ProgDOC - a New Program Documentation System

Volker Simonis and Roland Weiss

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Sand 13, 72076 Tübingen, Germany
{simonis,weissr}@informatik.uni-tuebingen.de

Abstract. Though programming languages and programming styles evolve with remarkable speed today, there is no such evolution in the field of program documentation. And although there exist some popular approaches like Knuth's literate programming system WEB [26], and nowadays JavaDoc [15] or Doxygen [16], tools for managing software development **and** documentation are not as widespread as desirable.
This paper analyses a wide range of literate programming tools available during the past two decades and introduces ProgDOC, a new software documentation system. It is simple, language independent, and it keeps documentation and the documented software consistent. It uses LaTeX for typesetting purposes, supports syntax highlighting for various languages, and produces output in Postscript, PDF or HTML format.

1 Introduction

The philosophy of ProgDOC is to be as simple as possible and to pose as less requirements as possible to the programmer. Essentially, it works with any programming language and any development environment as long as the source code is accessible from files and the programming language offers a possibility for comments. It is non-intrusive in the sense that it leaves the source code untouched, with the only exception of introducing some comment lines at specific places.

The ProgDOC system consists of two parts. A so called *weaver* weaves the desired parts of the source code into the documentation, and a *highlighter* performs the syntax highlighting for that code. Source code and documentation are mutually independent (in particular they may be processed independently). They are linked together through special handles which are contained in the comment lines of the source code and may be referenced in the documentation.

ProgDOC is a good choice for writing articles, textbooks or technical white papers which contain source code examples and it proved especially useful for mixed language projects and for documenting already existing programs and libraries. Some examples of output produced by ProgDOC are available at [45].

The remainder of this paper is organized as follows: The first three sections will discuss some general aspects of literate programming, give a historical overview of the existing literate programming tools and present some new approaches for software documentation. In section 5, the ProgDOC system will be introduced and discussed in detail. Finally, section 6 will end the paper with conclusions and an outlook.

2 Some words on Literate Programming

With an article published 1984 in the Computer Journal [23] Donald Knuth coined the notion of "Literate Programming". Since those days for many people literate programming is irrevocable inter**weave**d with Knuth's WEB [26] and TEX [24] systems.

Knuth justifies the term "literate programming" in [23] with his belief that "... the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature." To support this programming style, he introduced the WEB system which is in fact both a language and a suite of utilities. In WEB, the program source code and the documentation are written together into one source file, delimited by special control sequences. The program source can be split into parts which can be presented in arbitrary order. The `tangle` program extracts these code parts from the WEB file and assembles them in the right order into a valid source file. Another program called `weave` combines the documentation parts of the WEB files with pretty printed versions of the code parts into a file which thereupon can be processed by TEX.

This system has many advantages. First of all, it fulfills the "one source" property. Because source code and documentation reside in one file, they are always consistent with each other. Second, the programmer is free to present the code he writes in arbitrary order, thus simplifying it for a human reader to understand the program. This can be done by rearranging code parts, but also by using macros inside the code parts, which can be defined later on in the WEB file. This way a top-down development approach is supported, in which the structure of a program as a whole is presented in the beginning and then subsequently refined, as well as a bottom up design, in which a program is assembled out of low level code fragments defined before. `tangle` will always expand these macros at the right place when constructing the source file out of the WEB file.

Another feature of the WEB system is the automatic construction of exhaustive indexes and cross references by `weave`. Every code part is accompanied by references which link it to all other parts which reference or use it. Also, an index of keywords with respect to code parts is created and the source code is pretty printed for the documentation part. The best way to convince yourself of WEB's capabilities is to have a look at Knuth's TEX implementation [25]. It was entirely written in WEB and is undoubtfully a masterpiece of publishing and literate programming.

2.1 WEB and its descendants

Besides its many advantages, the WEB system also has a couple of drawbacks. Many of them apply only to the original WEB implementation of Knuth and have been corrected or worked around in numerous WEB clones implemented thereafter. In this section we will present some of them[1] and discuss their enhancements.

One of the biggest disadvantages of WEB was the fact that it was closely tied to TEX as typesetting system and to Pascal as implementation language. So one of the first

[1] Only systems known to the authors will be mentioned here. A more complete overview may be found at the Comprehensive TEXArchive Network (CTAN) under http://www.ctan.org/tex-archive/web or at http://www.literateprogramming.org.

flavors of WEB was CWEB [27] which extended WEB to C/C++ as implementation languages. It was implemented by Knuth himself together with Silvio Levy. CWEBx [30] is an alternative CWEB implementation with some extensions by Marc van Leeuwen. They both suffer from the same problems like WEB, as they are closely coupled to TEX and the C programming language.

To overcome these language dependencies, noweb [39] (which evolved from spiderWEB) and nuweb [7] have been developed by Norman Ramsey and Preston Briggs, respectively. They are both language independent concerning the programming language, whereas they still use LATEX for typesetting. Nuweb is a rather minimalistic but fast WEB approach with only four control sequences. Both noweb and nuweb offer no pretty printing by default, but noweb is based on a system of tools called filters which are connected through pipes. The current version comes with pretty printing filters for C and Java (see the actual documentation).

Another descendant of an early version of CWEB is FWEB [29]. FWEB initially was an abbreviation for “Fortran WEB”, but meanwhile FWEB supports not only Fortran, but C, C++, Ratfor and TEX as well. These languages can be intermixed in one project, while FWEB still supports pretty printing for the different languages. On the other hand, FWEB is a rather complex piece of software with a 140 page user’s manual.

Ross Williams’ funnelWEB [53] is not only independent of the programming language, but of the typesetting language as well. It defines own format macros, which can be bound to arbitrary typesetting commands (currently for HTML and LATEX).

2.2 General drawbacks of WEB based literate programming tools

Though many of the initial problems of the WEB system have been solved in some of the clones, their sheer number indicates that none of them is perfect.

One of the most controversial topics in the field of literate programming is pretty printing where *pretty printing* stands for syntax highlighting[2] **and** code layout and indentation. There are two questions here to consider: Is pretty printing desirable at all, and if yes, how should the pretty printed code look like? The answer is often a matter of personal taste, however there also exist some research results in this area like for example [5].

From a practical point of view it must be stated that doing pretty printing is possible for Pascal, although a look at the WEB sources will tell you that it is not an easy task. Doing it for C is even harder[3]. Taking into account the fact that `weave` usually processes only a small piece of code, which itself even does not have to be syntactically correct, it should be clear that pretty printing such code in a complex language like for example C++ will be impossible.

To overcome these problems, special tags have been introduced by the various systems to support the pretty printing routines. But this clutters the program code in the WEB file and even increases the problem of the documentation looking completely

[2] *Syntax highlighting* denotes the process of graphically highlighting the tokens of a programming language.

[3] The biggest part of CWEB consists of the pretty printing module. Recognition of keywords, identifiers, comments, etc. is done by a hard coded shift/reduce bottom up parser.

different than the source. This can be annoying in a develop/run/debug cycle. As a consequence, the use of pretty printing is discouraged. The only feasible solution could be simple syntax highlighting instead of pretty printing, as it is done by many editors nowadays.

Even without pretty printing and additional tags inserted into the program source, the fact that the source code usually appears rearranged in the WEB file with respect to the generated source file makes it very hard to extend or debug such a program. A few lines of code laying closely together in the source file may be split up to completely different places in the WEB file.

Once this could be called a feature, because it gave the programmer new means of structuring his program code for languages like Pascal which offered no module system or object hierarchy. As analysed in [9] it could be used to achieve a certain amount of code and documentation reuse. However the WEB macro system could also be misused by defining and using macros instead of defining and using functions in the underlying programming language.

Another problem common to WEB systems is their "one source" policy. While this may help to hold source code and documentation consistent, it breaks many other development tools like debuggers, revision control systems and make utilities. Moreover, it is nearly impossible for a programmer not familiar with a special WEB system to debug, maintain or extend code devolved with that WEB.

Even the possibility of giving away only the tangled output of a WEB is not attractive. First of all, it is usually unreadable for humans[4], and second this would break the "one source" philosophy. It seems that most of the literate programming projects realized until now have been one man projects. There is only one paper from Ramsey and Marceau [38] which documents the use of literate programming tools in a team project. Additionally, some references can be found about the use of literate programming for educational purpose (see [8] and [44]).

The general impression confirms Van Wyk's observation in [60] "... that one must write one's own system before one can write a literate program, and that makes [him] wonder how widespread literate programming is or will ever become." The question he leaves to the reader is whether programmers are in general too individual to use somebody else's tools or if only individual programmers develop and use (their own) literate programming systems. The answer seems to lie somewhere in between. Programmers are usually very individual and conservative concerning their programming environment. There must be superior tools available to make them switch to a new environment.

On the other hand, integrated development environments (IDEs) evolved strongly during the last years and they now offer sophisticated navigation, syntax highlighting and online help capabilities for free, thus making many of the features of a WEB system, like indexing, cross referencing and pretty printing become obsolete (see section 3). Finally the will to write documentation in a formatting language like TEX using a simple text editor is constantly decreasing in the presence of WYSIWYG word processors.

[4] NuWEB is an exception here, since it forwards source code into the tangled output without changing its format.

2.3 Other program documentation systems

With the widespread use of Java a new program documentation system called JavaDoc was introduced. JavaDoc [15] comes with the Java development kit and is thus available for free to every Java programmer. The idea behind JavaDoc is quite different from that of WEB, though it is based on the "one source" paradigm as well. JavaDoc is a tool which extracts documentation from Java source files and produces formatted HTML output. Consequently, JavaDoc is tied to Java as programming and HTML as typesetting language[5]. By default JavaDoc parses Java source files and generates a document which contains the signatures of all public and protected classes, interfaces, methods, and fields. This documentation can be further extended by specially formatted comments which may even contain HTML tags.

Because JavaDoc is available only for Java, Roland Wunderling and Malte Zöckler created DOC++ [59], a tool similar to JavaDoc but for C++ as programming language. Additionally to HTML, DOC++ can create LaTeX formatted documentation as well. Doxygen [16] by Dimitri van Heesch, which was initially inspired by DOC++, is currently the most ambitious tool of this type which can also produce output in RTF, PDF and Unix man-page format. Both DOC++ and Doxygen can create a variety of dependency-, call-, inclusion- and inheritance graphs, which may be included into the documentation.

These new documentation tools are mainly useful for creating hierarchical, browesable HTML documentations of class libraries and APIs. They are intended for interface descriptions rather than the description of algorithms or implementation details. Although some of them support LaTeX, RTF or PDF output, they are not well suited for generating printed documentation.

Another approach which must be mentioned in this chapter is Martin Knasmüller's "Reverse Literate Programming" system [22]. In fact it is an editor which supports folding and so called *active text elements* [34]. Active text elements may contain arbitrary documentation, but also figures, links or popup buttons. All the *active text* is ignored by the compiler, so no tangle step is needed before compilation. Reverse Literate programming has been implemented for the Oberon system [54].

The GRASP [18] system relies on source code diagramming and source code folding techniques in order to present a more comprehensible picture of the source code, however without special support for program documentation or literate programming. In GRASP, code folding may be done according to the programming language control structure boundaries as well as for arbitrary, user-selected code parts.

3 Software documentation in the age of IDEs

Nowadays, most software development is done with the help of sophisticated IDEs (Integrated Development Environments) like Microsoft Visual Studio [32], IBM Visual Age [19], Borland JBuilder [6], NetBeans [35] or Source Navigator [40] to name just

[5] Starting with Java 1.2, JavaDoc may be extended with so called "Doclets", which allow JavaDoc to produce output in different formats. Currently there are Doclets available for the MIF, RTF and LaTeX format (see [49]).

a few of them. These development environments organize the programming tasks in so called projects, which contain all the source files, resources and libraries necessary to build such a project.

One of the main features of these IDEs is their ability to parse all the files which belong to a project and build a database out of that information. Because the files of the project can be usually modified only through the builtin editor, the IDEs can always keep track of changes in the source files and update the project database on the fly.

With the help of the project database, the IDEs can offer a lot of services to the user like fast, qualified searching or dependency-, call-, and inheritance graphs. They allow fast browsing of methods and classes and direct access from variables, method calls or class instantiations to their definitions, respectively. Notice that all these features are available online during the work on a project, in contrast to the tools like JavaDoc or Doxygen mentioned in the previous section which provide this information only off-line.

The new IDEs now deliver under such fancy names like "Code Completion" or "Code Insight" features like syntax directed programming [20] or template based programming which have been proposed already in the late seventies by [50,33]. In the past, these systems couldn't succeed because of two main reasons: they where to restrictive in the burden they put on the programmer and the display technology and computing power have not been good enough[6]. However, the enhancements in the area of user interfaces and the computational power available today allow even more: context sensitive prompting of the user with the names of available methods or with the formal arguments of a method, syntax highlighting and fast recompilation of affected source code parts.

All this reduces the benefits of a printed, highly linked and indexed documentation of a whole project. What is needed instead, additionally to the interface description provided by the IDE, is a description of the algorithms and of certain complex code parts. One step into this direction was Sametinger's DOgMA [41,42] tool which is an IDE that also allows writing documentation. DOgMA, like modern IDEs today, maintains an internal database of the whole parsed project. It allows the programmer to reference arbitrary parts of the source code in the documentation while DOgMA automatically creates and keeps the relevant links between the source code parts and the documentation up to date. These links allow a hypertext like navigation between source code and documentation.

While it seems that modern IDEs adopted a lot of DOgMA's browsing capabilities, they didn't adopted its literate programming features. However, systems like NetBeans [35], SourceNavigator [40] or VisualAge [48]) offer an API for accessing the internal program database. This at least would allow one to create extensions of these systems in order to support program documentation in a more comfortable way.

The most ambitious project in this context in the last few years was certainly the "Intentional Programming" project lead by Charles Simonyi [46,47] at Microsoft. It revitalized the idea of structured programming and propagated the idea of programs being just instantiations of intentions. The intentions could be written with a fully fledged

[6] A good survey about the editor technology available at the beginning of the eighties can be found in [31].

WYSIWYG editor which allowed arbitrary content to be associated with the source code. Of course, this makes it easy to combine and maintain software together with the appropriate documentation. Some screen-shots of this impressive system can be found in chapter 11 of [11], which is dedicated solely to Intentional Programming.

4 Software documentation and XML

With the widespread use of XML [57] in the last few years it is not surprising that various XML formats have been proposed to break out of the "ASCII Straitjacket" [1] in which programming languages are caught until now. While earlier approaches to widen the character set out of which programs are composed like [1] failed mainly because of the lack of standards in this area, the standardization of UNICODE [51] and XML may change the situation now.

There exist two concurring approaches. While for example JavaML [4] tries to define an abstract syntax tree representation of the Java language in XML (which, by the way, is not dissimilar from the internal representation proposed by the early syntax directed editors) the CSF [43] approach tries to define an abstract XML format usable by most of the current programming languages. Both have advantages as well as disadvantages. While the first one suffers from it's dependency on a certain programming language, the second one will always fail to represent every exotic feature of every given programming language.

A third, minimalistic approach could ignore the syntax of the programming language and just store program lines and comments into as few as two different XML elements. Such an encoding has been proposed by E. Armstrong [3].

However, independent of the encoding's actual representation, once that such an encoding would be available, literate programming and program documentation systems could greatly benefit from it. They could reference distinct parts of a source file in a standard way or they could insert special attributes or even elements into the XML document which could be otherwise ignored by other tools like compilers or build systems. Standard tools could be used to process, edit and display the source files, and internal as well as external links could be added to the source code.

Peter Pierrou presented in [37] an XML literate programming system. In fact it consists of an XML editor which allows one to store source code, documentation and links between them into an XML file. A tangle script is used to extract the source code out of the XML file. The system is very similar to the reverse literate programming tool proposed by Knasmüller, with the only difference that it is independent of the source language and stores its data in XML format. An earlier, but very similar effort described in [14] used SGML as markup language for storing documentation and source code.

Anthony Coates introduced xmLP [10], a literate programming system which uses some simple XML elements as markup. The idea is to use these elements together with other markup elements, for example those defined in XHTML [56], MathML [55] or DocBook [52]. XSLT [58] stylesheets are then used in order to produce the woven documentation and the tangled output files.

Oleg Kiselyov suggested the representation of XML as an s-expression in Scheme called SXML [21]. SXML can be used to write literate XML programs. Different

Scheme programs (also called stylesheets in this case) are available to convert from SXML to LaTeX, HTML or pure XML files.

Some of the approaches presented in this section are quite new, but the wide acceptance of XML also in the area of the source code representation of programming languages could give new impulses to the literate programming community. A good starting point for more information on literate programming and XML is the Web site of the OASIS consortium, which hosts a page specifically dedicated to this topic [36].

5 Overview of the ProgDOC system

With this historical background in mind, ProgDOC tries to combine the best of the traditional WEB and the new program documentation systems. It releases the "one source" policy, which was so crucial for all WEB systems, thus giving the programmer maximum freedom to arrange his source files in any desirable way. On the other hand, the consistency between source code and documentation is preserved by special handles, which are present in the source files as ordinary comments[7] and which can be referenced in the documentation. ProgDOC's weave utility `pdweave` incorporates the desired code parts into the documentation.

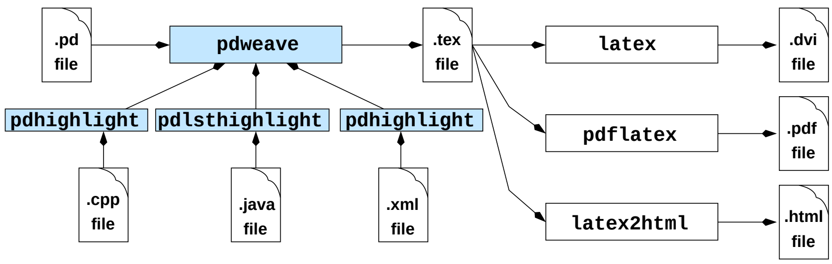

Fig. 1. Overview of the ProgDOC system.

But let us start with an example. Suppose we have a C++ header file called `ClassDefs.h` with some class declarations. Following this section you can see a verbatim copy of the file:

```
class Example1 {
private :
  int x;
public :
  explicit Example1(int i) : x(i) {}
};

class Example2 {
private :
  double y;
public :
  explicit Example2(double d) : y(d) {}
  explicit Example2(int i) : y(i) {}
  explicit Example2(long i) : y(l) {}
  explicit Example2(char c) : y((int)c) {}
  void doSomething(); // do something
};
```

[7] As far as known to the authors, any computer language offers comments, so this seems to be no real limitation.

With ProgDOC, the class may be written as follows:

```
// BEGIN Example1
class Example1 {
private :
  int x;
public :
  explicit Example1(int i) : x(i) {}
};
// END Example1

// BEGIN Example2
class Example2 {
// ...
private :
  double y;
// ...
public :
  // BEGIN Constructors
  explicit Example1(double d) : y(d) {}
  explicit Example2(int i) : y(i) {}
  explicit Example2(long i) : y(l) {}
  explicit Example2(char c) : y((int)c) {}
  // END Constructors
  void doSomething(); // do something
};
// END Example2
```

The only changes introduced so far are the comments at the beginning and at the end of each class declaration. These comments, which of course are non-effective for the source code, enable us to use the new `\sourceinput[`*options*`]{`*filename*`}{`*tagname*`}` command in the LaTeX documentation. This will result in the inclusion and syntax highlighting of the source code lines which are enclosed by the "`// BEGIN` *tagname*" and "`// END` *tagname*" lines, respectively.

Consequently the LaTeX code presented in the following box

```
``... next we present the declaration
of the class {\tt Example1}:
\sourceinput[fontname=pcr, fontsize=7,
  listing, linenr, label=Example1]
  {ClassDefs.h}{Example1}
as you can see, there is no magic at
all using the {\tt \symbol{92}
sourceinput} command ...''
```

will result in content of the box shown on the right side:

"... next we present the declaration of the class `Example1`:

Listing 1: ClassDefs.h [Line 2 to 7]

```
class Example1 {
private :
  int x;
public :
  explicit Example1(int i) : x(i) {}
};
```

as you can see, there is no magic at all using the `\sourceinput` command ..."

The source code appears nicely highlighted, while its indentation is preserved. It is preceded by a caption line similar to the one known from figures and tables which in addition to a running number also contains the file name and the line numbers of the included code. Furthermore, the code sequence can be referenced everywhere in the text with a usual `\ref` command (e.g. see Listing 1). Notice that the boxes shown here are used for demonstrational purpose and are not produced by the ProgDOC system.

As shown in Figure 1, ProgDOC isn't implemented in pure LaTeX. Instead, the weaver component `pdweave` is an AWK [2] script while the syntax highlighter `pdhighlight` is a program generated with `flex` [13]. It was originally based on a version of Norbert Kiesel's `c++2latex` filter. It not only marks up the source code parts for LaTeX, but also inserts special HTML markup into the LaTeX code it produces. In that way an HTML-version of the documentation may be created with the help of Nikos Drakos' and Ross Moore's `latex2html` [12] utility. However, `pdweave` is not restricted on `pdhighlight` as highlighter. It may use arbitrary highlighters which conform to the interface expected by the weaver. And indeed, ProgDOC provides a second highlighter, called `pdlsthighlight`, which is in fact just a wrapper for the LaTeX listings package [17].

Listings 2 and 3 demonstrate some other features of ProgDOC like displaying nested code sequences, hiding of code parts which can be thought of as a kind of code folding [22,18] and linking these parts together by either references or active links in HTML/PDF output. Furthermore ProgDOC is highly customizable. See the ProgDOC manual [45] for a complete reference.

Listing 2: ClassDefs.h [Line 11 to 24]

```
class Example2 {
...
public :
  <see Listing 3 on page 447>
  void doSomething(); // something
};
```

Listing 3: ClassDefs.h [Line 18 to 21]
(Referenced in Listing 2 on page 447)

```
explicit Example2(double d) : y(d) {}
explicit Example2(int i) : y(i) {}
explicit Example2(long l) : y(l) {}
explicit Example2(char c) : y((int)c) {}
```

6 Conclusions

This paper listed and discussed most of the literate programming and program documentation systems which have been proposed during the past 20 years and it introduced ProgDOC, the authors' own program documentation system. Even though ProgDOC is a fully functioning system it also suffers from one drawback criticized in some of the other systems: its tight coupling with LaTeX as typesetting system. But ProgDOC should be thought of as just one incarnation of a very simple, yet powerful idea of keeping source code and documentation synchronised by connecting them through links. The authors' opinion is that nowadays programs are best written with sophisticated IDEs and documentation is best written with powerful word processors, where both of these tools are best suited for their own specific task.

However, importing parts of the source code into the documentation should be just as easy as the import of tables or figures. With today's technology this is only possible with special tools, like ProgDOC. But it is not hard to imagine that one day IDEs will export source code just the way spreadsheet programs export tables. The emerging use of the XML technology may be helpful for this purpose.

7 Acknowledgements

We want to thank all the users who used ProgDOC and supplied feedback information to us. Among others these are Martin Gasbichler, Blair Hall and Patrick Crosby. We are truly indebted to Holger Gast, who always answered patiently all our questions and solved many of our problems concerning TeX.

References

1. P. W. Abrahams. *Typographical Extensions for Programming Languages: Breaking out of the ASCII Straitjacket.* ACM SIGPLAN Notices, Vol. 28, No. 2, Feb. 1993
2. A.W. Aho, B.W. Kernighan and P. J.Weinberger. *The AWK Programming Language.* Addison-Wesley, 1988
3. E. Armstrong. *Encoding Source in XML - A strategig Analysis.* http://www.treelight.com/software/encodingSource.html
4. G. J. Badros. *JavaML: A Markup Language for Java Source Code.* 9th Int. WWW-Conference, Amsterdam, May 2000
5. Ronald M. Baecker, Aaron Marcus. *Human Factors and Typography for More Readable Programs.* Addison-Wesley, 1990
6. Borland Software Corporation. *Borland JBuilder.* http://www.borland.com/jbuilder
7. Preston Briggs. *nuWeb*, http://ctan.tug.org/tex-archive/web/nuweb
8. Bart Childs *Literate Programming, A Practitioner's View* TUGboat, Volume 13, No. 2, 1992, http://www.literateprogramming.com/farticles.html
9. B. Childs and J. Sametinger. *Analysis of Literate Programs from the Viewpoint of Reuse.* Software - Concepts and Tools, Vol. 18, No. 2, 1997, http://www.literateprogramming.com/farticles.html
10. A. B. Coates and Z. Rendon *xmLP - a Literate Programming Tool for XML & Text.* Extreme Markup Languages, Montreal, Quebec, Canada, August 2002, http://xmlp.sourceforge.net/2002/extreme/
11. K. Czarnecki and U. W. Eisenecker. *Generative Programming.* Addison-Wesley, 2000
12. by Nikos Drakos and Ross Moore. *Latex2HTML.* http://saftsack.fs.uni-bayreuth.de/˜latex2ht/ or: http://ctan.tug.org/ctan/tex-archive/support/latex2html
13. Free Software Foundation *The Fast Lexical Analyzer.* http://www.gnu.org/software/flex/
14. D.M. German, D.D. Cowan and A. Ryman. *SGML-Lite – An SGML-based Programming Environment for Literate Programming.* ISACC, Oct. 1996, http://www.oasis-open.org/cover/germanisacc96-ps.gz
15. J. Gosling, B. Joy and G. Steele *"Java Language Specification"* Addison-Wesley, 1996
16. Dimitri van Heesch. *Doxygen.* http://www.doxygen.org
17. Carsten Heinz *"The Listings package"*, ftp://ftp.dante.de/tex-archive/help/Catalogue/entries/listings.html
18. T. D. Hendrix, J. H. Cross II, L. A. Barowski and K. S. Mathias. *Visual Support for Incremental Abstraction and Refinement in Ada95.* SIGAda Ada Letters, Vol. 18, No. 6, 1998
19. IBM Corporation. *Visual Age C++.* http://www-3.ibm.com/software/ad/vacpp
20. A. A. Khwaja and J. E. Urban. *Syntax-Directed Editing Environments: Issues and Features.* ACM SIGAPP Symposium on Applied Computing, Indianapolis, Indiana, 1993
21. O. Kiselyov. *SXML Specification.* ACM SIGPLAN Notices, Volume 37, Issue 6, June 2002 http://pobox.com/˜oleg/ftp/Scheme/xml.html
22. M. Knasmüller. *Reverse Literate Programming.* Proc. of the 5th Software Quality Conference, Dundee, July 1996
23. Donald E. Knuth *Literate Programming* The Computer Journal, Vol. 27, No. 2, 1984
24. Donald E. Knuth *The TEXbook* Addison-Wesley, Reading, Mass., 11. ed., 1991
25. Donald E. Knuth *TEX: The Program* Addison-Wesley, Reading, Mass., 4. ed., 1991
26. Donald E. Knuth *Literate Programming* CSLI Lecture Notes, no. 27, 1992 or Cambridge University Press
27. Donald. E. Knuth and Silvio Levy *The CWEB System of Structured Documentation* Addison-Wesley, Reading, Mass., 1993
28. Uwe Kreppel. *WebWeb.* http://www-ca.informatik.uni-tuebingen.de/people/kreppel/

29. John Krommes. *fWeb.* http://w3.pppl.gov/˜krommes/fweb.html
30. Marc van Leeuwen. *CWebx.* http://wallis.univ-poitiers.fr/˜maavl/CWEBx/
31. N. Meyrowitz and A. van Dam. *Interactive Editing Systems: Part I and II.* Computing Surveys, Vol. 14, No. 3, Sept. 1982
32. Microsoft Corporation. *Visual Studio.* http://msdn.microsoft.com/vstudio
33. J. Morris and M. Schwartz. *The Design of a Language- Directed Editor for Block-Structured Languages.* SIGLAN/SIGOA Symp. on text manipulation, Portland, 1981
34. H. Mössenböck and K. Koskimies. *Active Text for Structuring and Understanding Source Code.* Software - Practice and Experience, Vol. 27, No. 7, July 1996
35. NetBeans Project. *The NetBeans Platform and IDE.* http://www.netbeans.org
36. The Oasis Consortium. *SGML/XML and Literate Programming.* http://www.oasis-open.org/cover/xmlLitProg.html
37. P. Pierrou. *Literate Programming in XML.* Markup Technologies, Philadelphia, Pensylvania, US, Dec. 1999, http://www.literateprogramming.com/farticles.html
38. N. Ramsey and C. Marceau *Literate Programming on a Team Project* Software - Practice & Experience, 21(7), Jul. 1991, http://www.literateprogramming.com/farticles.html
39. Norman Ramsey *Literate Programming Simplified* IEEE Software, Sep. 1994, p. 97 http://www.eecs.harvard.edu/˜nr/noweb/intro.html
40. Red Hat, Inc. *Source Navigator.* http://sourcenav.sourceforge.net
41. J. Samtinger *DOgMA: A Tool for the Documentation & Maintenance of Software Systems.* Tech. Report, 1991, Inst. fur Wirtschaftsinformatik, J. Kepler Univ., Linz, Austria
42. J. Samtinger and G. Pomberger *A Hypertext System for Literate C++ Programming.* JOOP, Vol. 4, No. 8, SIGS Publications, New York, 1992
43. S. E. Sandø, The Software Development Foundation *CSF Specification.* http://sds.sourceforge.net
44. Stephan Shum and Curtis Cook *Using Literate Programming to Teach Good Programming Practices* 25th. SIGCSE Symp. on Computer Science Education, 1994, p. 66-70
45. Volker Simonis *The* PROGDOC *Program Documentation System* http://www.progdoc.org
46. C. Simonyi. *Intentional Programming - Innovation in the Legacy Age.* IFIP WG 2.1 meeting, june 4th, 1996
47. C. Simonyi. *The future is intentional.* IEEE Computer Magazine, Vol. 32, No. 5, May 1999
48. D. Soroker, M. Karasick, J. Barton and D. Streeter. *Extension Mechanisms in Montana.* Proc. of the 8th Israeli Conf. on Computer Based Systems and Software Engineering, 1997
49. Sun Microsystems, Inc. *The Doclets API.* http://java.sun.com/j2se/javadoc/
50. T. Teitelbaum and T. Reps. *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.* Communications of the ACM, Vol. 24, No. 9, Sept. 1981
51. The Unicode Consortium. *The Unicode Standard 3.0.* Addison-Wesley, Reading, Mass., 2000, http://www.unicode.org/
52. N. Walsh and L. Muellner. *DocBook: The Definitive Guide.* O Reilly & Associates, 1999, http://www.oasis-open.org/committe/docbook
53. Ross N. Williams. *funnelWeb.* http://www.ross.net/funnelweb/
54. N. Wirth and J. Gutknecht. *The Oberon System.* Software - Practice & Experience, 19(9), 1989, p. 857-893
55. World Wide Web Consortium. *Mathematical Markup Language.* http://www.w3.org/Math
56. WorldWideWeb Consortium. *Extensible Hypertext Markup Language.* http://www.w3.org/MarkUp
57. The World Wide Web Consortium. *Extensible Markup Language.* http://www.w3.org/XML
58. World Wide Web Consortium. *Extensible Stylesheet Language Transformations.* http://www.w3.org/Style/XSL
59. R. Wunderling and M. Zöckler. *DOC++.* http://www.zib.de/Visual/software/doc++/
60. Christopher J. Van Wyk *Literate Programming Column.* Communications of the ACM, Volume 33, Nr. 3, March 1990. p. 361-362