Theme-Based Literate Programming

Andreas Kacofegitis Neville Churcher

Software Visualisation Group, Department of Computer Science,
University of Canterbury, Private Bag 4800,
Christchurch, New Zealand
E-mail: {aka24,neville}@cosc.canterbury.ac.nz

Abstract

*The source code for computer programs is destined to be read by compilers and consequently its internal structure is heavily constrained. The compiler neither knows nor cares about such things as a program's internal structure, the relationships between its components and their specifications, the way design patterns are instantiated, the best way to explain its algorithms or how it is intended to be used. People do. Literate programming (LP) was invented by Donald Knuth as a way to address such problems. The idea is appealing but LP has not been adopted widely: the lack of good tools, difficulties with object-oriented languages and the limitations of a single psychological order are among the reasons. In this paper we report the development of theme-based literate programming (*TBLP*). Themes are extremely flexible: they may be aimed at particular reader groups or represent aspects of the program. Features of* TBLP *include an extended chunk model which accommodates a richer variety of types, an extended connection model which allows chunks to be threaded together into multiple themes, an enhanced processing model which generalises tangling and weaving and a chunk-level version management system. XML is used to represent the web structure and XML-based technologies such as XSLT are used in processing. This provides flexibility and extensibility, allowing users to define new chunk types. An application which implements* TBLP *is presented and the integration of* TBLP *with software engineering processes is discussed.*

Keywords: Literate programming, software engineering, XML

1 Introduction

Programming is difficult. Software engineering is even more challenging. Much effort is spent, not on the interesting, creative and scientifically significant aspects of software development, but on the mundane activities associated with realising a software system. Chief amongst these is coding.

Programming has always embodied a central contradiction. The artifacts produced are required to be read by a machine and are subject to constraints such as the rigid syntax of programming languages. In particular, the order in which components appear is generally constrained—increasingly so at lower levels.

However, the human authors of programs have quite different priorities and needs. Software systems are typically both large and complex, consisting of components of different kinds with a variety of types of relationships among them. Ideally, this structure will be closely related to the structure of the "real" world problem the program is intended to solve.

A fundamental tenet of software engineering is that, however complex a program may be, it is always possible to describe it in terms of simple parts and simple relationships between them. Unfortunately, no "right" way to obtain and express such a description has emerged.

Various approaches to the decomposition of large systems into smaller components have been suggested. These include top-down and bottom-up techniques and the nucleus-centred approach [14], which is arguably the forerunner of object oriented design techniques. Software engineers may wish to use any combination of existing techniques and new ones emerge frequently. Any generally applicable system representation technique should be independent of the methodologies and approaches used on individual projects.

Source code is not the only product of the software engineering process. Many other artifacts represent aspects such as requirements, design, tests and evolution: such "documentation" is typically a combination of text and diagrams and is intended primarily for human readers.

The rigid syntax of programming languages limits the possibilities for internal documentation to some form of comments, though these may be supported to some extent by documentation generators such as Sun's javadoc tool for Java.

In an ideal world, the documentation elements would not simply be attached to source code fragments but would also have their own higher-level relationships. The connections between documentation and code should be "live" in order to enable changes in one to be reflected by updates to the other so that the overall system is always both self-documenting and self-consistent.

The importance of this aspect cannot be emphasised too strongly. Experience suggests that during the maintenance phase of the development cycle some 50% of a programmer's time is spent trying to understand existing code. The maintenance phase dominates the overall lifecyle costs, with textbooks typically suggesting 50–75%, so the potential savings are substantial. If conventional documentation is inadequate or out of date then it will be ignored: programmers will simply rely on code listings.

Points such as the following arise when the issue of providing support for the development of self-documenting and self-consistent software.

- Programmers should be free to describe the system's parts and relationships in whatever order is best for human comprehension—not in some rigidly predetermined order such as top-down.

- Programmers implement features in an order that seems natural. It is likely that readers will better comprehend software by studying its various parts in roughly the order in which they were written.

- Although top-down programming has the advantage that there is a clear sense of direction, it also has the disadvantage that details are not pinned down until rather late. Conversely, bottom-up programming has the advantage that the vocabulary of available components becomes more and more powerful as coding proceeds but also has the disadvantage that overall program organisation and higher level design is postponed. Consequently, development may be somewhat directionless.

Each of these paraphrases a point made by Donald Knuth, who proposed literate programming (LP) as a solution [8]. In LP, software is assembled from *chunks*, the atomic unit of code and documentation. Programmers are free to choose the size, content and relationships of chunks independent of any design methodology.

The representation of software as connected chunks is known as a *web* and pre-dates the WWW by some years. An LP implementation supports two operations, *tangle* and *weave*, on a web. Tangling assembles the source code ready for compilation: nested code chunks are expanded and documentation chunks are omitted. Weaving produces the output intended for human readers: code and documentation chunks are interleaved in the order specified by the author. A simple example is shown in Figure 1.

Despite its potential advantages, LP has not been widely adopted. There are a number of reasons for this: they are more to do with issues such as tools and support for modern languages than with fundamental LP principles. Nevertheless, LP has many proponents and continues to be used on both large and small projects [4, 10, 2, for example] in development and education.

A good literate program is easy to recognise but, with current tools, difficult to write. This is reminiscent of such techniques as Nassi-Shneiderman charts [13], used to represent control flow, which were essentially "read-only" until CASE tool support became available in the 1980s.

Nevertheless, we believe that LP has much to offer in the current software development context and, with appropriate extensions, can realise its potential more fully than has been the case thus far.

LP has much in common with techniques, such as aspect-oriented programming [7] and wikis [11], which are currently attracting much attention.

In this paper, we develop a new model for LP which allows LP to be integrated more seamlessly with software engineering development processes as well as supporting greater functionality. Chunk types are not limited to either code or documentation and tangling and weaving are subsumed in a more general processing model. We introduce *themes* to allow authors greater freedom to express their intentions. Themes allow multiple processing orders to be specified for a given web. We show how XML technology permits the construction of flexible and extensible implementations.

The remainder of this paper is structured as follows. In the next section, we describe LP in more detail and discuss some current implementation approaches. In Section 3 we examine the drawbacks of LP in the current development context and identify opportunities for addressing these. In Section 4 we introduce our theme-based literate programming (TBLP) model. Implementation issues are discussed in Section 5 and a prototype implementation, CBDE (Context Based Development Environment) is presented. Our conclusions and future work appear in Section 6.

2 Literate Programming

A number of LP systems have been developed (for further information see the FAQ for the Usenet group `comp.programming.literate` and the review by Smith [16]). Each supports the creation of documentation and code chunks and the specification of relationships between them. However, their functionality, sophistication and underlying models vary considerably.

Some systems are restricted to specific programming languages and documentation formats (e.g. Knuth's original WEB system was for the Pascal + TeX combination). In some cases this allows language-specific features, such as cross-referencing and pretty-printing, to be provided.

The noweb tool [15] was an important advance. It provided only simple language-independent features but also introduced a pipeline model which allowed extensibility. Features such as cross-referencing and pretty-printing could be implemented by writing filters which processed the pipeline representation.

One common criticism levelled at LP systems is the *three syntax problem*: users must concurrently use a programming language, a text formatting language (typically LaTeX) and an LP web structuring language. This is not only difficult for people but is also challenging for tool builders attempting to provide language sensitive editors and the like.

A very simple example is shown in Figure 1. Note the use of nesting (chunks ⟨*includes*⟩, ⟨*definitions*⟩ and ⟨*main program*⟩ are nested within chunk ⟨*greet*⟩) and that the order of chunks differs in the output of tangle and weave. The ⟨*includes*⟩ chunk appears last in the explanation of the program, reflecting its low relevance to the specific goals of the application, but first in the tangled code, satisfying the syntax requirements of the C language.

LP tools have been used on substantial documents such as the documentation for TeX [9] (WEB), software libraries [10] (CWEB) and books [4] (noweb).

Little real progress has been made in the last few years. Some tools have been extended to produce XML output (see `http://www.csse.monash.edu.au/~ajh/research/literate/index.html` for an example) but the underlying model typically remains the same. One interesting variation is LEO (`http://personalpages.tds.net/%7eedream/front.html`) which is based on an outline-processing model.

3 Issues and Opportunities

Some consideration of the way LP might be used in software engineering has been made [1, 3]. Software engineering has changed greatly since the advent of LP in the mid 1980s. Unfortunately, many of these changes have further hindered the adoption and utility of current LP implementations.

Several problems arise if the programmer is unable to interact with the tangled code artifacts. The correspondence between concepts in the woven documents (classes, methods, design patterns, algorithms, . . .) and those of their tangled counterparts (code fragments, scopes, references, . . .) is not always direct. This mismatch can lead to difficulties in areas such as design, coding and debugging. We will use the term *tangle scope* to refer to the relationships and struc-

```
The first program one writes in a new language
is invariably the ‘‘Hello World’’ program.
Here it is in C.
<<greet>>=
<<includes>>
<<definitions>>
<<main program>>
@ The [[<<definitions>>]] chunk hard-codes the text.
<<definitions>>=
const char *GREETING = "Hello World";
@ The [[<<main program>>]] could be changed to read the
greetee’s name from a command line argument.
<<main program>>=
int main(int argc, char **argv) {
    printf("%s", GREETING);
}
@ Only the standard IO library is needed.
<<includes>>=
#include <stdio.h>
@ That’s all there is to it.
```

(a) noweb input

```
#include <stdio.h>
const char *GREETING = "Hello World";
int main(int argc, char **argv) {
    printf("%s", GREETING);
}
```

(b) output of tangle

The first program one writes in a new language is invariably the “Hello World” program. Here it is in C.

⟨*greet*⟩≡
  ⟨*includes*⟩
  ⟨*definitions*⟩
  ⟨*main program*⟩

The ⟨*definitions*⟩ chunk hard-codes the text.

⟨*definitions*⟩≡
```
  const char *GREETING = "Hello World";
```

The ⟨*main program*⟩ could be changed to read the greetee's name from a command line argument.

⟨*main program*⟩≡
```
  int main(int argc, char **argv) {
      printf("%s", GREETING);
  }
```

Only the standard IO library is needed.

⟨*includes*⟩≡
```
  #include <stdio.h>
```

That's all there is to it.

(c) output of weave

Figure 1. Simple LP example: hello world

tures which pertain to the tangled program. These include the boundaries of control constructs, scopes and the requirement for definitions to precede uses. Similarly, we will use the term *web scope* to indicate features relevant in the woven documents. These include chunking criteria, nesting and psychological order.

Relevant factors include (in no particular order):

- The proliferation of object oriented languages presents several challenges. For example, overriding and overloading tax the indexing and cross-referencing facilities and are beyond the capabilities of almost all implementations. Effective browsing facilities are a vital part of OO development environments.

- Programs no longer consist of a single Pascal source file. Projects routinely involve many source files containing code in several languages. Typical LP implementations have poor support for multiple input files and even less for multiple language projects.

- The distinction between code and documentation is becoming blurred. For example, a web page may consist of mark-up (HTML, XML, ...) and scripts (Javascript, ...), as well as text and images. It isn't obvious what tangle and weave really mean in such contexts—one program's documentation may be another's code.

- HTML has changed software documentation for ever. One of the underlying assumptions of conventional LP is that the woven program will effectively appear as monolithic hard copy output which will be read in a single order. Documents intended to be accessed interactively may make much more use of cross-referencing and other dynamic presentation techniques.

- Version and configuration management are an essential part of the software engineering process and integrating with LP is hard.

- The atomic units for version control exist in the post-tangle world *tangle scope*, as do their scoping and other relationships. However, chunks and their relationships exist in *web scope*. The transformation from web scope to tangle scope is deterministic, though complex: the reverse transformation is beyond conventional LP systems.

- Debugging literate programs can be awkward: compilers report errors in terms of line numbers in the tangled output whereas programmers work with the woven documents.

- Documentation generators, the best known being Sun's javadoc, make use of some LP ideas. Something along the lines of documentation chunks is encoded in "magic" comment syntax. Such systems are suitable for delivering API documentation in HTML with extensive cross-references. However, the chunk granularity is fixed and corresponds strictly to constructs, such as methods, in the tangle scope.

- The three syntax problem is a challenge to tool builders. Some facilities, such as emacs modes sensitive to different syntaxes in documentation and code chunks, exist but are generally neither powerful nor convenient.

- It is not clear how best to design software with LP. Issues such as choosing chunk granularity and ordering are no less challenging than the corresponding activities in other techniques.

LP is effectively defined by implementations: no formal model for programs and operations on them exists.

4 Theme-Based Literate Programming

We propose TBLP in order not only to address some of the shortcomings of conventional LP and its current implementations but also to provide a secure basis for further LP evolution.

Our approach is motivated by a several factors:

chunk model More chunk types, each with its own attributes, should be supported.

theme model Many navigation paths through a given set of chunks are appropriate for different reader groups or purposes. Each such path corresponds to a conventional weave operation.

processing model The appearance and content of output documents should be independent of the source chunk set and be configurable by the user.

evolution We expect new languages and techniques to continue to emerge and wish to be able to support as many as possible.

Our chunk model, shown in simplified form in the class diagram of Figure 2, is based on a composite design pattern [5]. Some points to note are:

- There is no free text—everything is encapsulated in a well-defined chunk. This contrasts with typical LP systems where only chunk beginnings are defined and various kinds of "litter" are permitted.

- Chunks are not limited to the traditional code and documentation types. New types, with their own properties and behaviours, may be derived as required. For example, diagram fragments and unit tests might be placed in specific chunk types.

- Nesting, limited to code chunks in conventional LP, is extended to all chunk types—including (possibly heterogeneous) groups.
- Webs may be checked automatically to ensure that they remain *well-formed*.

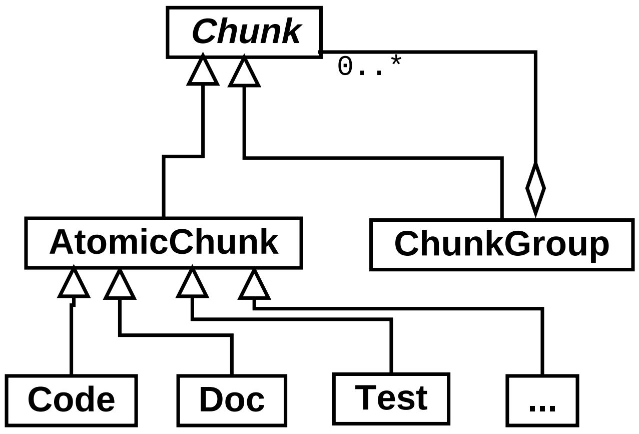


Figure 2. Chunk model (simplified)

In conventional LP the order of chunks in the (single) woven form is determined by their lexical order in the source file (see Figure 1). We refer to this as the *weave order*. The order of chunks in the tangled form is determined by both the nesting structure and lexical order. Figure 3 shows three documentation chunks (D1–D3) and three code chunks (C1–C3) together with the tangle order (solid arrows) and conventional weave order (unfilled arrows).

We may think of the LP web as a graph whose nodes are chunks and whose (directed) edges represent the nesting and ordering relationships between them. We may then reasonably consider other paths, such as that represented by the dashed arrows in Figure 3, through the web. Such paths need not include all nodes or edges.

Paths of this sort occur naturally in many ways including:

- descriptions of a program aimed at different reader groups.
- discussion of non-local features such as patterns.
- transient information such as the description of the effects of a defect identified during testing.
- pervasive aspects of the system.

We term these paths *themes*. Within this model, tangling and weaving are just two possible themes. Many more are possible. For example, the evolution of a set of chunks may be recorded by a theme which links versions (indicated by the grey arrow linking three versions of chunk C1 in Figure 3).

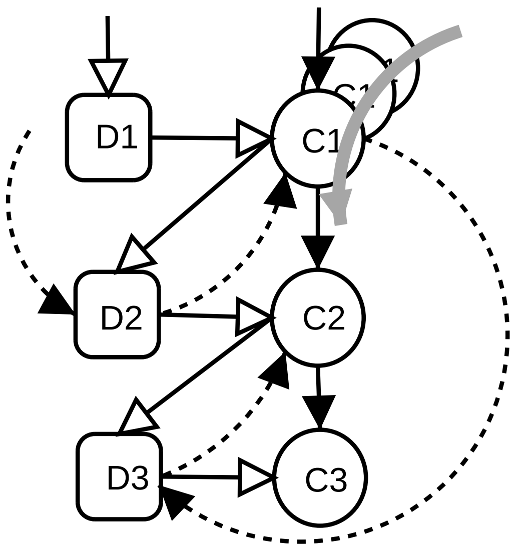


Figure 3. Chunk orders

Facilities for specifying and maintaining themes are required. Similarly, the processing of themes must admit nesting and ordering operations as well as selection and navigation.

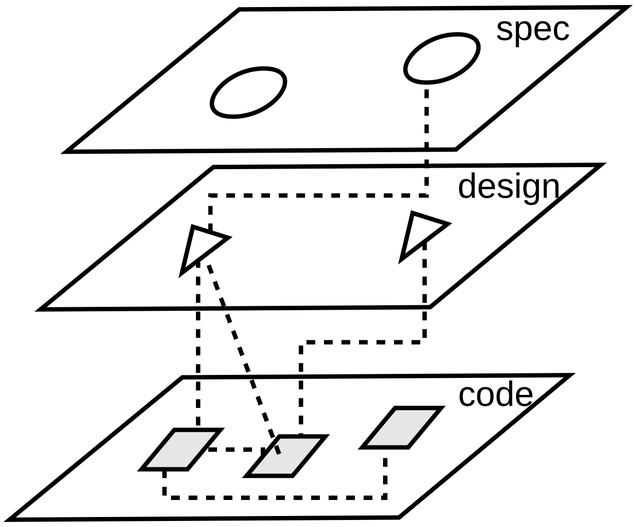


Figure 4. Themes and layers

Themes are a powerful and flexible way to provide multiple views of a given system. They also provide a cleaner way to relate LP to the software process. Figure 4 illustrates phases of the process as planes with corresponding chunk types. Dashed lines indicate how themes can connect chunks within and between phases. A design chunk (perhaps part of a UML class diagram) may be connected to several code chunks which implement it. A code chunk may be connected to the requirements it realises, its specification (perhaps a Z schema) and its test cases.

Themes may also be associated with the crosscutting *aspects* of aspect oriented programming [7] which represent system-wide issues such as quality of service. It is challenging to represent abstract and crosscutting concepts as well as design- and implementation-specific information. Themes potentially allow LP to play a part in integrating concepts such as aspects with conventional software engineering artifacts.

Wikis allow groups of developers to add and modify both content pages and links to provide a (WWW) web-based documentation system[11]. Any user is free to change both content and structure and a Wiki is likely to evolve in an unconstrained manner. The need for navigation aids through dynamically changing structures such as Wikis is reminiscent of theme-based LP.

We are continuing the development of the TBLP model. In particular, we are working towards a LP algebra and calculus analogous to the relational algebra and calculus. An algebraic formulation will enable operations to be specified in a procedural manner and is useful in designing implementations. Fundamental operators support chunk creation, ordering and nesting. A calculus formulation provides a complementary view of LP based on predicates. This is suitable for formulating queries (e.g. What common sections do these themes have? Does this theme lead from this chunk to that chunk?) and paths through the web. Calculus expressions are likely to be translated to their algebraic equivalents for evaluation. A sound algebra and calculus model are desirable for implementations based on technologies other than XML.

5 Implementation

Having proposed a chunk model for the basic components of LP and a theme model for their interconnection, we now turn to the issue of implementation. Our chosen architecture is shown schematically in Figure 5.

We use XML [12] as the fundamental representation of LP documents: XML parsing tools are used in our authoring tool and XSLT [6] transformations are used to perform much of the processing which takes the place of tangle and weave in conventional LP environments.

Individual chunks are stored in a repository. Chunks will be created, modified and deleted as the program evolves: all versions of each chunk may be maintained in the repository. This is important since themes may describe the temporal

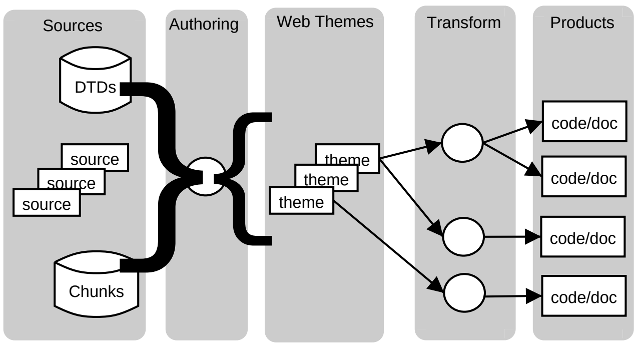


Figure 5. System architecture

evolution of components at a finer granularity than conventional file-based version management tools.

Document Type Definitions (DTD) allow a grammar for individual chunks and their relationships to be specified. Conformance to the chunk model of Figure 2 and the theme model may then be checked by validating parsers to ensure that documents are not only *well formed* but also *valid*.

Sources may also include other LP documents, such as noweb files or products of other TBLP projects, which may be converted using XML-based parsing and transformation tools.

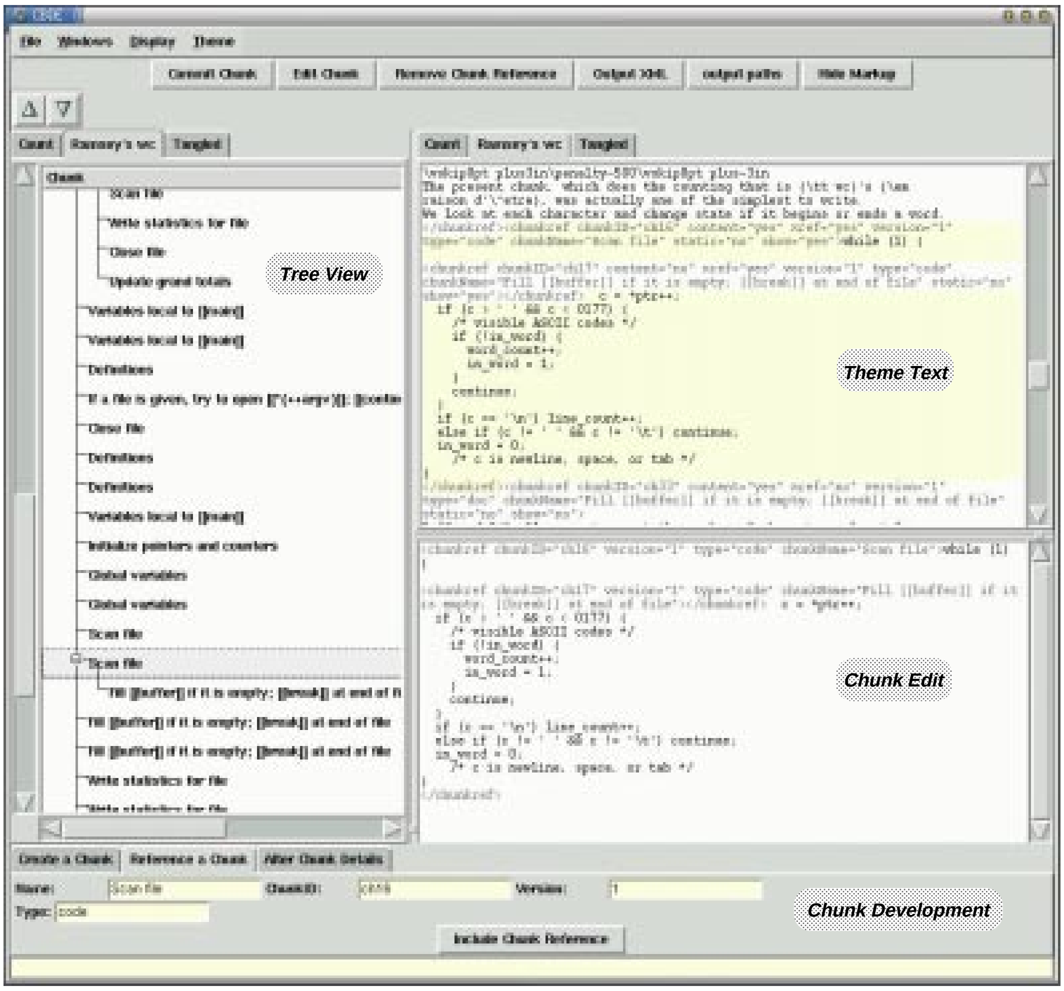
Figure 6(a) shows part of the CBDE interface, with some of the major components labelled. CBDEprovides authoring features and facilities for chunk and theme management. The web being edited is that of the wc utility—following its use as an example in the description of noweb [15].

Tabbed panes are used to permit rapid switching between themes. Figure 6(a) shows three open themes. The selected theme corresponds to the order used by Ramsey [15]. Another (Count) results from a different partitioning of the wc source into chunks. The third (Tangle) corresponds to the output of a conventional tangle operation: it contains only code chunks arranged in the "right" order.

The *tree view* panel at left shows the labels of chunks which appear in the selected theme. Expanding a tree node displays the names of chunks which are referenced within the expanded chunk.

The *theme text* view contains a text view of the content of the currently selected theme whose structure is visible in the tree view pane. This provides authors with information about the context of the current chunk editing operations.

Display attributes control the level of detail presented in the theme text view and, in some cases, the processed output. In Figure 7 the Show (display chunk names), Xref (display chunk cross-references), Content (display entire chunk content) attributes have been selected for each chunk but the Static (don't update content if newer chunk version is created) has not.

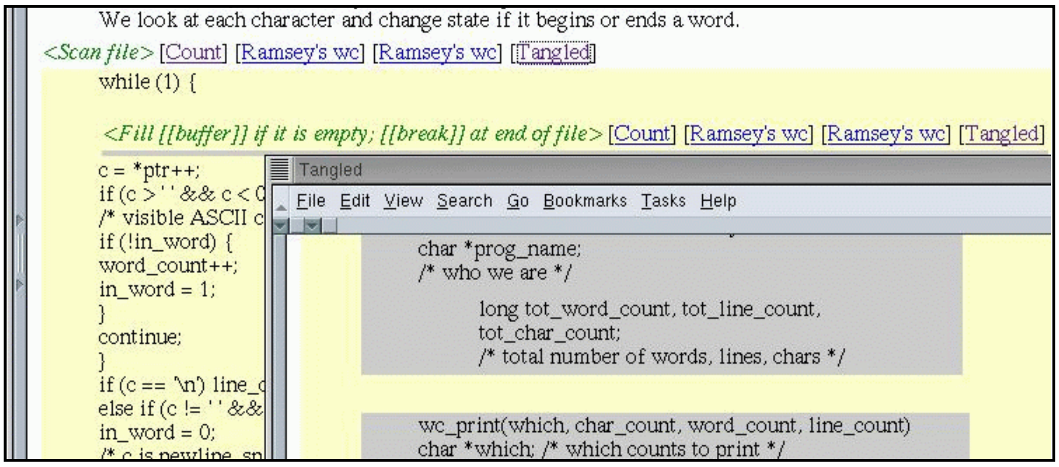

LSE
File Windows Display Theme
Commit Chunk
Edit Chunk
Remove Chunk Reference
Output XML
output paths
Hide Markup
Count
Ramsey's wc
Tangled
Chunk
Scan file
Write statistics for file
Close file
Update grand totals
Variables local to [[main]]
Variables local to [[main]]
Definitions
If a file is given, try to open [[*(++argv)]]; [[contin
Close file
Definitions
Definitions
Variables local to [[main]]
Initialize pointers and counters
Global variables
Global variables
Scan file
Scan file
Fill [[buffer]] if it is empty; [[break]] at end of fi
Fill [[buffer]] if it is empty; [[break]] at end of file
Fill [[buffer]] if it is empty; [[break]] at end of file
Write statistics for file
Write statistics for file
Tree View
Theme Text
Chunk Edit
Create a Chunk
Reference a Chunk
Alter Chunk Details
Include Chunk Reference
Chunk Development

```
\vskip8pt plus1in\penalty-500\vskip8pt plus-1in
The present chunk, which does the counting that is {\tt wc}'s {\em
raison d'\^etre}, was actually one of the simplest to write.
We look at each character and change state if it begins or ends a word.
</chunkref><chunkref chunkID="ch16" content="yes" xref="yes" version="1"
type="code" chunkName="Scan file" static="no" show="yes">while (1) {

<chunkref chunkID="ch17" content="no" xref="yes" version="1" type="code"
chunkName="Fill [[buffer]] if it is empty; [[break]] at end of file" static="no"
show="yes"></chunkref>  c = *ptr++;
  if (c > ' ' && c < 0177) {
    /* visible ASCII codes */
    if (!in_word) {
      word_count++;
      in_word = 1;
    }
    continue;
  }
  if (c == '\n') line_count++;
  else if (c != ' ' && c != '\t') continue;
  in_word = 0;
    /* c is newline, space, or tab */
}
</chunkref><chunkref chunkID="ch17" content="yes" xref="no" version="1"
type="doc" chunkName="Fill [[buffer]] if it is empty; [[break]] at end of file"
static="no" show="no">
```

```
<chunkref chunkID="ch16" version="1" type="code" chunkName="Scan file">while (1)
{

<chunkref chunkID="ch17" version="1" type="code" chunkName="Fill [[buffer]] if it
is empty; [[break]] at end of file"></chunkref>  c = *ptr++;
  if (c > ' ' && c < 0177) {
    /* visible ASCII codes */
    if (!in_word) {
      word_count++;
      in_word = 1;
    }
    continue;
  }
  if (c == '\n') line_count++;
  else if (c != ' ' && c != '\t') continue;
  in_word = 0;
    /* c is newline, space, or tab */
}
</chunkref>
```

Name:	Scan file	ChunkID:	ch16	Version:	1
Type:	code				

(a)



We look at each character and change state if it begins or ends a word.

<Scan file> [Count] [Ramsey's wc] [Ramsey's wc] [Tangled]

```
while (1) {

 <Fill [[buffer]] if it is empty; [[break]] at end of file> [Count] [Ramsey's wc] [Ramsey's wc] [Tangled]
c = *ptr++;
if (c > ' ' && c < 0
/* visible ASCII c
if (!in_word) {
word_count++;
in_word = 1;
}
continue;
}
if (c == '\n') line_c
else if (c != ' ' &&
in_word = 0;
/* c is newline, sp
```

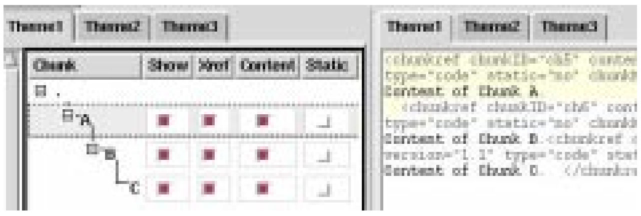
```
char *prog_name;
/* who we are */
        long tot_word_count, tot_line_count,
        tot_char_count;
        /* total number of words, lines, chars */

wc_print(which, char_count, word_count, line_count)
char *which; /* which counts to print */
```

(b) Sample Output

Figure 6. Prototype authoring environment

The *chunk edit pane* at the lower right allows the content of the chunk selected in the tree view panel to be edited. Chunk creation and the association of chunks with themes is supported by the components in the *chunk development panel* at the bottom of Figure 6(a).

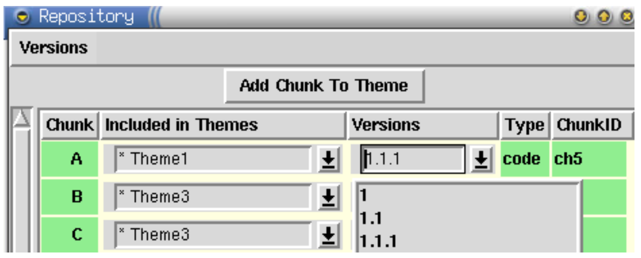
Individual chunks and their evolution are managed separately. Hence, themes do not consist of in-line elaboration of chunks: they are constructed from *chunk references*. This permits a given chunk to appear in more than one theme without duplication. Figure 8 shows part of a CBDE dialog for managing the association of chunk versions with themes. Version 1.1.1 of code chunk A is part of theme Theme1 and earlier versions are available for selection. Similarly, drop-down boxes are also used to browse the list of themes a chunk is associated with. This approach uses less screen space than a tree-based approach.



Chunk	Show	Xref	Content	Static
A	■	■	■	☐
B	■	■	■	☐
C	■	■	■	☐

```
<chunkref chunkID="ch5" conte
type="code" static="no" chunkN
Content of Chunk A
  <chunkref chunkID="ch6" cont
type="code" static="no" chunkN
Content of Chunk B <chunkref c
version="1.1" type="code" stat
Content of Chunk D  </chunkre
```

Figure 7. Chunk display attributes



Chunk	Included in Themes	Versions	Type	ChunkID
A	* Theme1	1.1.1	code	ch5
B	* Theme3	1 1.1 1.1.1		
C	* Theme3			

Figure 8. Chunk version theme association

Once themes have been authored, the next step (see Figure 5) is to select appropriate transformations to produce the desired output products. Themes, specifying document content, are separated from presentation details. This allows considerable flexibility. A given theme may be transformed in different ways—such as a tangled code listing with and without pretty-printing.

Figure 6(b) shows two fragments of output from processing operations carried out on the web shown in Figure 6(a). In each case, XSLT transformations have been used to process the theme elements into HTML. The fragment in the foreground of Figure 6(b) shows part of the processed tangle theme. Only code chunks appear, in the order is that expected by the compiler, but indenting and background shading has been used to illustrate the relationships between the code chunks and the resulting program. XSLT transformations are used to achieve such effects and others such as inclusion of chunk names and links into themes.

The fragment in the background of Figure 6(b) corresponds to the Ramsey theme. Text from a documentation chunk appears at the top, background shading indicates code chunks and nested code chunks are visible. Chunk names have been included for code chunks and links to other references to the chunk are included—each is labelled with the name of the referring theme but suitable icons may also be used.

6 Conclusions and future work

Theme-based literate programming offers significant advantages over conventional LP. Developers are no longer restricted to a single psychological order: themes may be introduced to present information to readers in quanta and orders appropriate to the author's purpose.

The underlying chunk model allows chunks of many types to be defined and used as appropriate to the project in hand. Chunks are stored in a repository and are the unit of version control and management. This permits the reuse of chunks in related themes, each illustrating some significant aspect of the program. The closer relationship between themes supports more effective development practices and helps overcome problems such as debugging.

XML provides a powerful and expressive format for representing TBLP webs. DTDs may be used in conjunction with validating parsers to maintain the integrity of documents at all processing stages.

XSLT transformations allow flexible interaction with other tools. This is important if LP is to be integrated into the software development process. For example, tangled code may be exchanged with IDEs and other themes may be exported to design tools.

Extreme programming, with its emphasis on staying close to the code, seems a potential application area for TBLP. For example, themes describing testing and refactoring might be included. Other application areas include aspect oriented programming and object oriented programming—both experience difficulty in managing simultaneously both system-wide and local constructs.

CBDE demonstrates the feasibility of the TBLP approach. Interactive development tools such as CBDE effectively hide one of the three syntaxes—that of the tool-specific LP web structuring language. The advent of more powerful XML editing components will hide another—that of the text formatting language.

CBDE currently includes only simple authoring tools and we intend to explore further the possibilities for interfaces based on graphical representations of the web. We anticipate that such interfaces will not only better support local editing operations but will also enable better visualisation of the large scale structure of LPs.

References

[1] M. Brown and D. Cordes. Literate programming applied to conventional software design. *Structured Programming*, 11(2):85–98, 1990.

[2] A. Cockburn and N. Churcher. Towards literate tools for novice programmers. In *ACM Australasian Computer Science Education Conference '97. Melbourne, Australia. 2–4 July.*, pages 107–116. ACM Press, 1997.

[3] D. Cordes and M. Brown. The literate-programming paradigm. *IEEE Computer*, 24(6):52–61, June 1991.

[4] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] M. Kay. *XSLT Programmer's Reference*. Wrox, 2000.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.

[8] D. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[9] D. Knuth. *TEX: the Program*. Addison-Wesley, 1986.

[10] D. Knuth. *The Stanford Graphbase: a platform for combinatorial computing*. Addison-Wesley, 1993.

[11] B. Leuf and W. Cunningham. *The Wiki way : Quick Collaboration on the Web*. Addison-Wesley, 2001.

[12] D. Martin, M. Birkbeck, M. Kay, B. Loesgen, J. Pinnock, S. Livingstone, P.Stark, K. Williams, R. Anderson, S. Mohr, D. Baliles, B. Peat, and N. Ozu. *Professional XML*. Wrox Press, 2000.

[13] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8(8):12–26, 1973.

[14] D. Parnas. On criteria to be used in decomposing systems into modules. *Commun. ACM*, 14(1):221–227, 1972.

[15] N. Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, Sept. 1994.

[16] M. Smith. Towards modern literate programming. Project Report HONS 10/01, Department of Computer Science, University of Canterbury, Christchurch, New Zealand, 2001.