# Literate Programming in XML

**Norman Walsh**
XML Standards Architect
Sun Microsystems, Inc.

One Network Drive, MS UBUR02-201
Burlington, MA 01803-0902
US

$Id: paper.xml,v 1.2 2002/12/16 18:14:08 ndw Exp $

15 Oct 2002

Literate programming is a programming and documentation methodology. Its central tenet is that documentation is more important than source code and should be the focus of a programmer's activity.

Literate programming facilitates this approach by combining code and documentation into a single, unified source document. One interesting aspect of this combined form is that it is neither source code nor documentation. Instead, a literate programming system provides tools that allow a user to extract the source code or documentation automatically, but neither of these extracted forms is ever modified.

Using a literate programming system offers some interesting benefits for many programming styles. Because the combined format is machine processed to produce the source code, the author is no longer required to maintain or write the code in the linear fashion that the computer ultimately expects. This is clearly advantageous for top-down and bottom-up design strategies. It may also have benefits for more modern programming methodologies, such as Extreme Programming.

Typical literate programming systems are quite complex. They are built on top of some underlying documentation system (such as TeX) and described in terms of the macros and other documentation markup required to describe an xweb document.

However, it quickly becomes apparent that XML can greatly simplify this situation. By stipulating that the documentation format include a few (namespaced) elements, it is possible to implement literate programming in XML on top of any format that the author chooses: DocBook, TEI, XHTML, you name it.

In the past few years, the number of XML vocabularies has exploded. Where there used to be just a few, there are now hundreds. In addition, many of these new vocabularies have all sorts of sophisticated processing expectations: XSLT, W3C XML Schema, RELAX NG, Schematron, SAML, SVG, and MathML just to name a small handful.

Luckily, Literate Programming with XML applies equally well to XML, so it is possible to apply a literate programming methodology to the development of XML vocabularies.

This paper describes the design and implementation of a literate programming system using XML and XSLT. The resulting system is equally capable of authoring systems in traditional programming languages and systems that are themselves built from XML. The paper includes several examples to demonstrate these features and pointers to real-world systems that are actively exploiting the power it offers.

## Table of Contents

Literate programming [DMallLP] is a programming and documentation methodology. Its central tenet is that documentation is more important than source code and should be the focus of a programmer's activity. Donald Knuth [DKnuth84] observed:

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

Literate programming facilitates this approach by combining code and documentation into a single, unified source document. One interesting aspect of this combined form is that it is neither source code nor documentation. Instead, a literate programming system provides tools that allow a user to extract the source code or documentation automatically, but neither of these extracted forms is ever modified.

The tool that extracts source code is traditionally called *tangle* while the tool that extracts documentation is called *weave*. The unified authoring format is traditionally called a *web document*[1]. Because in the system described here we have XML web files, we call them *xweb* files. So an author “tangles” her code for the compiler and “weaves” documentation, but always she authors the “xweb”.

There are significant learning and training issues associated with adopting a system like this. It is so radically different from what most programmers are used to doing that there may be some resistance to adopting this new methodology. The system definitely imposes short-term productivity losses, but the long-term rewards in program maintainability and understandability can be tremendous. In any event, these issues are outside the scope of this paper.

Using a literate programming system offers some interesting benefits for many programming styles. Because the xweb format is machine processed to produce the source code, the author is no longer required to maintain or write the code in the linear fashion that the computer ultimately expects. This is clearly advantageous for top-down and bottom-up design strategies. It may also have benefits for more modern programming methodologies, such as Extreme Programming [KBeck99].

This paper assumes that you have some familiarity with the concepts of literate programming. For more general information about literate programming, see [DKnuth84] and [DMallLP]. For discussion of other SGML/XML literate programming systems, see [CoverLP].

# How Does Literate Programming Work?

In a nutshell, literate programming allows an author to develop a program as a logical set of “program fragments”. Each program fragment contains some small, functional unit of thoroughly documented processing logic. Program fragments can refer to each other, allowing the author to construct complex fragments from simple ones. The author is free to organize these fragments in any order that facilitates their documentation.

[1]The literate programming system predates the World Wide Web by at least 15 years; the name “web” is derived from the notion that code plus documentation form a unified network or web of information.

The web of fragments and fragment references forms the complete program. Given a starting point, the web can be tangled together into an executable program (with the program fragments reordered to suit the requirements of the language processor rather than the requirements of prose exposition) or woven into documentation.

The sophistication of tangle and weave varies between literate programming systems. Some tangle processors, for example, take care to adjust identifier names so that they will be both short and unique (this was a necessity for the original Web system where the language processor was a Pascal compiler that did not accept identifiers longer than seven characters). Knuth's tangle also performed a systematic source-code “uglification” to discourage maintainance programmers from working directly on the tangled code instead of the Web sources. Similarly, weave programs can provide quite sophisticated facilities such as pretty printing of the source code fragments and the generate of special indexes and concordances.

# Harnessing the Power of XML

Typical literate programming systems are quite complex. They are built on top of some underlying documentation system (such as TeX) and described in terms of the macros and other documentation markup required to describe an xweb document.

Indeed, my first forays towards an XML literate programming system were built with the expectation that DocBook would be the underlying documentation format. However, it quickly became apparent that this was unnecessary. By stipulating that the documentation format would include two new (namespaced) elements, and the xweb just a few more, it was possible to implement literate programming in XML on top of any format that the author chooses: DocBook, TEI, XHTML, you name it.

This is possible because a literate programming system only needs to be able to do four things:

1. Parse the xweb document.
2. Identify the fragments of code.
3. Identify cross-references to those fragments.
4. Identify where the source code “begins”.

XML makes each of these tasks easy.

The literate programming system described in this paper uses two elements to achieve these ends: `src:fragment` identifies a fragment and `src:fragref` identifies a cross-reference to a fragment.

The `src:fragref` uses XML ID/IDREF to point to the `src:fragments`. The fragment where processing begins is identified by an ID. The default ID is “top”.

# An Example of Literate Programming

Figure 1 shows a literate programming example[2]. It is a Perl program for calculating members of the Fibonacci series.

### Figure 1. The Fibonacci Series XWeb File

```
<article xmlns:src="http://nwalsh.com/xmlns/litprog/fragment"
         xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<articleinfo>
```

[2]The examples in this paper are fairly trivial and intentionally short to conserve space. They are meant only to demonstrate the way the process works. In particular, they are woefully lacking on documentation which is unfortunate since that's really the point of literate programming (though not this paper)

```
<title>Calculating Members of the Fibonacci Series</title>
<author>
  <firstname>Norman</firstname>
  <surname>Walsh</surname>
</author>
</articleinfo>

<para>This trivial document describes a simple, recursive
implementation of the Fibonacci series in Perl. The principal
motivation for this document is to demonstrate the use of Literate XML.
</para>

<section><title>Recursive Definition</title>

<para>The Fibonacci series begins: 1 1 2 3 5 8 13... Each member of
the series, after the first two, is the sum of the preceding two
members.</para>

<para>This can be implemented recursively by calculating the preceding
two members of the series and returning their sum:</para>

<src:fragment id="sub.fib.recursion">
&amp;fib($n-2) + &amp;fib($n-1);
</src:fragment>
</section>

<section><title>The <function>fib</function> Function</title>

<para>The heart of this program is the recursive function that
calculates the members of the Fibonacci series.</para>

<para>The first and second members of the Fibonnacci series are
<quote>1</quote>; all other values are calculated recursively.</para>

<src:fragment id="sub.fib">
sub fib {
    my $n = shift;

    if ($n &lt;= 2) {
        return 1;
    } else {
        return <src:fragref linkend="sub.fib.recursion"/>
    }
}
</src:fragment>
</section>

<section><title>Code Preamble</title>

<para>The program preamble simply establishes a default location for
the Perl executable and informs the interpreter that we want to use
the strict pragma.</para>

<src:fragment id="preamble">
#!/usr/bin/perl -w

use strict;
</src:fragment>
</section>
```

```
<section><title>Argument Checking</title>

<para>This program expects its argument on the command line and it expects
that argument to be an unsigned decimal integer.</para>

<src:fragment id="argcheck">
my $num = shift @ARGV || die;

die "Not a number: $num\n" if $num !~ /^\d+$/;
</src:fragment>
</section>

<section><title>The Program</title>

<para>The program prints out the Fibonacci number requested:</para>

<src:fragment id="top">
<src:fragref linkend="preamble"/>
<src:fragref linkend="argcheck"/>

print "Fib($num) = ", &amp;fib($num), "\n";

<src:fragref linkend="sub.fib"/>
</src:fragment>
</section>

</article>
```

This xweb can be tangled into Perl, as shown in Figure 2, or woven into DocBook [example/fib.xml] that can be subsequently transformed into HTML [example/fib.html], or PDF [example/fib.pdf] or used for any other sort of DocBook processing.

## Figure 2. The Fibonacci Series Tangled Perl File

```
#!/usr/bin/perl -w

use strict;


my $num = shift @ARGV || die;

die "Not a number: $num\n" if $num !~ /^\d+$/;


print "Fib($num) = ", &fib($num), "\n";


sub fib {
    my $n = shift;

    if ($n <= 2) {
        return 1;
    } else {
        return
&fib($n-2) + &fib($n-1);
```

```
    }
}
```

# Writing a Literate Programming XWeb

An xweb file can be written in any XML vocabulary (or mixture of vocabularies) that is convenient for the author. The system described in this paper relies on the ability to add a few new elements to the markup vocabulary from the "`http://nwalsh.com/xmlns/litprog/fragment`" namespace. In this article, we use the namespace prefix "`src:`" to identify elements from that namespace.

The most important elements are `src:fragment` and `src:fragref`.

## Identifying Fragments

Fragments of source code are identified in the xweb file with the `src:fragment` element. This element has a required `id` attribute (of type ID).

All of the content (and only the content) of the `src:fragment` element will be tangled together to produce the source code of your application.

## Referencing Other Fragments

Writing a bunch of disconnected fragments would not be very useful if there was no way to indicate how they are related. The `src:fragref` element is used to reference other fragments. It has a `linkend` attribute which points by ID reference to the ID of a `src:fragment`. It is an error for a `src:fragref` to point to any other element type.

Inside a `src:fragment`, a fragment reference implies that the referenced code will appear at the location of the reference when the xweb is tangled. Outside a fragment, the reference creates a link to the fragment.

## Processing Expectations of Fragments

In order to tangle a xweb, the tangle processor must know which fragment is the "top" of the program. In the system described by this paper, the top fragment is identified by ID. The default ID for the top fragment is "`top`", but any other ID may be used.

Tangle constructs the program by building the transitive closure of fragments referenced from the top fragment.

There is one special semantic rule associated with fragments. If a fragment begins with one or more newlines, the first newline is discarded. Similarly, if it ends with one or more newlines, the last newline is also discarded.

This trick makes it possible to write fragments like this:

```
<src:fragment id="preamble">
Some lines of code.
Some more code.
</src:fragment>
```

Without introducing otherwise spurious whitespace above and below the code. If this trick were not employed, the preceding fragment would have to be written like this:

```
<src:fragment id="preamble">Some lines of code.
Some more code.</src:fragment>
```

which is harder to read and makes cutting and pasting much more tedious.

# Literate XML

In the past few years, the number of XML vocabularies has exploded. Where there used to be just a few, there are now hundreds. In addition, many of these new vocabularies have all sorts of sophisticated processing expectations: XSLT, W3C XML Schema, RELAX NG, Schematron, SAML, SVG, and MathML just to name a small handful.

Luckily, Literate Programming with XML applies equally well to XML, so it is possible to apply a literate programming methodology to the development of XML vocabularies.

In fact, the system described by this paper has been used not only to develop itself, but also in the development of the DocBook XSL Stylesheets. In fact, one of the motivating factors in the development of this system was to provide a better mechanism for building and documenting the DocBook stylesheets.

Writing Literate XML is easy. Figure 3 shows a a simple W3C XML Schema implemented in the literate style. This example uses XHTML as its documentation vocabulary. Like Figure 1, this xweb can be tangled into XSD [example/doc.xsd] or woven into documentation [example/doc.xml] that can be subsequently transformed into HTML [example/doc.html].

Weaving is still a two step process because weave produces an XML document that contains embeded `src:fragment` and `src:fragref` elements. Even in the case where the surrounding XML markup is XHTML, these elements need to be styled. It would be relatively easy to make a one-step weave stylesheet for the special case where the source was XHTML, but that would need to be adjusted for each user that wanted to change the way fragments are formatted. The burden of maintaining these extra weave stylesheets seems to outweigh the convenience.

### Figure 3. A Simple Document Schema in W3C XML Schema

```
<html xmlns="http://www.w3.org/TR/xhtml-basic"
      xmlns:xs='http://www.w3.org/2001/XMLSchema'
      xmlns:src="http://nwalsh.com/xmlns/litprog/fragment"
      xmlns:ex='urn:publicid:-:Norman+Walsh:Schema Example:EN'>
<head>
<title>Document Schema</title>
</head>
<body>

<div>
<h1>A Simple Document W3C XML Schema</h1>

<p>This schema defines the
<tt>urn:publicid:-:Norman+Walsh:Schema Example:EN</tt> namespace
by defining several elements and their complex types.
</p>

<p>Documents that conform to this schema have the general form:</p>

<pre>&lt;doc xmlns="urn:publicid:-:Norman+Walsh:Schema Example:EN">
  &lt;title>Sample Document&lt;/title>
  &lt;para>Some paragraphs.&lt;/para>
&lt;/doc></pre>
</div>

<div>
<h1><a name="types"/>The Types</h1>

<p>This schema only defines three element types: <tt>doc</tt>,
<tt>title</tt>, and <tt>para</tt>.</p>
```

```
<div>
<h2>The <tt>doc</tt> Type</h2>

<p>This is a document.</p>

<src:fragment id="doc.type">
  <xs:complexType name='doc'>
    <xs:sequence>
      <xs:element ref="ex:title" minOccurs='0' maxOccurs='1'/>
      <xs:choice minOccurs='1' maxOccurs='unbounded'>
        <xs:element ref='ex:para'/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</src:fragment>
</div>

<div>
<h2>The <tt>title</tt> Type</h2>

<p>This is a title.</p>

<src:fragment id="title.type">
  <xs:complexType name='title' mixed="true">
    <src:fragref linkend="role.attrib"/>
    <xs:anyAttribute namespace="##other" processContents="lax"/>
  </xs:complexType>
</src:fragment>
</div>

<div>
<h2>The <tt>para</tt> Type</h2>

<p>This is a paragraph.</p>

<src:fragment id="para.type">
  <xs:complexType name='para' mixed="true">
    <src:fragref linkend="role.attrib"/>
    <xs:anyAttribute namespace="##other" processContents="lax"/>
  </xs:complexType>
</src:fragment>
</div>
</div>

<div>
<h1>The <tt>role</tt> Attribute</h1>

<p>Each of the complex types in this schema allows an optional role attribute.
The role attribute is simply a string.</p>

<src:fragment id="role.attrib">
    <xs:attribute name="role" type="xs:string"/>
</src:fragment>
</div>

<div>
<h1>The Elements</h1>

<p>This schema defines one element of each <a href="#types">complex
type</a>.</p>
```

```
<src:fragment id="elements">
  <xs:element name="doc" type="ex:doc"/>
  <xs:element name="para" type="ex:para"/>
  <xs:element name="title" type="ex:title"/>
</src:fragment>
</div>

<div>
<h1>The Schema</h1>

<p>The schema wrapper surrounds all these definitions.</p>

<src:fragment id="top" mundane-result-prefixes="ex xs">
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
           xmlns:ex='urn:publicid:-:Norman+Walsh:Schema Example:EN'
           targetNamespace='urn:publicid:-:Norman+Walsh:Schema Example:EN'
           elementFormDefault='qualified'>
  <src:fragref linkend="doc.type"/>
  <src:fragref linkend="title.type"/>
  <src:fragref linkend="para.type"/>
  <src:fragref linkend="elements"/>
</xs:schema>
</src:fragment>
</div>

</body>
</html>
```

## Limitations

The fact that the Literate XML document must be a well-formed XML document introduces a few limitations. Most we can work around.

### Document Type Declaration

There is no straightforward way to include a document type declaration for the source document. The document type declaration in the xweb is for the xweb itself, not the source it generates.

To work around this problem, the "`http://nwalsh.com/xmlns/litprog/fragment`" namespace includes a `src:passthrough` element. When tangled, the content of `src:passthrough` element (which must contain only text) is written with output escaping disabled.

Tangling the following fragment will create a source vocabulary with a document type declaration:

```
<src:fragment id="top">
<src:passthrough>&lt;!DOCTYPE foo PUBLIC "foo" SYSTEM "http://example.com/foo"&gt;
</src:passthrough>

<foo>...</foo>
</src:fragment>
```

## Namespace Declarations

When fragments are distributed through the xweb document, it is sometimes necessary to rearrange namespace declarations. Unfortunately, XSLT cannot distinguish between namespaces that are declared on a given element and other namespaces that are simply in scope.

Sometimes this results in woven documentation with far more namespace declarations than are actually necessary or interesting. In order to work around this problem, every `src:fragment` can identify a set of uninteresting prefixes. Namespaces with the specified prefixes will not be represented in the woven documentation for that fragment.

The attribute `mundane-result-prefixes` contains a list of prefixes for which namespace declarations will not be generated. This can also be specified as a stylesheet parameter, `$mundane-result-prefixes`.

Consider the following example:

```
<src:fragment id="equation">
  <xsl:template match="eq">
    <m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
      <xsl:apply-templates/>
    </m:math>
  </xsl:template>
</src:fragment>
```

In this example, the declaration of the MathML namespace is probably important in the context of the documentation (because it's locally declared), but the declaration of the XSL namespace is probably not. If we simply output all of the namespace declarations, we'll get:

```
<src:fragment id="equation">
  <xsl:template xmlns:xsl="http://www.w3.org/1999/XSL/Transform" match="eq">
    <m:math xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
            xmlns:m="http://www.w3.org/1998/Math/MathML">
      <xsl:apply-templates xmlns:xsl="http://www.w3.org/1999/XSL/Transform"/>
    </m:math>
  </xsl:template>
</src:fragment>
```

This is rather confusing, so we suppress the XSL namespace with `mundane-result-prefixes`:

```
<src:fragment id="equation" mundane-result-prefixes="xsl">
  <xsl:template match="eq">
  ...
```

and get:

```
<src:fragment id="equation">
  <xsl:template match="eq">
    <m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
      <xsl:apply-templates/>
    </m:math>
  </xsl:template>
</src:fragment>
```

The mundane result prefixes setting from the root fragment automatically applies to all fragments, so often it is the only one that needs to be set.

The tangle stylesheet ignores these settings, it must build a document with a completely valid set of in-scope namespaces. The fact that it may sometimes build a document with extra namespace declarations is not problematic becuase they will all be consistent.

### Balanced Fragments

XML fragments must be properly balanced. In a non-XML program, you can generally break the code into fragments arbitrarily. When the source is XML, you must maintain the well-formedness constraints of XML, so all fragments must be properly balanced. This sometimes means that the documentation contains an extra level of fragmentation, but it is not a serious problem.

### Schemas for Literate XML Documents

It isn't easy to develop schemas that can validate Literate XML documents. Luckily, validating the documentation is usually pretty easy and using literate programming has no impact on validating the source. (This validation problem is not unique to Literate XML documents, it also occurs in XSLT stylesheets that use literal result elements and in other languages that allow fairly arbitrary mixing of namespaces.)

## The Literate Programming in XML System

The system is quite small, it consists principally of three XSL Stylesheets. Literate Programming with XML is implemented using literate programming. Consequently, each of these stylesheets is available as a xweb document and as published documentation.

- `tangle.xsl` tangles xweb files (for non-XML programs) into source code.
- `xtangle.xsl` tangles xweb files for XML vocabularies into source XML.
- `weave.xsl` weaves xweb files into documentation.

There are some additional support tools for DocBook-based xwebs:

- `dtd/ldocbook.dtd` is a Literate DocBook XML customization layer.
- `wdocbook.xsl` weaves Literate DocBook XML documents.
- `html/ldocbook.xsl` makes HTML documentation out of woven Literate DocBook XML documents.
- `fo/ldocbook.xsl` makes XSL FO documentation out of woven Literate Docbook XML documents.

Weaving is simply a matter of processing the xweb document with the appropriate variant of the weave stylesheet. The resulting XML document can then be processed by whatever documentation processing system you choose.

Tangling is about the same, except that if the top fragment in your xweb is not identified with the id "`top`", you will need to pass the appropriate value in the `$top` parameter.

If you're interested in downloading these tools, see http://docbook.sourceforge.net/projects/litprog/.

## Acknowledgements

Several people saw early drafts of this work and made many insightful and helpful comments. In particular, I would like to thank Tony Coates, Chris Maden, Eve Maler, and Michael Sperberg-McQueen.

# References

[CoverLP] Cover, Robin. *The XML Cover Pages: Literate Programming with SGML and XML*. http://xml.coverpages.org/xmlLitProg.html.

[KBeck99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999. ISBN 201-61641-6.

[DMallLP] Mall, D. *Literate Programming*. http://www.literateprogramming.com/.

[DKnuth84] Donald E. Knuth. "Literate Programming". *The Computer Journal*, 27(2):97-111, May 1984.

[DKnuth86] Donald E. Knuth. *Computers and Typesetting: Volume B, TeX: The Program*. Addison-Wesley, 1986. ISBN 0-201-13437-3.

[DBHPref] Walsh, Norman. *HTML Parameter Reference*. http://docbook.sourceforge.net/projects/xsl/doc/html/param.html.