot use structuring (e.g., be a nested multi-file document).
Documentation is not a single file (but the generated files never need editing or looking at).	Original documentation cannot be distributed without complete program and loom.

Nevertheless a high integrity version of loom or warp might closely restrict what is allowed so that code cannot be modified.

Design decisions with warp

The first version of warp used comment codes like `//>` *`file`*, which while neat, brief and Unix-like, were easy to mistype, hardish to search for when editing (all the symbols used were existing Java operators), incomprehensible to third-parties, and lacking in redundancy — many errors in their use were undetectable (e.g., there would be no warning if you failed to shift the `>` key and accidentally typed a dot instead: `//.` — which would be just ignored comment). In short, they suffered from all of the problems of conventional literate programming codes, `@[`, `@;`, `@'` and so on (there are over thirty such codes, plus a collection of cryptic TEX macros, such as `\0`). Indeed when Knuth and Levy say of the `@'` code that "this code is dangerous"[49] you know something is wrong, and to be avoided for a reliable system!

The next version of warp used mnemonic codes (such as `// $save$` *`filename`*), more like loom, but these were still arbitrary and difficult to make powerful-enough without inventing a whole new language. The insight, suggested by George Buchanan, was to replace warp's codes with XML. At a stroke, one was using a standard notation, and warp changed from being a special purpose tool into a general purpose tool.

XML is verbose for different reasons than warp; nevertheless the overhead in typing the extra characters should be seen in perspective. XML text can be shared, reused and checked easily, and the savings in errors avoided can be considerable. Perhaps programmers' habitual preference for special characters and short cuts is, more, a symptom of poor design and the fact that, normally, we have to repeatedly edit and re-edit things for multiple purposes — we naturally wish to reduce the effort of repeated editing. A better way is to increase the reuse of shared texts, as here, to multiply the impact of our work rather than to make it quicker to edit *per se*: easier editing is not the goal, writing reliably is, and therefore editing less often. With longer keywords and a stricter syntax, ultimately we do less work and we achieve our real goals faster — much the same rationale as high level languages.

Generalised explanations of code

Literate programming and the other approaches described here explain programs to people who are interested in their operation. Future research should find better and more reliable ways of explaining programs to their non-specialist users as well, perhaps by automatically taking advantage of some comment scheme, and perhaps by generating reliable interactive help. Programmers would then become a bit closer to the explanatory needs of their programs' end users. This might encourage programmers to make their programs easier to understand.

Elsewhere we have discussed writing better manuals for users[12,67,68] (user manuals being explanations of programs for users, rather than for computer scientists); we have also discussed the useful impact of quality explanation on programming language design[69] and on physical device design.[70]

Of course, programming, writing and explaining are vast areas in their own right; see reference 71 for explanation by visualisation, reference 72 for a review of annotation, reference 73 for a proposed approach to multiple-use documents, and reference 48 for a summary of commercial linking, embedding, and publish-and-subscribe technologies.

Conclusions

Programming can be undertaken for many reasons, but if the purpose is to advance computer science, by developing and testing code to a high enough standard to publish then we are doing science. A range of tools support doing quality computer science, from managing the internal documentation of the lab book to supporting reliable publication of code. Although there are overlaps, literate programming mainly supports the laboratory book end, and tools like loom and warp mainly support the journal publication end.

Compared to literate programming, both loom and warp are much simpler for the author, for the reader, and for the publisher. Their simplified approaches avoid the intellectual and typographical hurdles conventional literate programming approaches impose. The approaches are ideal for explaining algorithms, which typically requires code extracts, rather than for writing *complete* readable programs, which remains conventional literate programming's forte.

Unlike loom, warp treats programs as XML documents, and it does so with such a light touch that the programs can still be edited and developed as normal, using existing editors and tools. Although XML can be processed in many ways, warp provides a scheme for extracting sections of code and for transforming them into LaTeX, XML or other formats for processing in other documents, and for including in explanation in particular. Warp is probably the simplest possible scheme, yet remains enormously versatile thanks to its use of XML.

In an ideal world, it would not be necessary to have any separation between code and published explanation. Even though some tools, such as *Mathematica*, make them almost the same, this is the exception — and the awkwardness of working with "almost the same" rather than "exactly the same" is a great impediment for reliable writing, because the author is continually either compromising or using work arounds — for example *Mathematica* does not allow an author to change the order of code or to quote code fragments to explain them better. Unless we want to publish code in 'toy' languages designed specifically for the purpose, we will probably achieve no fundamentally better solutions than those discussed here.

Knuth writes that "Science is what we understand well enough to explain to a computer";[74] yes, and computer science will progress better when those programs are explained more reliably to the scientific community.

Warp can be obtained from `http://www.uclic.ucl.ac.uk/harold/warp`

Acknowledgements

Ann Blandford, Roland Backhouse, George Buchanan, Paul Cairns, Anthony Finkelstein, Michael Harrison, Matt Jones, Peter Ladkin, Gary Marsden, Norman Ramsey and Russel Winder made constructive comments on this work. Graham White solved a Java system bug by compiling warp on another system that behaved differently (and hence revealed the cause of the bug) — so much for Java being portable! The author would also like to thank the anonymous referees for their contributions to this paper. Harold Thimbleby is a Royal Society Wolfson Research Merit Award holder.

REFERENCES

1. D. E. Knuth, "Literate Programming," *Computer Journal*, **27**(2):97–111, 1984.
2. D. Deutsch, *The Fabric of Reality*, Penguin, 1997.
3. N. Cartwright, *How The Laws of Physics Lie*, Oxford University Press, 1983.
4. J. Ziman, *Reliable Knowledge*, Canto ed., Cambridge University Press, 1991.
5. H. Abelson, G. J. Sussman & J. Sussman, *Structure and Interpretation of Computer Programs*, First Edition Preface, MIT Press, 1985.
6. A. J. Perlis "A New Policy for Algorithms?" *Communications of the ACM*, **9**(4):255, 1966.
7. C. B. Dunham, "The Necessity of Publishing Programs," *Computer Journal*, **25**(1):61–62, 1982.
8. R. Sedgewick, *Algorithms*, Addison-Wesley, 1983.
9. D. Libes, "Writing a Tcl Extension in Only 7 Years," Proceedings of the Fifth Annual Tcl/Tk Workshop'97, 14–7, 1997.
10. P. Underhill, *Why We Buy: The Science of Shopping*, Texere, 2000.
11. P. B. Ladkin & H. Thimbleby, "From Logic to Manuals," *Software Engineering Journal*, **11**(6):347–354, 1997.
12. P. B. Ladkin & H. Thimbleby, "From Logic to Manuals Again," *IEE Proceedings Software Engineering*, **144**(3):185–192, 1997.
13. Kuan (Kwan or Guǎn) Mei-Ko, "Graphic Programming Using Odd or Even Points," *Chinese Mathematics*, **1**:273–277, 1962.
14. Y. Lin & Y. Zhao, "A New Algorithm for the Directed Chinese Postman Problem," *Computers and Operations Research*, **15**(6):577–584, 1988.
15. G. Chartrand & O. R. Oellermann, *Applied and Algorithmic Graph Theory*, McGraw-Hill, 1993.
16. S. Skiena, *The Algorithm Design Manual*, Springer Verlag, 1998.
17. D. Jungnickel, *Graphs, Networks and Algorithms*, Algorithms and Computation in Mathematics, **5**, Springer, 1999.
18. R. K. Ahuja, T. L. Magnanti & J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall (Simon and Schuster), 1993.
19. M. Dror, ed., *ARC ROUTING: Theory, Solutions and Applications*, Kluwer Academic Publishers, 2000.
20. D. E. Knuth, *The Art of Computer Programming*, **1**, 3rd. ed., Addison-Wesley, 1997.
21. M. F. Porter, "An Algorithm for Suffix Stripping," *Program*, **13**(3):130–137, 1980.
22. M. F. Porter, "Snowball: A Language for Stemming Algorithms," `http://snowball.sourceforge.net/texts/introduction.html`, 2001.
23. H. W. Thimbleby, "An Efficient Equivalence Class Algorithm with Applications to Autostereograms," *Software—Practice & Experience*, **26**(3):309–325, 1996.
24. H. W. Thimbleby, "Using Sentinels in Insert Sort," *Software—Practice and Experience*, **19**(3):303–307, 1989.
25. H. W. Thimbleby, "Specification-led Design for Interface Simulation, Collecting Use-data, Interactive Help, Writing Manuals, Analysis, Comparing Alternative Designs, etc," *Personal Technologies*, **4**(2):241–254, 1999.
26. H. W. Thimbleby, P. Cairns & M. Jones, "Usability Analysis with Markov Models," *ACM Transactions on Computer Human Interaction*, **8**(2):99–132, 2001.
27. J. R. Anderson & C. Lebiere, *The Atomic Components of Thought*, Lawrence Erlbaum Associates, 1998.
28. D. MacKenzie, *Mechanizing Proof*, MIT Press, 2001.
29. T. D. Sterling & J. J. Weinkam, "Sharing Scientific Data," *Communications of the ACM*, **33**(8):112–119, 1990.
30. R. P. Feynman, "Cargo Cult Science," in *Surely You're Joking Mr. Feynman!* R. Hutchings ed., Vintage, 1992.
31. G. E. Forsythe, "Algorithms for Scientific Computation," *Communications of the ACM*, **9**(4):255–256, 1966.
32. R. W. Hall & J. G. Partyka, "On The Road to Efficiency," *OR/MS Today*, **24**(3):38–47, `http://lionhrtpub.com/orms/orms-6-97/Vehicle-Routing.html`, 1997.

33. J. S. Brown & P. Duguid, *The Social Life of Information*, Harvard Business School Press, 2000.
34. H. W. Thimbleby, "You're Right About the Cure: Don't Do That," *Interacting with Computers*, **2**(1):8–25, 1990.
35. R. Park, *Voodoo Science*, Oxford University Press, 2000.
36. H. A. Simon, *The Sciences of the Artificial*, MIT Press, 3rd. ed., 1996.
37. K. Beck & E. Gamma, JUnit, `http://www.junit.org/`, 2001.
38. E. S. Raymond, *The Cathedral & The Bazaar*, O'Reilly, Revised Edition, 2001.
39. H. Thimbleby, "A Critique of Java," *Software—Practice & Experience*, **29**(5):457–478, 1999.
40. *ACM Code of Ethics and Professional Conduct*, `http://www.acm.org/constitution/code.html`, 1992.
41. H. W. Thimbleby, "Experiences with Literate Programming Using CWEB (A Variant of Knuth's WEB)," *Computer Journal*, **29**(3):201–211, 1986.
42. D. B. Resnik, *The Ethics of Science*, Routledge, 1998.
43. C. J. Van Wyk, "Literate Programming: An Assessment," *Communications of the ACM*, **33**(3):361&365, 1990.
44. R. Harold & W. S. Means, *XML In a Nutshell*, O'Reilly, 2001.
45. D. Tidwell, *Mastering XML Transformations: XSLT*, O'Reilly, 2001.
46. D. R. Hanson with C. J. Van Wyk (moderator) & J. Gilbert (reviewer), "Literate Programming," *Communications of the ACM*, **30**(7):594–599, 1987.
47. D. R. Hanson, "loom — weave fragments together," `loom.1` in `ftp://ftp.cs.princeton.edu/pub/people/drh/loom.tar.gz`, 1987.
48. Microsoft Corporation, *Getting Started: Microsoft Office*, Document No. OF64076-0295, 1992–1994.
49. D. E. Knuth & S. Levy, *The CWEB System of Structured Documentation, Version 3.0*, Addison-Wesley, 1994.
50. N. Ramsey, "Literate Programming Simplified," *IEEE Software*, **11**(5):97–105, 1994.
51. N. Ramsey, `http://www.eecs.harvard.edu/ñr/noweb/`, 2002.
52. N. Ramsey, "A simple solver for linear equations containing nonlinear operators," *Software—Practice & Experience*, **26**(4):467–487, 1996.
53. D. Mall, `http://www.literateprogramming.com`, 2001.
54. N. Ramsey & C. Marceau, "Literate Programming on a Team Project," *Software—Practice and Experience*, **21**(7):677–683, 1991.
55. D. E. Knuth, *The Stanford GraphBase*, Addison-Wesley, 1993.
56. E. Artzt, Autoduck, `http://www.literateprogramming.com/autoduck.pdf`, 2001.
57. F. Mittelbach, *The doc and shortvrb Packages*, Technical Report, Gutenberg Universität Mainz, 1997.
58. S. Peyton Jones & J. Hughes, eds., L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman & P. Wadler, *Haskell 98: A Non-strict, Purely Functional Language*, `http://haskell.org/onlinereport`, 1999.
59. K. Arnold, J. Gosling & D. Holmes, *The Java^{TM} Programming Language Second Edition*, 3rd. ed., Addison-Wesley, 2000.
60. L. Friendly, "The Design of Distributed Hyperlinked Programming Documentation," in S. Fraïssé, F. Garzotto, T. Isakowitz, J. Nanard & M. Nanard (Eds.), *Hypermedia Design*, Proceedings of the International Workshop on Hypermedia Design (IWHD'95), pp.151–173, Springer, 1996.
61. C. Queinnec, *Literate Programming from Scheme to TeX*, Université Paris 6 & INRIA-Rocquencourt, 2000.
62. S. Wolfram, *The Mathematica Book*, 4th. ed., Addison-Wesley, 1999.
63. H. W. Thimbleby, "Specification-led Design for Interface Simulation, Collecting Use-data, Interactive Help, Writing Manuals, Analysis, Comparing Alternative Designs, etc," *Personal Technologies*, **4**(2):241–254, 1999.
64. H. W. Thimbleby, "Analysis and Simulation of User Interfaces," BCS Conference on Human-Computer Interaction, S. McDonald, Y. Waern & G. Cockton, eds., **XIV**:221–237, 2000.
65. D. R. Hanson, *C Interfaces and Implementations*, Addison-Wesley, 1997.
66. R. S. Scowen, "Some Aids for Program Documentation," *Software—Practice & Experience*, **7**(6):779–792, 1977.
67. M. A. Addison & H. Thimbleby, "Intelligent Adaptive Assistance and Its Automatic Generation," *Interacting with Computers*, **8**(1):51–68, 1996.
68. P. B. Ladkin & H. Thimbleby, "A Proper Explanation When You Need One," in M. A. R. Kirby, A. J. Dix & J. E. Finlay eds., *BCS Conference HCI'95*, People and Computers, **X**:107–118, Cambridge University Press, 1995.

69. H. W. Thimbleby, "Java: A Critique," *Software—Practice & Experience*, **29**(5):457–478, 1999.
70. H. W. Thimbleby, "Calculators are Needlessly Bad," *International Journal of Human-Computer Studies*, **52**(6):1031–1069, 2000.
71. B. Braune & R. Wilhelm, "Focusing in Algorithm Explanation," *IEEE Transactions on Visualization and Computer Graphics*, **6**(1):1–7, 2000.
72. I. A. Ovsiannikov, M. A. Arbib & T. H. Mcneill, "Annotation Technology," *International Journal of Human-Computer Studies*, **50**(4):329–362, 2000.
73. T. A. Phelps & R. Wilensky, "Multivalent Documents," *Communications of the ACM*, **43**(6):83–90, 2000.
74. D. E. Knuth, Foreword, in M. Petkowvšek, H. S. Wilf & D. Zeilberger, $A = B$, A K Peters, 1996.