Explaining Programs Reliably

Harold Thimbleby*

School of Computing Science, Middlesex University, Bounds Green Road, LONDON, N11 2NQ.

SUMMARY

Ensuring integrity between code and documentation, so that programs can be written about reliably, whether for explaining them in scientific papers or books, requires tool support. A light-weight and flexible approach that is easy to use and easy to implement is described.

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

1. Introduction

Science is about finding better explanations [4], and to do so it is guided by principles, such as Occam's Razor, and a conviction that reality is fixed. In computer science, instead, we build the objects we explain, and we change them — and perhaps we explain them again. Computer science, then, is about finding better explanations for things that can be changed to help improve the explanations.

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. All these activities involve human intervention, typically using word processors: as and when possible improvements are spotted, the author or authors revise the documents. Since all the documents in a particular project are related, a lot of text in the various documents is similar if not exactly the same, thus providing the opportunity for computer support to help explain programs reliably. In this paper we are concerned specifically with writing better, more reliable, explanations for programs, for instance as might be required in journal papers or books on algorithms.

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,

*Email: harold@mdx.ac.uk

alignment, point size and so on; if a mark-up language (like 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For example, the symbol '`&`' has to be edited to '`&`' for HTML or to '`\&`' for LaTeX — and of course this mark-up is no longer valid in programs, so accurate conversion to documentation cannot readily be confirmed with a compiler.

Occasionally tools will be constructed to help automate the process (Loom [6] is an example of this used for Sedgewick's book *Algorithms* [18]), but for writing small papers, the overhead of building the tools seem out of proportion. 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 a diversion.

Typically a fragment of program code is cut-and-pasted from a working program into the documentation (if it is small, it may even be retyped *in situ*), and it is then edited carefully to conform to the documentation system's requirements (and to the typographical requirements, particularly line length and indentation). Inevitably some changes will eventually be made, say, in the program. This may be the first of several changes anticipated, so it perhaps isn't worth going to the trouble of cutting-and-pasting-and-editing again. Maybe it seems easier to try to manually duplicate changes in the previously prepared text?

If there is any hurdle to making changes, eventually not all of them will be made. Managing documents and programs is normally such hard work that either improvements do not get made, or changes do get made (say to the documentation) but are not accurately reflected everywhere else (say in the program). For example, the documentation might read better if a name is changed ... this is the thin end of the wedge, and a subtle threat to integrity.

Reliable writing has to be made *so easy* that there is never a temptation to avoid improving *both* code and documentation consistently. For once even a few discrepancies accummulate, it becomes an even harder exercise to restore integrity.

To ensure documents are reliable, despite being largely edited by hand, it makes sense to share common text as much as possible automatically. Apart from generating high quality documents, the key idea is to make it very easy to keep together and maintain what are normally separate — and sometimes increasingly independent! — documents. *Shared text should only be edited in one place.* Various solutions to this problem have been proposed, and all have considerable advantages over approaching the process manually.

Literate programming was introduced by Knuth [8, 19] to combine programs and their documentation to create readable programs. Numerous programs have been prepared using literate programming: sizable examples include Knuth's own outstanding books (e.g., [7]). There is no duplication of either code or documentation: there is only one shared copy. In literate programming code and documentation are interleaved in a file, and as they are adjacent 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 programs support the internal documentation for entire programs. When literate programs are processed, cross referencing, tables of contents, and indexes are generated automatically. The overall result, apart from a compilable program, is essentially a book providing unusually good internal documentation. Although literate programming has considerable advantages, which lead to improvements in both program code and in

documentation, it is a heavy-weight approach. It involves using numerous control codes, and traditionally generates specialised TEX code (to create nice typography). An impediment to using literate programming is that the typographical conventions used may be inappropriate (and too complex to fix): for example, a journal article must conform first to the requirements of the journal and only secondarily to any literate programming conventions.

Another reason to avoid using literate programming is that it is based on a core source file, the so-called web document. In many programming environments, this special document is meaningless. For instance an integrated visual debugger probably needs to access the source code file directly; if the programmer is tempted to edit the source code from within the debugger, then the correspondence with the web file is compromised.

Nevertheless literate programming is certainly a 'good thing' and numerous alternative approaches have been developed to achieve the same sorts of advantages. The following are examples:

Noweb [17] is a simplified literate programming tool, closely in the style of Knuth's approach, but which aims to reduce the learning and effort hurdles to using literate programming. As a tool, it is language independent. Noweb can convert a noweb source program into a conventional program, putting the noweb explanation into conventional program comments: this makes the documented program more accessible, but unconstructively encourages programmers to edit a copy of the actual explanation.

Javadoc [2, 5] generates program interface documentation for Java. Javadoc uses an extended Java comment and is invisible to Java programming tools. It is not sensible to modify the generated documentation, and the approach is not suitable for writing *about* programs.

Mathematica [23] 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 does not want to write about. An example literate paper written in *Mathematica* is [21].

LiSP2TEX [16] is one of several systems that places program code in the documentation. A tool reads the code, evaluates it, and splices the results into the documentation. This approach assumes that the source documentation fully defines the program.

Haskell [14] is a programming language with two styles: in the standard one, comments are conventional (i.e., text on lines following a `--` code, like Java's `//`); in the converse, literate, style comments and program are 'swapped,' so the documentation itself is the default, and lines of progam are 'uncommented' (by `>` characters).

Quine [11] is a simple language designed to generate manuals. The published paper, [11], was generated in HTML by the system documenting itself.

Elsewhere we have discussed writing better manuals for users [1, 10, 11]; we have also discussed the useful impact of quality explanation on programming language design [20] and on physical device design [22]. Of course, programming, writing and explaining are vast areas in their

own right; see [3] for explanation by visualisation, [9] for a review of annotation, [15] for a proposed approach to multiple-use documents, and [12] for a summary of commercial linking, embedding, and publish-and-subscribe technologies.

This paper describes yet another approach. Its main novelty is that it fits in both with existing interactive program development tools and with existing authoring tools: the program source files and the documentation source files are in their standard formats, and can be edited and manipulated directly without affecting the integrity of the shared material. It is a language independent approach. It is very simple. It is called warp: warps are part of woven material, so there are etymological connections with literate programming, if not entomological — literate programming uses terms such as web and weave; warp is conveniently both a noun and a verb; warp has a connotation of high speed achievement, thanks to *Star Trek*; and ironically, warp means a devious twist from the truth.

The only requirements to warp are:

- On the documentation system: that it can include files — in LaTeX, and in many other systems, this is easy;
- On the programming language: that it has comments.

In warp's case the programming language is Java (or any language with the same comment conventions, such as C or C++) and the documentation is written in LaTeX but it would be easy to work with other languages.

Here is a very small example illustrating discussion of a trivial Java program:

> This method
>
> ```
> public static void main()
> { // The world's standard program
>   System.out.println("Hello world!");
> }
> ```
>
> is defined in this class
>
> ```
> class Example
> {
>   ...
> }
> ```

The code shown above is the exact code, direct from the program. This code has been checked by a compiler and has been run. But,

- Especially with such a simple example, it might have been tempting to do it all by hand. We don't want a heavy-weight approach that encourages any hand-editing — even when starting out, when it looks seductively easy!
- We don't want an approach that does not scale, and introduces fifteen of its own problems with larger projects.

`// $save$` *file*	Causes the following lines to be saved to the specified file, automatically using the appropriate LaTeX mark-up, fonts, etc.
`// $skip$`	Ignore following lines.
`// $save$`	Resume copying lines to the last named file.
`// $literal$` *text*	Literal text, copied directly.
`/**` *text* `*/`	Javadoc documentation, converted to LaTeX comment.
`// $comment$` *text*	Real comment; ignored both by Java and by the copying process.
`// $note$` *text*	Put a note in the documentation.

Figure 1. Warp codes

2. From program to explanation

Java allows ordinary comments:

```
// ordinary Java comment
```

which simply get ignored by the compiler (and sometimes by the programmer).

Warp extends Java's comments by using keywords after the `//`. In fact the notation can be used in any file (or stream), not just Java source code, including program output.

The following is the complete Java source file that created the example above (albeit unrealistically cramped to illustrate many features concisely!).*

*A feature, specifically for use in this paper, but for not mentioned above, copies entire files including all comments, and converts them to LaTeX. This feature was used to ensure the complete source code example in this paper was printed correctly.

```
// $save$ class.tex
// $comment$ A toy example!

class Example
{
  // $skip$
  private int secret; // e.g., something we don't want to explain

  // $save$ main.tex

  /** This is a javadoc style comment,
   *  which would normally be used to document the main() method, next.
   */

  public static void main()
  { // The world's standard program
    System.out.println("Hello world!");
  }

  // $save$ class.tex
  // $comment$ complete the end of the class with some dots
  // $literal$ ~~\ldots\\
}
```

Warping this creates the two named files (`main.tex` and `class.tex`), and one could then write about the program by including the two files in documentation — with results like the example we started with. The corresponding LaTeX source is as follows:

```
This method
\input{main.tex}
is defined in this class
\input{class.tex}
```

Because the LaTeX version of the program text is not actually present in the documentation it is unlikely to be edited accidentally — nor is the program source code itself.

When it is run, the output of the simple example will be much as expected — and it is not worth explaining further in the context of this paper! More complex output, however, often requires explanation and faithful presentation in the context of the explanation, just as code does. It is useful that warp can be used to process output too: all that is necessary is to insert appropriate comments (such as `$save$`) and warp it. Output will be converted reliably to LaTeX's conventions and saved in a file so it, or the interesting parts of it, can be included into the explanation.

Javadoc comments are conventionally meant to provide internal documentation for Java: here warp turns them into LaTeX comments. For example, the Java code

`/** documentation...`

is turned into the LaTeX comment

`% /** documentation...`

so the original programmer's explanation can still be read in the LaTeX source within the rest of the LaTeX translation of the code, but it will not appear in the typeset result.

The purpose of the `$note$` warp code is to help the author remember ideas for later, whether when reading the program or the documentation, or to pass on hints to a documentation writer (who is often someone else). Notes are also displayed when the program text is processed.

2.1. High integrity issues

There are two sorts of high integrity issues: general ones, and ones specific to warp.

2.1.1. General integrity issues

The paper about Quine [11] was accidentally an illustration of a general point about writing reliably about programs. The paper was a corrected reprint of a paper previously published, which contained errors because of manual intervention during the publishing process. If one wants to explain programs reliably, the entire process should be supported.

The opposite approach to using markup (as LaTeX does) would be to use a WYSIWYG mechanism, such as publish-and-subscribe [12]: but in all commercial implementations, the design goals of "ease of use" makes it *far too* easy to make changes — as the equivalent of the explicit markup is not visible to users, small edits can lead to unnoticed and unknown consequences. The advantage of markup (like `// $save$`) is that it is visible and says what it means; whereas meaning in WYSIWYG systems generally depends on user actions, which are not directly recorded, so there is no explicit record of how one got to the current document state or exactly what it means. (In particular it makes version control difficult: it makes it difficult to get all the supposedly coordinated writing and programs back to earlier stages. It makes thoroughly coordinating multiauthored documents impossible.) Nevertheless, the inappropriateness of current WYSIWYG mechanisms for the purpose does not preclude the eventual development of a properly integrated program development and explanation environment as reliable as the current system and which is easier to use (or perhaps just more appealing).

One aspect of integrity concerns authors' file access rights. Apart from the choice of file names, which will be fairly stable as a program is developed, warp separates explanation from program code into different files, so there is very little necessary overlap between programming and explaning. This is an advantage for organisations or projects that enforce a rigid separation between programming and technical authoring activities.

2.1.2. Specific integrity issues 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 (all the symbols used were existing Java operators), incomprehensible to third-parties, and lacking in redundancy — so many errors in their use were undetectable (e.g., there would be no warning if you failed to shift the `>` key and accidentally typed `//.`). 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 & Levy say of the `@'` code that "this code is dangerous" [8] you know something is wrong, and to be avoided for a reliable system!

It is interesting that while warp's verbose approach for such a simple system stands out as unconventionally excessive, when the text can be shared, reused and checked so easily, the overhead in typing the extra characters should be seen in perspective. 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. With longer keywords, we do less editing, and achieve our real goals faster.

Warp has various loopholes and is not a high integrity tool. These are its weaknesses:

- Java can disappear between `// $skip$` and `// $save$` codes; indeed Java disappears before the very first `// $save$` that starts copying to a specified file. In a high integrity approach, losing code should be forbidden; warp merely notes if any text is lost.
- The `// $literal$` code could be used, accidentally or deliberately, to divert Java code into a black hole inside LATEX. In a high integrity approach, either `// $literal$` should be banned, or there would be a robust way of checking it (e.g., allowing only a very limited range of LATEX commands).
- Warp does not check whether fragments of code are used exactly once: in particular it does not guarantee that any document actually uses the generated files. (A more sophisticated system might scan the LATEX files for `\input` commands.)
- Auditing and internal documentation is separate (cf. [2]) as is, for example, ISO 9000 quality conformance.

Yet warp is simple and flexible. Code can be deliberately skipped invisibly, as this is generally helpful for clear writing; indeed, two files can interleave the code they document. A more disciplined approach might be usefully enforced for some applications: for instance one where files strictly nest, and where an ellipsis (e.g., " ... ") is automatically inserted to indicate where any text has been diverted to a nested file. (In fact this is the form of the example used in this paper.) It would also be clearer if, further, the documentation file nesting was restricted to match well-formed Java structures. More generally, there could be two modes: one for internal documentation (where nothing can be invisibly concealed) and one for external documentation (which perhaps prefers brevity over pedantry).

One aspect of integrity is how reliable the explanation can be or is likely to be. In conventional literate programming (whether Knuth's style [8] or noweb's [17]), it is possible — indeed routine — to break code into chunks to make it easier to document. For example, the body of a `for` loop can be separated from the loop itself. This flexibility allows the explanation of the body to be discussed separately from the implementation of the body. In contrast, warp can only handle 'chunks' of adjacent program code, so the middle of a construct cannot be separated out to be explained separately. This is an advantage: to explain a `for` loop one might mention invariants and so forth; but if the body of the loop is elsewhere, then where are the

guarantees that the body implements the specification? In any case, in modern programming languages a programmer would likely rewrite a complex or lengthy piece of code as a procedure or method. Doing so would make the semantic relation between the loop and its body well-defined (specifically, the semantics of procedure calls), and therefore easier to explain reliably.

2.2. Transparent features

When warp generates a file from program source, it is made to start with a special marker text, currently

```
% WARNING! Generated from <file>.
% Can be overwritten automatically.
```

This is comment in LaTeX, and has no impact on the typeset documentation. Before warp saves text to a file, it checks the file starts like this (with the full name of the file inserted), or it must be a new file. Note that the original Java source file is mentioned: this prevents the same-named file being overwritten by different Java files. This normally-invisible precaution stops a program author accidentally overwriting files that were not originally generated by the documentation process, or from overwriting files because they accidentally used the same file name more than once in different Java files. The final case is if they had used the same file name more than once within a single Java file: code would be appended, so again nothing is lost.

(There are a few more lines providing further information, which are not part of the security check.)

When all the lines have been collected warp 'left shifts' them to remove any uniform indentation, but LaTeX code is provided so the original indentation can be restored if desired. This is done so that methods and nested blocks, and so on, can be explained without their original nesting (which was appropriate in the program source code) looking inappropriate in the context of the final document.

When files are saved, they are of course converted to adhere to LaTeX's conventions (e.g., `&` becomes `\&`): in fact, there is a small bug in LaTeX 2_ε, and this is fixed too, every time, and with no effort from the user.

2.3. Auditing

Although the correspondence between the source code program and the extracts in the explanation is assured, there is no automatic control over the quality of the explanation itself. The author might include the wrong file, be confused over the explanation, or the code might be changed and someone forget to update its explanatory text. Even for publishing papers, the scope for error is high — especially given the long lead times to publication where the author has many opportunities to be interrupted and may forget to work through all consequences of a change.

Warp could be extended relatively easily to achieve two goals of long-term integrity: that (i) somebody can record there is an acceptable correspondence between every program extract

and the explanation; (*ii*) when the correspondence changes, through updates to the program source, the author (or auditor) is told where explanatory changes may be required.

A documentation author can say, as it were, "this document is not finished" and the auditing approach will continue to report where further work is required. Or a documentation author can say "this document is finished" and when the program is subsequently changed, the author will be directed to just those places that need direct attention. One might also want to inform the program author of relevant changes made to the documentation, but this requires more assumptions about the documentation structure (but see [10]).

When the modified warp generates a source file, it would include in the transparent preamble a command with as parameter a unique identity for the source file's contents. (The identity may be a hashcode, or could be intrinsically useful, such as a time stamp, the programmer's name and a duplicate of the source code itself.) The documentation associates with every included fragment of code a call to an auditing function, which compares the tool-generated identity with the author's explicitly provided identity. The author is notified when the identities are different, which can be done in several ways, such as causing the explanation to be typeset distinctively, inserting a footnote, or creating an entry in an auditing appendix.

When an author is developing documentation, whenever a section passes its audit, the relevant identities are noted and copied into the documentation. Further typesetting of the documentation confirms that the explanation and code correspond satisfactorily because the identities match. Now when any source code changes its identity will change and the typesetting process will draw attention to that change, in whatever way is appropriate for that type of documentation.

Finally, there is a simple improvement to warp. Every file generated should include some information so that (provided at least one file is read) the documentation system can check all files generated are included and, if desired, check they are also included in the right order.

3. Conclusions

Literate programming not only makes explaining programs easier, it also improves quality by combining two normally separate activities — programming and documenting — and making both easier to do well. Program and documentation are side by side, and they are more likely to be consistent. In our light-weight approach, exemplified by warp, many of the same benefits as in conventional literate programming are achieved, but the purpose is to write about programs, rather than to document them.

The approach exhibited in this paper is ideal for explaining, writing *about* programs (and their outputs), rather than for writing *complete* readable programs, which remains conventional literate programming's forte. However light-weight literate programming is much simpler and avoids the intellectual hurdles conventional approaches impose.

Literate programming and the warp approach described here explain programs to people who are interested in their algorithms. Future research should find better and more reliable ways of explaining programs to their non-specialist users, perhaps by automatically taking advantage of some comment scheme, and perhaps by generating reliable interactive help: then

programmers would become a bit closer to the explanatory needs of their programs' end users. In itself this might encourage programmers to make their programs easier to understand.

Knuth writes that "Science is what we understand well enough to explain to a computer" [13]; yes, and it progresses when those programs are explained reliably to the scientific community.

4. Acknowledgements

Paul Cairns, Matt Jones, Peter Ladkin, Gary Marsden and Russel Winder made constructive comments on this work.

REFERENCES

1. M. A. Addison & H. Thimbleby, "Intelligent Adaptive Assistance and Its Automatic Generation," *Interacting with Computers*, **8**(1):51–68, 1996.
2. K. Arnold & J. Gosling, *The Java^{TM} Programming Language Second Edition*, 2nd. ed., Addison-Wesley, 1998.
3. B. Braune & R. Wilhelm, "Focusing in Algorithm Explanation," *IEEE Transactions on Visualization and Computer Graphics*, **6**(1):1–7, 2000.
4. D. Deutsch, *The Fabric of Reality*, Penguin, 1997.
5. 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), 151–173, Springer, 1996.
6. D. R. Hanson with C. J. Van Wyk (moderator) & J. Gilbert (reviewer), "Literate Programming," *Communications of the ACM*, **30**(7):594–599, 1987.
7. D. E. Knuth, *The Stanford GraphBase*, Addison-Wesley, 1993.
8. D. E. Knuth & S. Levy, *The CWEB System of Structured Documentation, Version 3.0*, Addison-Wesley, 1994.
9. I. A. Ovsiannikov, M. A. Arbib & T. H. Mcneill, "Annotation Technology," *International Journal of Human-Computer Studies*, **50**(4):329–362, 2000.
10. 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.
11. P. B. Ladkin & H. Thimbleby, "From Logic to Manuals Again," *IEE Proceedings Software Engineering*, **144**(3):185–192, 1997.
12. Microsoft Corporation, *Getting Started: Microsoft Office*, Document No. OF64076-0295, 1992–1994.
13. M. Petkowvšek, H. S. Wilf & D. Zeilberger, $A = B$, A K Peters, 1996.
14. 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.
15. T. A. Phelps & R. Wilensky, "Multivalent Documents," *Communications of the ACM*, **43**(6):83–90, 2000.
16. C. Queinnec, *Literate Programming from Scheme to TEX*, Université Paris 6 & INRIA-Rocquencourt, 2000.
17. N. Ramsey, "Literate programming simplified," *IEEE Software*, **11**(5):97-105, 1994.
18. R. Sedgewick, *Algorithms*, Addison-Wesley, 1983.
19. H. W. Thimbleby, "Experiences with Literate Programming Using CWEB (A Variant of Knuth's WEB)," *Computer Journal*, **29**(3):201–211, 1986.
20. H. W. Thimbleby, "Java: A Critique," *Software—Practice & Experience*, **29**(5):457–478, 1999.

21. 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.
22. H. W. Thimbleby, "Calculators are Needlessly Bad," *International Journal of Human-Computer Studies*, **52**(6):1031–1069, 2000.
23. S. Wolfram, *The Mathematica Book*, 4th. ed., Addison-Wesley, 1999.