# Literate Programming in XML

## Combining source code and documentation

*Author:*Peter Pierrou*Address:*Excosoft AB, Peter.Pierrou@excosoft.se
http://www.excosoft.se

Peter Pierrou is the Project Manager for the Documentor project at Excosoft.

### Abstract

This paper introduces a programming environment which is based on hierarchical structure, links, and literate programming. This programming environment aims to increase productivity in large software construction projects. Literate programming in XML is good for productivity as it makes source code readable, provides an overview, and helps keep documentation consistent with the source code.

### Background

New technologies and standards have recently emerged that allow the creation of an innovative programming environment. A programming environment where the primary focus is not on producing code in a particular programming language, but on the structure and logic of the program. Accordingly I advance a new approach to program development, combining the link technology popularized by HTML and the hierarchical structure of XML with the most important aspects of Donald E. Knuth's literate programming.

In this paper I describe a literate programming system that enables the programmer to work directly with documentation and pseudo code. The system consists of an XML editor with a programming environment that supports literate programming.

The key feature of the system is that the program is readable on the screen. It differs, therefore, from those literate programming systems that focus on transforming something that is totally unreadable on the screen into a readable printed format. Such an approach does not help the programmer and it is a criticism of current literate programming systems that they only confer long-term advantages and are not applicable to projects that operate in short time frames [CL]. A literate programming system should help the programmer to program.

The system presented here has more in common with a system devised by Markus Knasmüller [MK], which he has described as 'reverse literate programming'. The idea behind reverse literate programming is that you do not transform what is on the screen into documented source code. What is on the screen is the documented source code!

Why use XML? A literate programming system based on XML has several immediate advantages: XML is a standard format; XML has powerful linking properties; there are already a large number of tools that support XML (parsers, editors, etc.); there are already XML DTDs that can handle tables, mathematical formulae, etc.; most XML editors have a validating function that can compel users to write pseudo code and/or documentation.

The availability of a styling language such as XSL [XSL] that can format XML documents for printing is not of interest here. How the printed program looks is unimportant. It is what is displayed on the screen that matters. A printout pales in comparison to an on-screen presentation of the program that combines a hierarchical structure with links that help you follow the flow of the program.

However, the programmer will not benefit from using XML if the code is written in an ordinary text editor. XML code is as unreadable on the screen for the programmer as the current literate programming tools. What is needed is an XML editor with a programming environment that supports literate programming, or perhaps more accurately supports reverse literate programming.

What I describe here is how such an editor should function.

## The Thesis

## The Bottleneck in Software Construction

Most large program construction projects have an efficiency problem. The root of this problem has been identified by the proponents of literate programming as insufficient understanding of previously written code. This is especially a problem in environments where there are many files which relate to one another in various ways. For example, where many of the files are large and contain information which is hard to comprehend; where the files are of different kinds; and finally, where there are many people involved – and consequently where the people involved tend to come and go. In short, this is a problem for any professional programming enterprise.

This problem shows itself in a number ways. A disproportionate share of the development time –some say 90 percent –is spent on code maintenance, such as documenting and debugging, leaving only a little time for the actual development. A lot of the spent writing new code could be gained if old software could be more efficiently reused. Everyone who has worked in a project knows how frustrating it can be to read the code of other programmers; in fact, many programs are more difficult to read than to write. As a result the degree of reuse is considerably lower than it could be. In addition, old code is often discarded because it is easier to rewrite it, especially if the original programmer has left the project.

## The Efficient Programming Environment

The answer to this problem is literate programming in XML. The key innovation of literate programming is that source code and documentation are combined and stored together, ensuring that the documentation is up to date, and above all making the program easier to understand. When you take the code and the documentation together, store them in an XML file, structured according to the internal logic of the task that the code performs, and add links wherever they contribute to understanding, you get a program that is not only readable, but also surveyable. An understandable and surveyable program can be grasped in a very short time, which should be the most important concern in any large project.

## Readable Documents

## The Importance of Readability

At Excosoft, we have a set of programming rules. They are very general, and include factors such as stability, simplicity, and reusability. The foremost rule, however, is that the code must be readable. By that we mean not only that it must be well documented and easy to understand, but that it must also be structured in such a way that it is obvious what problem each part of the code is there to solve.

By strictly adhering to this way of writing source code, we have achieved a fertile environment with minimal overheads for maintaining and interpreting code. Our project is too large for any one person to be familiar with all of the code in detail (around 290 000 lines of code), so all of us reap the benefits of readability. A testament to the success of this approach to documentation and structuring is that new recruits can usually start working productively in the team after a learning period of only one to two weeks.

Another advantage of readable code is that you are not restricted to a development model in which each programmer is responsible for their own code through the entire lifetime of the code. As the time needed to understand new code is greatly reduced, in principle the people working on a certain detail can be replaced overnight, without stopping development.

Furthermore, readability is a precondition for all the other qualities that are desirable in the code. Incomprehensible code is almost a guarantee that errors will occur sooner or later. It is usually unstable –often nobody really knows for sure whether all the exceptional cases are handled correctly. And as I have mentioned, it is rarely reused: a waste of resources.

### Hierarchical Structure

The main advantage of programming in XML is that you can structure the program hierarchically. Documents can be divided into block elements, which can contain both source code and documentation, and whenever needed the information in a block can be further divided into smaller blocks. The people writing the code freely decide how to make these divisions. A powerful feature of this structure is that pseudo code is tied to each block. This introduces a new level of documentation: the logic of the program spelled out in words that have no syntactical restrictions.

```
Public COMPILED_XPOINTER::Execute()
int
COMPILED_XPOINTER::Execute(
   TREE_T *root_tree,    // in: the root of the file
   TREE_T *origin_tree,  // in: the tree the link goes from
   TREE_T *context_tree, // in: relative location for the XPTERMs
   NodeList &result_list // out: list of matching entry points
)
{
   Declarations
   while ((xpt=(XPTERM*)exec_it.Next())) {
      Call execute for each XPTERM in list
      if (xpt->Execute( root_tree, origin_tree, current_context, result_list) != OK) {
         m_ErrorTerm = xpt;   // save to ErrorMess function
         break;
      }
      Set up current context
   }
   return (result_list.getSize() ? OK : NOT_OK);
}
```

Figure 1: Pseudo code explains the program.

Most work takes place on this level. Only the pseudo code is visible until the user decides to open the block that it describes. An important aspect, therefore, of programs written in XML is the overview they provide. The general content of a large document can be grasped at a glance, because the detail are suppressed.

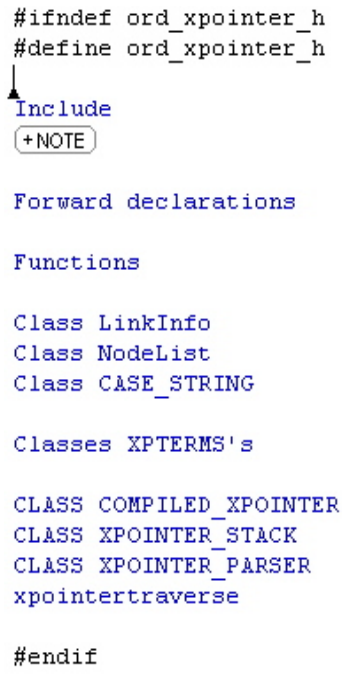



Figure 2: Hierarchical structure provides overview.

Even with small amounts of information that are relatively straightforward and understandable, the time that a structure like this can save is considerable. It might take minutes to read and understand 20 lines of code, whereas 2 lines of pseudo code describing those 20 lines may take only a second to understand.

```
int                        // returns: status
xpointertraverse(
   const JSTRING &cmd,     // in:  xpointer command language
   TREE_T *start_tree,     // in:  see note
   TREE_T *link_tree,      // in:  the tree the link goes from
   int case_sensitive,     // in:  tags and attribute names case sensitive?
   NodeList &out_list      // out: List of pointers to the trees resolved by the parser
)
{
   XPOINTER_PARSER parser(case_sensitive);
   COMPILED_XPOINTER *lang_tree = parser.Parse(cmd);

   if (!lang_tree) {   // parse error
      JSTRING mess;
      parser.ErrorMess( mess );
      PSTRING pars_failed("Parse of xpointer failed");
      PSTRING erro( mess );
      mess_Message( pars_failed, ERRO );
      mess_Message( erro, INFO);
      return NOT_OK;
   }
   int status = lang_tree->Execute( start_tree, link_tree, start_tree, out_list );
   if (status != OK) { // execution error
      JSTRING mess;
      lang_tree->ErrorMess( mess );
      PSTRING execute("Execute of xpointer failed");
      PSTRING erro(mess);
      mess_Message( execute, INFO);
      mess_Message( erro, INFO);
   }
   delete lang_tree;
   return status;
}
```

Figure 3: 20 lines of source code...

```
int                        // returns: status
xpointertraverse(
   const JSTRING &cmd,     // in:  xpointer command language
   TREE_T *start_tree,     // in:  see note
   TREE_T *link_tree,      // in:  the tree the link goes from
   NodeList &out_list      // out: List of pointers to the trees resolved by the parser
)
{
   Parse xpointer
   Execute xpointer
}
```

Figure 4: ... or 2 lines of pseudo code

Still, the details are not unimportant; after all, that is where the code is found. The block solution does not make the source code less accessible. On the contrary, when a block is opened it is presented together with the corresponding pseudo code in its context in the document. Blocks within blocks can be opened in the same way. Usually each block contains no more than a dozen lines, which can contain either source code or pseudo code, so each block can be easily understood when it is opened. Thanks to the pseudo code, it is easy to walk through the structure to find the desired information.

Working with documents, users continually change the focus of their attention. A block based structure allows the granularity of presentation to be adjusted to meet the needs of the users. Unopened blocks are presented on the screen as pseudo code lines, whereas the next level of details is presented as opened blocks. The code that is being examined is viewed in a more comprehensible context, presented at a more general level. The function of the code is, therefore, easier to understand.

The program is always presented according to its structure –when editing as well as browsing. When code is written, it is possible to work in two ways. Either top-down, which means writing the pseudo code first and then filling in the code, or bottom-up, writing the code first and then adding structure and pseudo code. Entire blocks are edited as if they were just one line –the way they are presented. Thus, logical units can be moved or copied in a single operation.

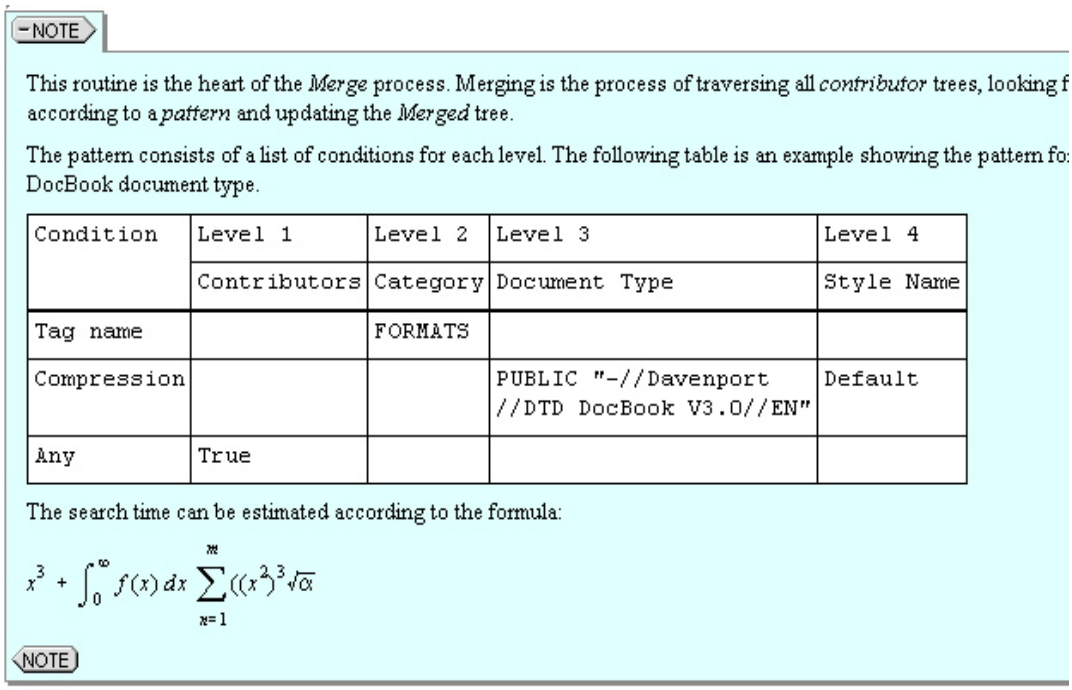
## Literate Programming

Literate programming is sometimes described as writing programs that can be read like books, linearly rather than jumping between documents. As I have shown, you can use XML to go a step further: from a linear to a hierarchical structure.

What I think is important about literate programming is that program files are complete documents. This means that they contain all the information associated with the task that they perform; or in other words, they combine source code and documentation. In XML, this is accomplished by dividing the document into sections.

Anywhere in the source code, "note sections" can be inserted. A note section contains documentation and will not be treated as source code.

This division provides an easy way to include any type of information in the source file. For instance, in a note section it is possible to use different fonts and styles to improve readability; or bulleted lists, mathematical formulae, diagrams, tables, or anything else that can help explain the code. There are no restrictions on the content of a note section: only the people writing the documents know what is required to provide the best possible documentation.




```
void
MERGER::MergeTrees(
   tree_ot top_source,     // In what tree to start search
   tree_ot source_folder,  // Where to look for already read files
   tree_ot top_target,     // Where to put merged result
   tree_ot patt_lst,       // List of match patterns for each level
   int merge_level,        // Stop merging at this level
   int stop_level,         // Stop collecting at this level
   int stop_when_found,    // Stop after first total merge
   MERGE_ADAPT *adapt      // User adaption at merging
)

{
   Check any source
   Init
   Merge
}
```

Figure 5: Note sections contain any type of information.

### Links

Another way to make a program easier to understand is to use links. Links should always be added to a document when they make it easier to follow the program. Links can be used in many ways. For example, there can be links from a place where a program function is called to the definition of the function; or in object oriented languages from the instantiation of a class to the class definition. In these cases the reader of the program can easily follow the flow of the program, even when it moves between different documents.

```
Parse xpointer
XPOINTER_PARSER parser(case_sensitive);
COMPILED_XPOINTER *lang_tree = parser.Parse(cmd);
```

Figure 6: Links can visualize the program flow.

Links can go to anywhere in any file. They can use the HTTP protocol and they can go to files that are version managed so that they always point at the current version. They can go to whole documents or to parts of documents. Except for language specific information like functions and classes, it might be useful to have links to requirement specifications, customer documentation, test documents, error reports, and so on.

Links are potentially powerful, but that potential is only partially realized in systems that do not support hierarchical programming. When trying to read a program, it would be unhelpful to be transported to a new context every time a link is opened. A link is more than just a way to get from A to B, it describes a meaningful relationship. That relationship is visualized by opening the target of the link in the context of the starting point. Thinking of links in such a way is natural in a hierarchical environment. When a link is opened the content of the link target is presented in a block connected to the linking position.

```
Parse xpointer
XPOINTER_PARSER parser(case_sensitive);
COMPILED_XPOINTER *lang_tree = parser.Parse(cmd);
COMPILED_XPOINTER *XPOINTER_PARSER::Parse(
   const JSTRING &instr
)
{
   StartUp parser
   Parse
}
```

Figure 7: Links open in context.

### What about Class Browsers?

One objection to all this is that so-called  class browsers can be used instead. There are two answers to this objection.

The first answer is that class browsers do not address the same problem. They are excellent for analyzing the structure of object oriented source code. But the structure that they present is only the structure of the programming language. While a literate program with a hierarchical structure and strategical links offers the same overview of classes and other language dependent information, that is not its main function. The important structure in a program is completely language independent. This is the structure that a literate program can visualize.

The second answer is that the structure and links of a literate program are always there. No code analysis or special tools are needed. The program is always viewed in the same XML-based program editor, whether the task at hand involves writing new code, modifying old code, or simply browsing. Structure is a natural part of the program, not something outside of the program.

## Intelligent Documents

### Functional Information

XML program documents are intelligent; marked up with tags so that it is possible to distinguish between different kinds of content, and with attributes to provide information about the status of the content as well as other special features.

My opinion is that tags for syntactical structures of a programming language are difficult to maintain and do more harm than good. The information is already present in the source code itself and it is pointless to duplicate such structures and try to keep the copy consistent with the original. Five element types are enough to describe the logical parts of an XML literate program: these are types for source code, pseudo code, interface descriptions, documentation, and metadata.

### Additional Code Markup

In our environment we also define various attributes that help the dynamic process of development. Working with information that changes all the time, we need to keep track of the changes. A version management tool is necessary, but insufficient. Comparing different versions of a document to see what has changed is too time consuming. Therefore the document gets marked up with revision marks, stored in attributes. In each new version of a document, it is easy to see and search for modifications made since the previous version.

We also use an attribute that we call hidden. Source code that is marked as hidden is excluded from the pure source code file that is generated from the document, so that it is not executed or compiled. Hidden attributes are inherited; if an element is hidden all contained elements are also hidden. One pseudo code line –i.e. one block –can be marked as hidden, so that all the source code that the block contains will be excluded.

This has proved useful in two cases. Firstly as a way to experiment with different implementations. If there are two ways to do something, then both implementations can be written and stored next to one another in the file, in two different blocks. One of the blocks is hidden, only the other one will be used. All one needs to do to change implementation is to move the hidden attribute. Secondly, when code changes we often keep the old code in the new version of the document, marked as hidden. This makes it easy to see what changes have been made, and to fix problems that might arise because of the modifications.


Figure 8: Hidden text is presented with gray background; the lines to the left are revision marks.

Another attribute we sometimes use is for conditional formatting. Like many large software projects we need to maintain different systems based on the same code, this is especially important as we deliver our programs for several platforms. My experience is that it is usually better to keep the different implementations for different systems in separate files. Still, sometimes it is more practical to solve the problem by keeping the various implementations in the same document, next to one another. Then conditional formatting solves the problem of choosing the right code for the system that one wishes to build.

```
RenameFile
void
RenameFile(
   const char *old_name,
   const char *new_name
)
{
   Declarations
   Trim file specifications
   Try keep protection on all platforms
?%if (arch='pc') or (os='vms');
    Remove old file if it exists (VMS and PC)
    if (rafcc_FilExist(tmp2)) {
       if (rafcc_WriteProtected(0, tmp2)) {
          if (OK != rafcc_SetFileProtection(tmp2, 0, 0)) {
             return;
          }
       }
       sprintf(mess, "rafcc_RenameFile: error in unlink: %s: ", tmp2);
       ERRNONZERO(unlink(tmp2));
    }
   Rename file
   Set protection
}
```

Figure 9: The block is only executed on PC or VMS.

## Consistency

The intelligent document concept also makes it much easier to keep information consistent. This is particularly true with the relationship between source code, pseudo code and documentation, which are stored and presented together. When the source code changes, it is natural to update dependent information at the same time.

Of course much related information is stored separately from the source code. Consistency between different documents is kept, therefore, intact thanks to links.

### Extractability

The source code of an XML literate program is extracted from the document by a formatter before it can be used as input to a source code parser. This is not a problem as all the source code is stored in code elements.

The same goes for the other element types. From the program, interface descriptions, documentation, or metadata can be extracted. You can also extract the pseudo code, and specify how many levels of pseudo code you want from a large hierarchical document. Naturally any combination is possible, for example, it is possible to print the source code formatted into chapters, where each block becomes one chapter and the pseudo code becomes a title for the chapter.

### Simplicity

### The User Interface

As we all know, SGML has not become very widespread because it is too difficult to use. From this we can learn that a programming environment such as the one I have described must be very straightforward and easy to use. Above all this means that the user must not be overloaded with the mechanisms used to describe an XML document, such as tags and attributes. The challenge here is to ensure that the user gets the advantages of XML, without having to think about XML as such.

Normal users should never have to bother about tags. Our solution to this problem is autotagging. Code and pseudo code tags are created automatically, and are normally invisible. Note and interface sections must be created specifically, but that is easily done with a "create section" operation.

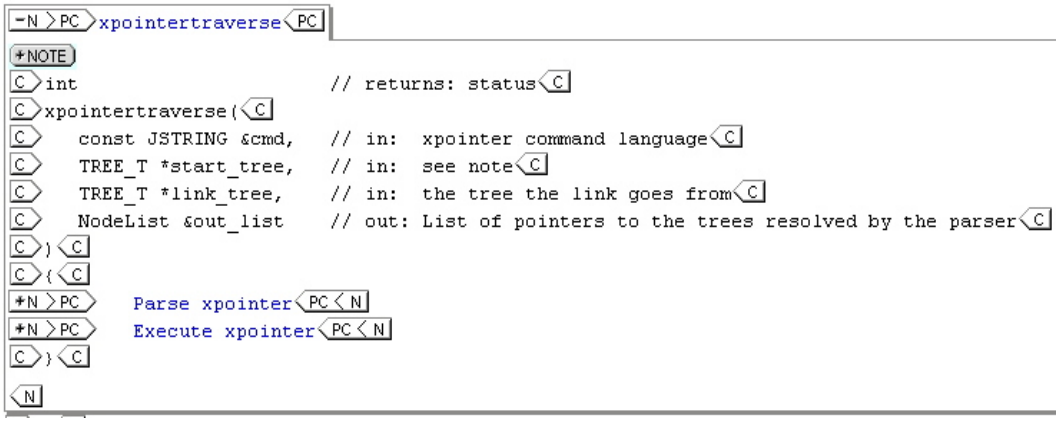



Figure 10: Tags are created automatically, and only visible if the user chooses to have them presented.

Attributes work the same way. A very simple user interface must be provided for the attributes that are in use. The user only needs to know that code can be marked as hidden, for example, but not that an XML attribute is used to store the information.

Only a few new commands are needed:

- create block division
- delete block division
- open block
- close block
- create link
- toggle hidden/not hidden

All commands must be available from menus and through keyboard shortcuts.

### Programming Language Independence

Even with a simple user interface, program management becomes complex if each programming language requires its own environment. Moreover, the logic of the program does not depend on the grammar of the language that is used to implement it. Although, programming languages are first of all machine-re adable, we want the program itself to be human-readable. It is a basic requirement for us that the structure of the program should be language-independent. The same goes for links –the programmer has complete freedom to make the links that are most useful in each case. Regardless of the language used, literate programming in XML generates the same benefits.

Long-term software construction projects may need to change language several times, by using the approach I have outlined this can be done using the same tool and the same way of structuring information. We have changed language a number times using Assembly languages, Pascal, C, and C++ and have successfully applied the principles I describe here to provide an overview of our programs. We also make literate programs out of makefiles and Unix-script as well as the other types of scripts. It would work just as well with Cobol, Java, etc.

### The Process

This chapter describes the process that programmers uses to compile their programs. A programmer writes the combined source code and documentation files in the XML editor. The files are not given the extension .xml as normal XML files but an extension that indictes the language the file is written in. This is so the 'makefile' can function. A file written in C++ is called, for example, litprog.ccx and its associated header file is called litprog.hx.

The makefile contains so-called suffix rules which state how a .ccx file should be outputted when compiling.

```
.ccx.obj
   perl xlow.pl $< > $*.cc
   $(cc) $(cflags) $*.cc
```

In this example we use perl script to extract the code from the XML file before compiling. A simple perl script used to extract the code is shown in the following example:

```
#!/usr/local/bin/perl
use XML::Parser;
sub char_handler {
  my ( $p, $data ) = @_;
  if ($p->current_element() eq "c") {
    $code_line = $code_line.$data ;
  }
}
sub end_element_handler {
  my ( $p, $data ) = @_;
  if ($data eq "c") {
    if ($code_line eq "") {
      $code_line = "\n";
    }
    print $code_line ;
    $code_line = "";
  }
}
sub default_handler {}
sub parse_file {
  my $file = shift(@ARGV);
  if ($file eq "") { print "Usage: xlow filename"; }
  else {
    die "Can't find file \"$file\"" unless -f $file;
    $parser = new XML::Parser(ErrorContext => 2);
    $parser->setHandlers(Char => \&char_handler,
                         End => \&end_element_handler,
                         Default => \&default_handler);
    $parser->parsefile($file);
  }
}
parse_file();
```

A simple DTD for literate programming is shown in the following example. This DTD can be considered as a template upon which any literate programming DTD can be based.

```
<!-- X-Link definition -->
<!ENTITY % linkattr
      "xml:link (simple|extended) 'simple'
      href CDATA #IMPLIED
      show (embed|replace|new) #IMPLIED
      actuate (auto|user) #IMPLIED
      behavior CDATA #IMPLIED"
>
<!-- Hidden definition -->
<!ENTITY % hidden "hidden (hidden) #IMPLIED">
<!-- The whole file -->
```

```
<!ELEMENT litprog (head, prog) >
<!-- The head contains metadata -->
<!ELEMENT head (date, time, user, file?, identity?, copyright?) >
<!ELEMENT date (#PCDATA) >
<!ELEMENT time (#PCDATA) >
<!ELEMENT user (#PCDATA) >
<!ELEMENT file (#PCDATA) >
<!ELEMENT identity (#PCDATA) >
<!-- The program section -->
<!ELEMENT prog (n | c | note)* >
<!-- A node containing pseudocode and more nodes -->
<!ELEMENT n (pc, (n | c | note)*) >
<!ATTLIST n
      %hidden;
      id ID #IMPLIED
>
<!-- Here is the documentation part -->
<!ELEMENT note (title, (p | div)*) >
<!ATTLIST note
      %hidden;
      id ID #IMPLIED
>
<!ELEMENT div (title, (p | div)*) >
<!ATTLIST div
      %hidden;
      id ID #IMPLIED
>
<!ELEMENT title (#PCDATA) >
<!ELEMENT p (#PCDATA | link)* >
<!ATTLIST p
   %linkattr;
   %hidden;
>
<!-- Pseudocode -->
<!ELEMENT pc (#PCDATA) >
<!-- Code that goes to the compiler -->
<!ELEMENT c (#PCDATA | link)* >
<!ATTLIST c
   %linkattr;
   %hidden;
>
<!-- Inline link -->
<!ELEMENT link (#PCDATA) >
<!ATTLIST link %linkattr; %hidden; >
```

The structure and pseudocode is built with the tags "n", for node, "pc" for pseudocode and "c" for code. This section is styled with fixed width fonts.

```
<n><pc>Declarations</pc>
<c>int a = 10;</c>
<c>int b = 20;</c>
<c>char *str = "Pierrou";</c>
</n>
```

To handle links we use XLink [W3C], the default show method is "embed". Why using embed is described in the "Links" chapter.

```
<n><pc>Parse xpointer</pc>
<c xml:link=simple href=#id(XPPARSER)>XPPARSER p(case_sensitive);</c>
<c xml:link=simple href=#id(XPPARSER_Parse)>COMPILED_XP *tree = p.Parse(cmd);</c>
</n>
```

Metadata is placed in the "head" tag.

The "note" element contains the text describing the part of the program. The example DTD above contains only simple paragraphs (p) and sections (div) with titles (title). The "note" element can be completed with tags for handling tables [CAL], mathematics [MAML], lists and other common documentation features. This section is styled with proportional fonts.

### Conclusion

The technologies that we bring together in a programming environment are structure, links, and literate programming. Structure gives us an overview and the ability to treat a part of the program that constitutes one logical unit as one textual unit. Links make it easier to follow the flow of the program and to find related information, without losing the original context. Literate programming lets us combine documentation and code, thus making the code easier to understand and reducing the risk that the documentation becomes outdated.

From the vantage point of our long experience with this way of thinking and working, I can recommend literate programming in XML for all large software projects. The benefits include:

- Code maintenance does not depend on the presence of the person who wrote the code.
- New employees become productive in a much shorter time.
- Software quality is improved.
- Innovation is increased when programmers are freed from the tedious task of interpreting old code.
- It is easier to reuse existing code.

I am convinced that for most projects the costs for getting started with this programming environment would be an excellent long-term investment.

BIBLIOGRAPHY


| | |
|---|---|
| [MK] | Markus Knasmüller, Reverse Literate Programming. |
| [CL] | Christopher Lee, Literate Programming -- Propaganda and Tools. |
| [FAQ] | Literate programming FAQ. |
| [ABC] | Anthony Bruce Coates, XML and Literate Programming. |
| [RC] | Robin Cover, Literate Programming with SGML and XML. (http://www.oasis-open.org/cover/xmlLitProg.htm l) |
| [W3C] | XML Linking Language (XLink), World Wide Web Consortium Working Draft 3-March-1998 (http://www.w3.org/TR/1998/WD-xlink-19980303) |
| [CAL] | CALS table model Document Type Definition, SGML Open Technical Memorandum TM 9502:1995 (http://www.oasis-open.org/cover/tr9502.html) |
| [MAML] | Mathematical Markup Language (MathML™) 1.01 (http://www.w3.org/TR/REC-MathML/) |
| [XSL] | Extensible Stylesheet Language (XSL) (http://www.w3.org/Style/XSL/) |