

Theme-Based Literate Programming

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science
in the
University of Canterbury
by
Andreas Kacofegitis

Examining Committee

Neville Churcher (University of Canterbury)	Supervisor, Examiner
Robert Biddle (Victoria University of Wellington)	Examiner

University of Canterbury
2002

To my little brother Christophoros and my nephews, Christopher and
Michael Davis.

Abstract

In this thesis we introduce and evolve the paradigm of theme-based literate programming (TBLP). TBLP enhances on the literate programming (LP) model, as invented by Donald Knuth in the early 1980s. TBLP provides a generic model that copes with current and future software development methodologies and practices. We show that through this extended chunk and processing model, XML-based support, and a pipelined document development process, an elegant and powerful system of exposition and development is facilitated. We introduce the concept of themes as a solution to breaking the tyranny of dominant decomposition and show how TBLP can provide equal opportunity perspectives.

Table of Contents

Chapter 1:	Literate Programming	2
1.1	The Comprehension Problem — What?	2
1.2	A Brief Overview of Program Literacy	5
1.2.1	A Birds' Eye View of LP	5
1.2.2	The Chunk	6
1.2.3	Ground-level	7
1.3	Human(e) Order vs. Computer Order — Psychological Ordering	10
1.4	Common Literate Programming Features	11
1.4.1	Chunks versus Macros	11
1.4.2	Pretty-printing	16
1.4.3	Cross-Referencing and Indexing	16
1.5	When is One Considered Literate?	17
1.6	The Propaganda on Programming Literately	18
1.7	Summary	18
Chapter 2:	A Review of Literate Programming Applications	20
2.1	WEB	21
2.2	Noweb	23
2.3	Nuweb	26
2.4	Funnelweb	27
2.5	CLiP	29
2.6	LP Integrated Development Environments	30
2.7	Leo	32
2.8	Spider	34
2.9	Documentation Tools	35
2.9.1	Javadoc	35
2.9.2	Elucidative Programming	38

2.9.3	perlpod	39
2.10	Summary and Comparison	41
Chapter 3: A Review of XML-Based Literate Programming Applications		45
3.1	xmltangle	45
3.2	xml-lit	47
3.3	xLP	49
3.4	LPML	49
3.5	litXML	54
3.6	xmLP	56
3.7	DBLP	58
3.8	Summary	59
Chapter 4: Literate Programming's Limitations		61
4.1	Application Specific Shortcomings	61
4.1.1	Debugging	61
4.1.2	The Three-Syntax Problem	62
4.1.3	Monolithic Files	64
4.1.4	Tangling Creates Tangled Code	65
4.1.5	Scoping	66
4.1.6	Object-Oriented Limitations	67
4.1.7	Primitive Cross-Referencing	67
4.1.8	Limited Output Formats	68
4.1.9	Static Documentation	69
4.1.10	Disparity between Document Editing and the Formatted Document	69
4.2	Model-Centred Shortcomings	70
4.2.1	One Psychological Flow — Limited Readership	71
4.2.2	Asymmetric Processing Model	72
4.2.3	Fixed Chunk Typing Mechanism — Real World Overloading	76

4.2.4	Refactoring — Chunk Version Control	77
4.3	Literature versus Documentation	79
4.4	Summary	80
Chapter 5:	Theme-Based Literate Programming	82
5.1	Theme Weaving	84
5.1.1	The Need for a Tool of Abstraction	85
5.1.2	Theme or Psychological Order?	88
5.2	Multiple Themes: Theme-Paths and Chunk-Nesting	89
5.2.1	Processing Model: Separation of Content and Ordering	90
5.2.2	Enhancing Chunk Composition	96
5.2.3	Chunk Representation	105
5.2.4	The Repository of Chunks	106
5.2.5	Theme Model	107
5.2.6	Processing Model: Blending Weave and Tangle	108
5.3	TBLP Development Emphasises Expression over Development	112
5.4	Equality of Concerns	113
5.5	Multiple Distributed Webs	113
5.6	Summary	114
Chapter 6:	Implementation of the Theme-Based Literate Programming Model	115
6.1	Why XML?	116
6.2	The Literate Document Development Process	117
6.2.1	The Repository — Chunk Composition	121
6.2.2	XML Theme Source Document	125
6.3	The Context-Based Development Environment	128
6.3.1	The Repository Widget	130
6.3.2	The Theme Tree Widget	138
6.3.3	The Theme-View Text Widget	145
6.3.4	Editing	151
6.3.5	The “Chunk Development” Panel	157

6.3.6	The Variant Attribute	164
6.3.7	Theme Functionality	166
6.3.8	Loading and Saving — Repositories, Themes, and Projects	166
6.4	Internal Architecture	168
6.5	Summary	175

Chapter 7: Document Output, Version Management, and Storage Concerns 176

7.1	XML Theme Document	176
7.1.1	An Condensed Processing Model	177
7.1.2	Theme Document Validity	179
7.2	Theme Document Output: Formatting	179
7.2.1	Is XSLT the Only Option? Other Technologies	179
7.2.2	Stylesheet Development	181
7.3	Chunk and Theme Storage	189
7.3.1	Storage Options	190
7.4	The ID attribute	193
7.4.1	Multi-Valued Chunk Identifiers	195
7.5	Version Control — Evolution and Utilisation	195
7.5.1	Themes of Versions	196
7.5.2	Branched Hierarchies of Chunks — Chunk Version Control	198
7.5.3	Theme-Based Version Control	199
7.6	Summary	200

Chapter 8: An Approach to the Practice of Theme-Based Literate Programming 201

8.1	Underlying Aims	202
8.2	TBLP in Software Engineering	202
8.3	Guidelines for the Good	203
8.3.1	Target the Intended Audience	203
8.3.2	Atomic Chunk Mapping Must be Strong	204

8.3.3	Consider Physical Chunk Scope as a Cohesive Measure	204
8.3.4	Use the Consequence of Cohesion To Determine a Chunk's Size	205
8.3.5	Distinguish Comments from the Source Code	205
8.3.6	One chunk — one idea.	206
8.3.7	Chunk first, code later	208
8.3.8	Dissociate intent from implementation	208
8.3.9	Create Self-Documenting Hierarchies	212
8.3.10	Use Smooth Transition Between Levels of Abstraction .	213
8.3.11	Be lazy — write self-documenting chunk names	214
8.3.12	Avoid Temporal Commentary — Reduce Chunk Cross- Coupling	216
8.3.13	Chunkify Programming Language Abstractions	217
8.3.14	Don't Use Implementation-Level Commands as Chunk Substitutes	219
Chapter 9:	Future Work	220
9.1	Extending the Model	221
9.2	Tool Support	223
9.3	Human Computer Interaction	227
Chapter 10:	Conclusion	230
Appendix A:	Thoughts on Literate Programming	233
A.1	Documentation — How Should It Function?	233
A.1.1	LP Documentation versus Source Code — Audience Specific	234
A.1.2	Perspective Simplifies Complexity	237
A.1.3	LP Versus Plain Old Documentation/Comments	239
A.2	An Abstract Process	240
A.3	The Affect of Programming Languages on LP Abstraction .	241
A.3.1	The Power Paradigm of Literate Programming	242
A.3.2	Psychological Scope	243

A.3.3	Medium of Focus	248
A.3.4	A Super-Language	249
A.3.5	LP as Program Description Language	249
A.4	Unordered Programming Languages vs. Psychological Ordering	251
A.4.1	Are Programming Languages Literately Enabled? . . .	251
A.5	Mis-direction of Focus?	253
Appendix B: TBLP Methods in Software Engineering		254
B.0.1	A Layered Approach to Themes	255
B.0.2	Cross-Sections of Layers	257
B.0.3	Flexibility of Approach	259
B.1	A Set of Example Themes	260
Appendix C: Example Literate Programs		269
C.0.1	Example NowebProgram	270
C.1	An example of <code>noweb</code>	270
C.1.1	Counting Words	271
C.2	Personal Greeter	283
C.3	Scoping (In)capabilities	285
C.4	Identifier Cross-Referencing (<code>nuweb</code>)	287
Appendix D: Theme Enabling Literate Tools		288
D.1	Supporting Multi-Themed Requirements with a Traditional Literate Tool	288
D.1.1	The Journey to Enlightenment	288
D.1.2	<code>Noweb</code> as a Development Platform	289
D.1.3	The Bubblesort Theme-Set	290
D.2	Requirements for an Alteration to <code>Noweb</code>	291
D.2.1	Model Enhancement of <code>Noweb</code> — The Initial Attempt .	294
D.2.2	A Summary of the Initial Implementation	298
D.3	Initial Model++: The Displacement Model	299
D.3.1	Implementation of an Enhanced Model	299
D.3.2	A Summary of the Second Implementation	303

D.4	Does Chunk Displacement Suffice?	305
D.4.1	Duplicate Chunks	306
D.4.2	Non-fixed Chunk Types — Higher Order Documentation	308
D.5	Theme 1 Example: Bubblesort	314
D.6	Theme 2 Example: Bubblesort Evaluation	321
D.6.1	Test Environment	324
D.7	Theme 3 Example: Bubblesort Algorithm	326
D.8	NowebTheme Converter	328
D.9	NowebDisplacement Theme Converter Overview	335
D.10	NowebDisplacement Theme Converter Elide	338
D.11	NowebDisplacement Theme Converter Displace	344
Appendix E: Theme-Based Version Control		353
E.1	Multi-chunk version control	353
Appendix F: Reverse Engineering		356
F.1	Psychological Scope — Its Affect on LP	356
F.2	Object Orientation	358
F.2.1	Class-Level Chunking	359
F.2.2	Method-Level Chunking	359
F.2.3	Abstract Class Chunking	360
F.2.4	Interface Chunking	361
F.2.5	Try/Catch Chunking	362
F.2.6	Sub-Method Level Chunking	364
F.3	Imperative Languages	364
References		367

Acknowledgments

I'd like to thank my supervisor, Dr. Neville Churcher, for his friendly and willing support, influential ideas and ever-available advice. I'd also like to acknowledge Ben Schmidt for acting as my (in)sanity checker during the initial stages of this thesis. Malcom Williams for his interest and the many interesting and thought-provoking discussions that we shared. James McNeill deserves special mention for his help in the reasoning of the CBDE's editing environment (but moreso for the loaves of bread that he supplied me with). Yen-Rong Sun (Dan) for his interest and insightful comments and importantly, cheerful personality. Dr. Greg Albertson for his advice and eagerness to help. Stacey Mickelbart repaired the broken English that was this thesis. And finally Nadine Fea for her support and understanding throughout (if not solely for her coffee making abilities).

This is a good idea to be sure; if you don't understand it, then I haven't explained it well. If you don't like it, then you don't understand it. —
Andreas Kacofegitis June 12, 2002 13:03:28

Chapter I

Literate Programming

... my enthusiasm is so great that I must warn the reader to discount much of what I shall say as the ravings of a fanatic who thinks he has just seen a great light. — D.E. Knuth [45]

Donald Knuth coined the term literate programming (LP) in the same manner as the naming of the 1970’s phenomenon structured programming, which encouraged the development of readable and maintainable programs. Much as no-one would admit to writing an unstructured program, he dared anyone to admit writing an illiterate program.

In this chapter, we define literate programming; what it means to be ‘literate’. We define literate programming by describing:

- the mechanical process of developing a literate program.
- what LP offers over other programming paradigms.

1.1 The Comprehension Problem — What?

“Even first hand instruction doesn’t help, as Chaim Weizmann found when he took a long Atlantic crossing with Einstein in 1921: “Einstein explained his theory to me every day,” Weizmann said, “and soon I was fully convinced that he understood it.” — [8]

“Program comprehension is the process of acquiring sufficient knowledge about a software artefact so as to be able to successfully accomplish a given task.” [15]

Literate programming facilitates the development of programs in an expression and order that a programmer would use to explain them to a fellow programmer, colleague, or maintainer: the target audience may vary. In this way, literate programming aids in the comprehension of software.

Maintenance programmers spend about half their time studying source code and related documentation [61], and 30–90 percent of software expenditure, over the lifetime of a system, is spent on software maintenance. It makes sense to seriously consider how to best communicate what the original programmer was thinking.

Software development is a difficult task. According to Brooks [10], it requires the programmer to possess both general semantic knowledge (program language and semantics) and domain knowledge (knowledge about the area of program application). The transfer of domain knowledge from the real world to source code must be communicated to the reader. Experimentation has suggested that programmers with good domain knowledge are better able to understand a program than programmers with general semantic knowledge [15].

If we assume a programmer’s instructions to the computer are complete and correct, and therefore reliable, it must follow that mis-comprehension can only occur because a reader is unable to understand what was previously communicated to the computer. This is likely due to the loss of domain knowledge information, rather than general semantic knowledge. Figure 1.1 exemplifies this scenario. The reader is unable to acquire enough domain knowledge to accomplish his task, causing a break-down in communication. If, however, the mental model of a software artifact is successfully generated, program comprehension is successful. Note that programmer A and programmer B are separated by a period of time — LP does not attempt to solve the immediacy of communication. Programmer B therefore could well be programmer A some time in the future attempting to comprehend his own work.

The ability to comprehend software is dependent upon the *efficiency of software comprehension*. We determine that the efficiency of software comprehension is based on two criteria:

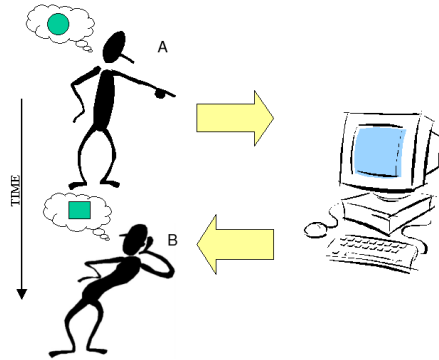


Figure 1.1: Programmer A expresses the problem domain to the computer. Because the explanation of the problem domain is directed towards the computer, a loss of domain-specific information occurs — Programmer B miscomprehends the instructions.

1. the speed of software comprehension, and
2. the ease of software comprehension.

Much as the analogy of the shortest distance between two points may not be the quickest to travel (by land — a boggy marsh may prevent this), so does it stand with comprehension. The most concise of program source code is not always the best understood. A map is often used as a navigational and distance measure even though the destination may be visible. So is it the case with software development — although the functionality of the software is apparent, further, more abstract instruction is required as to its method of implementation. The point? Program source code itself is not always its best descriptor (or documentation); contrary to what others may suggest [62].

How can literate programming aid in the speed and efficiency of comprehension? Let us answer this question by considering how literate programming attempts to aid software comprehension.

1.2 A Brief Overview of Program Literacy

Knuth provided the term *psychological order*, based on the idea that a computer program should be aimed at humans rather than the compiler that produces the program. Realising that the ordering of source code for a computer is different to the ordering of source code for a human, he developed the WEB literate programming tool (discussed in Section 2.1 on page 21 of Chapter 2), giving programmers the ability to promote comprehension by both of these audiences.

Literate programming fundamentally facilitates program comprehensibility by allowing the programmer to associate documentation with code. Furthermore, it allows the author of a program to place these segments of documented code in an order deemed pertinent to a reader's understanding.

Literate programming is not a panacea for application development. It does not necessarily make programming easy, nor does it attempt to. The literate programming process makes one think — hard. Literate programming is not a substitute for good design: a bad literate program will be worse than its non-literate equivalent. Literate programming is not a programming language per se, but rather, a technique — an approach to the practice of software development.

Having sufficiently lowered the reader's expectations, we can raise them again by noting that literate programming promotes the development of quality software. By virtue of the imposed overhead of thought, *good quality* literate programs ensue.

1.2.1 A Birds' Eye View of LP

What exactly is LP? How does LP aid in software comprehension? This section considers the semantic and physical capabilities that must be offered by a literate enabled environment.

To allow programming literacy, the following requirements must be met:

granularity of expression: It must possible to divide the program source code into discretely large or small units.

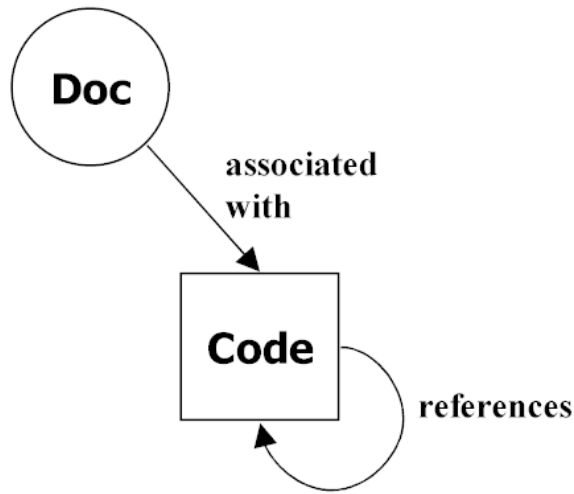


Figure 1.2: A documentation chunk has a relationship with a code chunk. This relationship is implied. Together, they form an atomic chunk. A code chunk may reference other code chunks.

multiple orders of exposition: It must be possible to present the program source units in at least two orderings, so that units are presented in a manner, or order, that promotes comprehension of the intended audience.

abstract representation: It must be possible to document each program source unit such that it is explained in a manner that satisfies the reader’s comprehension.

1.2.2 The Chunk

Although terminology differs among literate tools, we choose to adopt the definition of a segment of code or documentation as used by LP tool developers such as Norman Ramsey — a *chunk*¹.

A literate program is composed of a number of chunks. A code chunk is accompanied by a documentation chunk. A code chunk may reference any number of other code chunks. Together, a documentation and code chunk

¹ Other tools, some of which are covered in Chapter 2, use terms such as macro, module, scrap, or section.

for an atomic chunk, therefore representing the same unit of abstraction. Figure 1.2 on the preceding page illustrates this relationship. The referencing ability of code chunks builds a tree-like structure that, when processed in a depth-first order, outputs the source code. This relationship is implied from the lexical ordering of documentation and code chunks in the source web.

atomic chunk

A code chunk receives a label that describes the purpose of the code chunk. That is, a chunk’s label describes what it does, e.g., `<<variable declarations and initialisations>>` or `<<calculate grand total>>`. At initial glance, a chunk may seem similar to a function or method; however this is not always the case, as discussed further in Section 1.2.3.

A documentation chunk is able to explain, in further detail, how the programmer arrived at the contained segment of code, why it was implemented the way it was, alternate avenues that may have been explored, and other pertinent information aimed at the intended reader.

In literate programming, documentation is commonly written before the code is developed. Therefore, a documentation chunk generally physically appears immediately before a code chunk. This is termed as a *prevenient* approach to programming practice [56]. This is opposed to a post-hoc method of documentation; that is, documenting what the code is doing, rather than what it is meant to do.

A code chunk may reference, or include, any number of other code chunks. A code chunk may not, however, reference itself. The lexical ordering of code chunks throughout the document is what determines the “psychological flow” of a program.

1.2.3 *Ground-level*

In order for a literate tool to develop literate programs, it must possess fundamental capabilities/functionality. We describe, using a minimalist approach, what a literate programming tool must offer. A maximalist approach would warrant a more speculative perspective, something employed by tools such as **Leo** (see Section 2.7).

hierarchy of code chunks: It must be possible to develop an implicit hierarchy of chunks, which is generally performed by referencing defined

chunks of code.

In order to build such a hierarchy, it must be possible to reference a chunk from within the body of another chunk. For example, the following chunks (extracted from the `greet.nw` Noweb file in Appendix C.2) without their documentation or code content form:

```
<<*>>=  
  <<greet process>>  
  <<greet declarations>>  
<<greet process>>=  
  <<ask what the user's name is>>  
  <<read name from input>>  
  <<print greeting>>
```

Figure 1.5 on page 13 presents the hierarchy derived by following the references made by each chunk.

code chunk implementation: code chunks can be defined anywhere throughout the literate program.

chunk labelling: In order to reference a code chunk, it must have a unique identifier. This unique identifier has traditionally been the chunk's name or label (our tool differs by introducing a separate chunk attribute for chunk referencing — see Section 7.4).

granular code chunks: Chunks of all sizes must be allowed. As an example, it should be possible to encapsulate the entire program source code within *one* chunk². Conversely, it should be possible to generate chunks of zero characters in size. While these extremes of granularity may not be considered good practice, the “normal” granularity of a chunk would occur somewhere in between.

² by no means considered good practice — discussed in Chapter 8 on page 201

accompanying documentation: It must be possible to accompany a code chunk with documentation. No restrictions should be placed on the size of the documentation chunk.

Childs [17] provides a list of requirements that he considers *imply* the definition of a literate program. These add context to our formal definitions and therefore aid in the understanding of what literate programming is. A selection of the list is presented:

- Documentation should include an examination of alternative solutions and suggest future maintenance problems and extensions.
- Documentation should include a description of the problem and its solution.
- The system should be presented in an order of logical consideration, rather than syntactical constraints.

When chunks of the program are presented in an order aimed towards the human reader, the psychological order of the program is expressed. The ability to nest code chunks and the ability to attach to them supporting documentation is what differentiates literate programming from other programming and documentation techniques. Without this functionality, a tool is *not* literate enabling (or enabled).

Implicit Chunk Typing

Note that chunks are not explicitly typed. A chunk's type is implied from a delimiter, or the previous chunk's terminating character. For example, **Noweb** uses the '@' character as a documentation chunk delimiter (and code chunk terminator) and the '<<>>' characters to begin a code chunk.

Documentation Changes With Code

Documentation and code chunks form an atomic unit; both normally are an expression of the other, implemented in a different level of abstraction

however. The atomic chunk is contained in the same source file, in a central physical location. The physical proximity of documentation chunks to code chunks ensures the likelihood of simultaneous update to both chunks when an edit occurs. This decreases the chance of obsolete or inconsistent documentation or code.

1.3 *Human(e) Order vs. Computer Order — Psychological Ordering*

We know, from Section 1.2.2, that literate programming combines documentation with code. We also know that a code chunk may reference a number of other chunks.

Literate programming allows the programmer to express the ordering of code chunks to the human differently than he would express this ordering to a compiler.

Figure 1.3 presents a flow diagram that abstractly represents the nature of a literate program. Each named rectangle represents a code chunk in a literate program³. (The name of each rectangle is the name of the chunk.)

Two orderings of chunks exist:

weave ordering The top-down, or sequential, ordering of chunks, where order is maintained by the connecting dashed line from one chunk to the next, occurs in the order in which chunks are displayed in the literate document. Effectively, this is derived from the lexical ordering of chunks in the source file. This is the “psychological order” suggested by Knuth — the program directed to the human. The process of generating the literate document is commonly known as **weaving**.

weave

tangle ordering The order in which the computer will receive these chunks — the traditional order. This is indicated by the solid line that begins from the “root” (rectangle labelled ‘*’) chunk. The root chunk in this example happens to be the last chunk in the literate program. Following the thick directed line from chunk to chunk presents a different chunk

³ see Appendix C.2 for the **Noweb** source of this LP.

order to the compiler. The chunks are output to the source code file. This process is known as **tangling**.

tangle

The process of tangling and weaving is shown in Figure 1.4. The literate source can be found in Appendix C.2.

The ability of a code chunk to reference a number of other code chunks essentially generates a tree-like structure of chunks. Although Figure 1.4 represents the *order* of traversal of the chunks of code, Figure 1.5 more accurately reflects the *nested* nature of code chunks. It illustrates that the body of a code chunk may be composed of other code chunks.

Note also that each edge in the tree is a reference to a chunk — chunk definitions do not occur more than once in the literate source. A chunk reference can, however. The referencing of chunks is known as composition by reference.

Figure 1.6 shows the nesting of chunks in greater detail. Note that `<<Process all the files>>` contains references to several other chunks. Each chunk is enclosed in angle brackets. Note also that program source code may be interspersed throughout a code chunk (marker ‘7’ indicates the content of the `<<Process all the files>>` chunk).

Summarising, an LP tool generally offers the ability to tangle and weave a literate program. The tangling process produces source code. The weaving process produces the literate document.

1.4 Common Literate Programming Features

1.4.1 Chunks versus Macros

The chunk is described in Section 1.2.2. Some literate programming tools, such as WEB and FunnelWeb (examined in Chapter 2) offer *parameterised* macros. These are simply chunks that accept arguments. The arguments are represented in the chunk body by substituting the argument name with the argument value within the chunk definition. These are similar to a C macro that is expanded in the source code by the C pre-processor. The most basic of chunks is the one that accepts no parameters. Unless otherwise stated, the word chunk throughout the rest of this text shall refer generically to

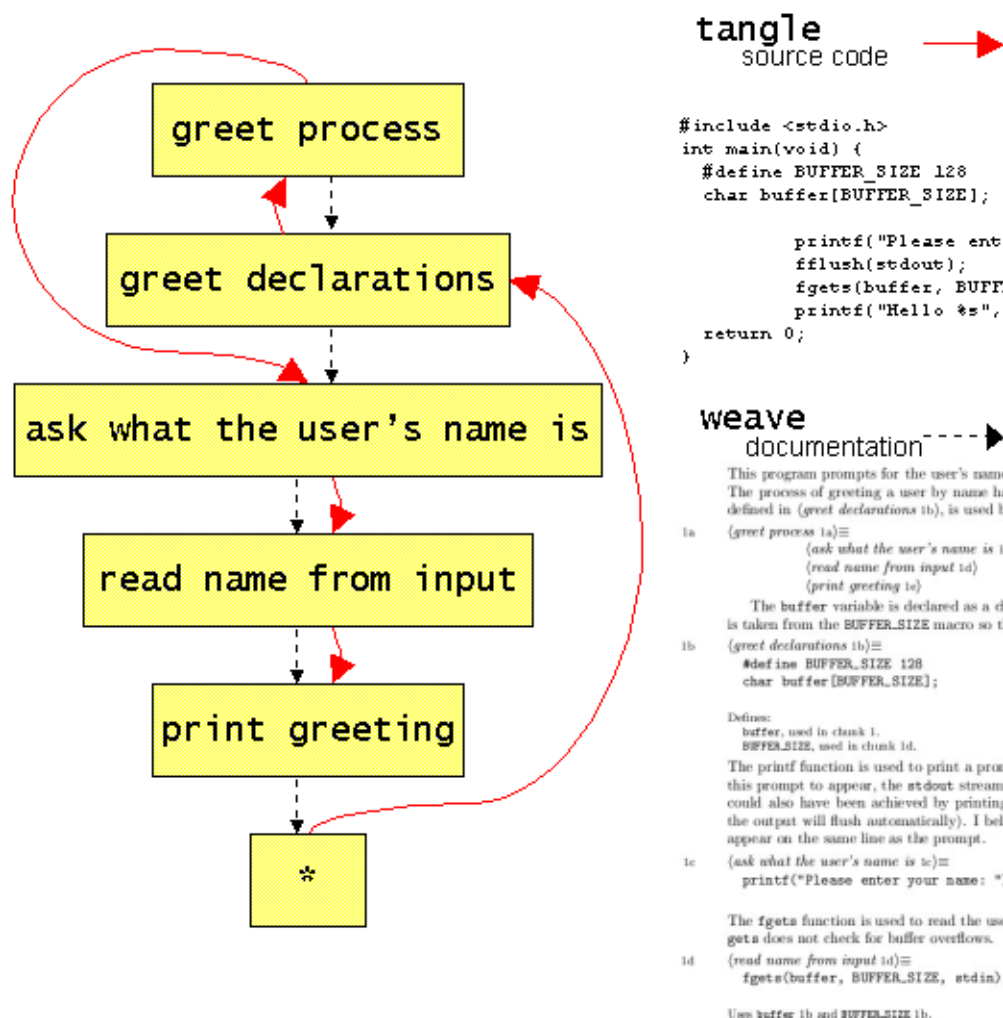


Figure 1.3: The helloworld LP. Weave/psychological order (dashed arrow) and tangled order (solid arrow) are shown on the left. Upper right is the tangled source. Lower right is the woven document.

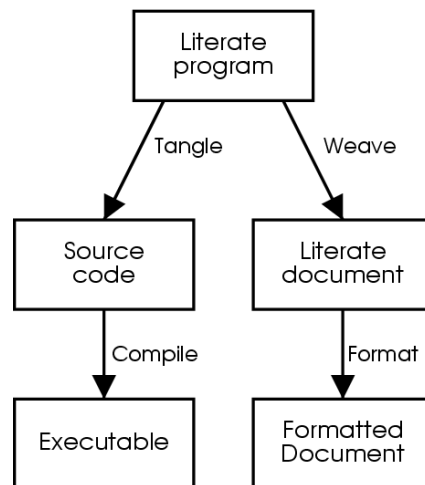


Figure 1.4: The document development process.

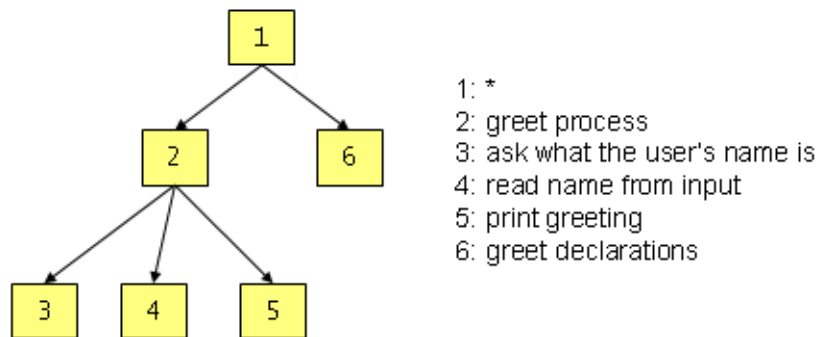


Figure 1.5: Code chunks may be nested such that a natural tree hierarchy is generated.

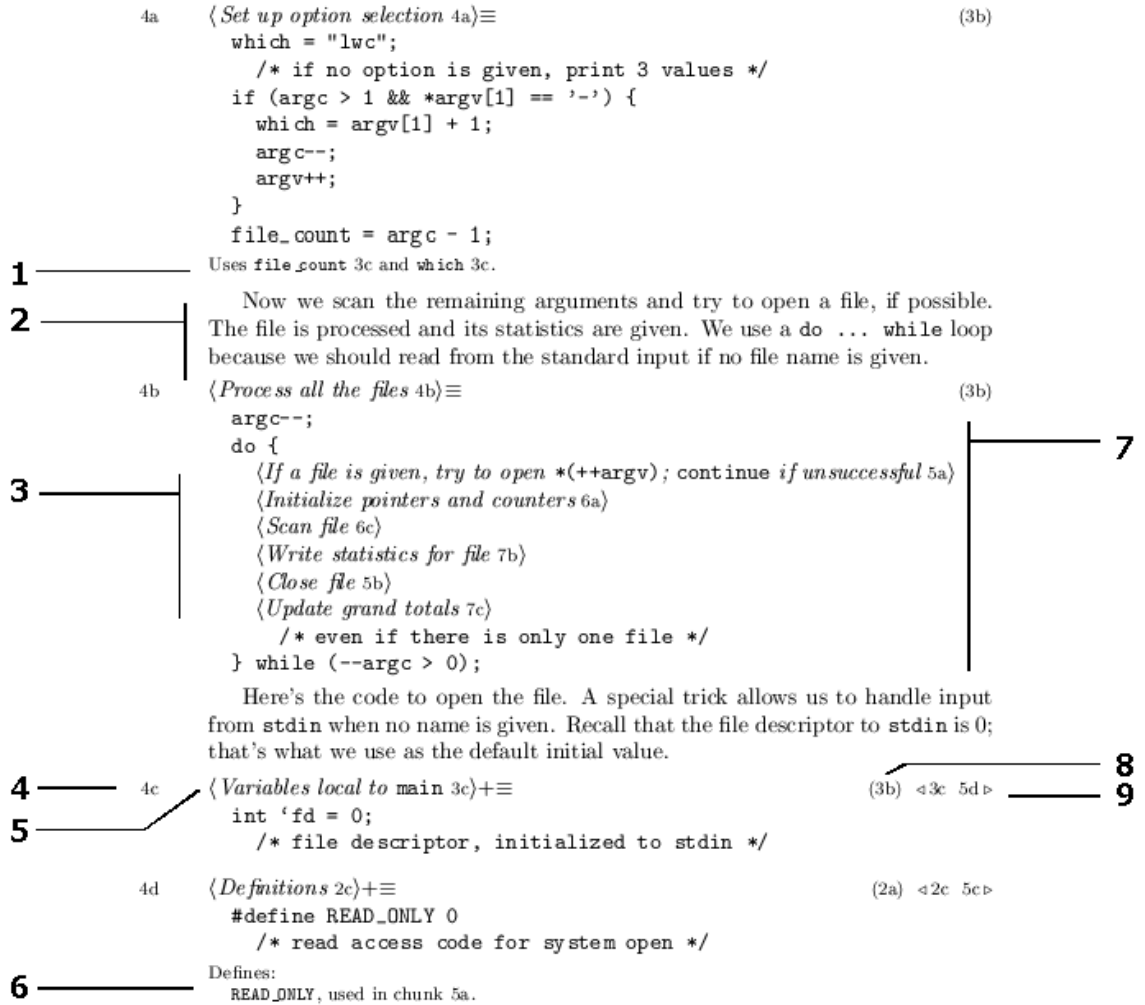


Figure 1.6: A woven extract of Ramsey’s `wc` LP. The markers illustrate features employed by some literate programming tools. ‘1’ and ‘6’: cross-referencing of the uses and definitions of identifiers. ‘2’: demonstrates a documentation chunk. ‘3’: demonstrates chunk referencing, ‘4’: shows a chunk’s reference marker, ‘5’: a chunk’s name and cross-reference to where its initial definition occurs, ‘7’: illustrates the body of a code chunk, ‘8’: cross-reference details indicating in which chunks the chunk is referenced, and ‘9’: cross-references to an additive chunk’s definitions.

both parameterised and parameterless chunks. When specifically addressing parameterised chunks, we shall use the term *macro*.

Additive Chunks

Many literate tools allow a code chunk to be augmented throughout a program. The resulting code chunk is a concatenation of all like-named code chunks in order of implementation. Thus, code chunks such as the `<<Variables local to main>>` chunk in Figure 1.6 on the preceding page are combined to create one segment of source code, and are termed as additive. Note the ‘3c’ identifier after the chunk’s name which is a cross-reference to the chunk’s initial definition. Also note the ‘+=’ symbols, which indicate that a chunk is additive. The chunks that comprise `<<Variables local to main>>` can be found in 3c and 5d as denoted by the cross-reference information for that chunk (indicated by marker ‘9’ in Figure 1.6).

Labels and Abbreviated Identifiers

Using a chunk’s name, one is able to cross-reference or reference a chunk in the body of another chunk. Thus, the name of a chunk is a chunk’s identification (in the case of additive chunks, multiple chunks can possess the same name). Furthermore, the chunk’s name is displayed in the resulting literate document (as illustrated by marker ‘5’ in Figure 1.6).

Some literate tools provide the ability to use abbreviated identifiers so that if the abbreviated name of a chunk is unique, the literate tool will resolve the reference. For example, a chunk may be named “alphabetically sort all elements”. Referencing this chunk may be possible by simply entering “alphabetically sort...” if “alphabetically sort” uniquely resolves to the intended chunk. Note the points of ellipses, which commonly indicate that the identifier is abbreviated.

1.4.2 *Pretty-printing*

Language-aware tools such as `WEB` offer pretty-printing of code. Pretty-printing is a point of debate in the literate programming⁴ community. Many argue that pretty-printing distorts source code such that it becomes unrecognisable and difficult to read. Fox [26], in particular comments that pretty printing can be far removed from its initial presentation in the source file that it becomes unrecognisable. Others argue that pretty-printing offers the ability to accentuate important parts of the source and thus makes it easier to comprehend. Many of these issues are influenced by the nature of the pretty-printer (some pretty-printers, for example the `a2ps` package simply highlights key words, rather than combining and replacing symbols).

It is our opinion that pretty-printing, while making source code visually pleasing, suffers because it is an automated process — which also happens to be one of its strengths. As such, it is unable to highlight semantically important areas of interest.

Chunks are abstract entities that capture a point of interest in the program. They tend to be smaller segments than those of methods or functions. With the help of an abstract and definitive chunk name and perhaps supporting documentation, it is usually unnecessary to use pretty-printing to highlight important areas of code. Pretty-printing is useful when a large amount of code is presented in sequence; it is then able to uniformly represent syntactic features of the programming language.

1.4.3 *Cross-Referencing and Indexing*

Cross-referencing of chunks facilitates navigation throughout the document. In a woven document, it can be used to follow the computer-oriented order rather than the psychological flow imposed by the author. Figure 1.6 illustrates chunk's `<<Variables local to main>>` cross referencing details. Marker '4' indicates that '4c' is the chunk's referencing identifier. Marker '8' indicates in which chunk's `<<Variables local to main>>` is referenced. Locating '3b' will reveal this information. Marker '3' illustrates a number of

⁴early mailing lists contained lively debate

chunk references. Each chunk reference displays the referenced chunk’s name and a cross-reference to its place of definition.

The declaration and use of identifiers allows the programmer to identify the scope and manipulation, or use, of an identifier. Language-aware tools usually generate this information automatically. Language-unaware tools rely upon the author of the program to manually signal the declaration of an identifier. The tool then uses a heuristic approach to find the uses of each identifier.

For example, Figure 1.6, markers ‘1’ and ‘6’ show the definition and uses of identifiers, respectively.

1.5 When is One Considered Literate?

What should a literate program look like? How should it read? What must be communicated by the author to the audience? In Chapter 8, we examine these questions and propose a number of guidelines for writing ‘good’ literate programs. This section considers how a literate program should read, rather than how it should be written.

Undue effort on the presentation of the documentation should not detract from the quality of the program.

Bentley [6] and Knuth assert that a literate program should resemble a novel-like entity that can be read from cover to cover; a work of literature; something that can be read whilst sitting in a “comfortable chair” whiling the evening away. Certainly the opportunity is available, through literate programming, to develop interesting documentation; however, the emphasis should ultimately lie in the quality of the program being developed. Effective documentation is necessary and important to the comprehension of a program, but should not be at the expense of the program’s quality. The two are intimately related; however, the program is usually the ultimate expression of concept⁵, and documentation the supporting agent.

⁵ unless, for example, a dissertation is the goal

We therefore suggest that a literate program is one that presents the reader with the correct domain model such that he is able to understand and extend the program effectively and efficiently. This would imply that in order to correctly transfer domain knowledge, a literate program is not necessarily read in its entirety. It could be (and is most likely) read as a reference source. For example, the reader may need to alter one particular part of the program. He therefore would need to know how that part of the program works, and how it affects the program as a whole. Enabling this understanding would allow the programmer to effectively and correctly alter the program — without reading the *entire* literate program.

In summary, we consider literate programming to be the act of developing programs using psychological ordering, cross-referencing, and supporting documentation to capture and present information that helps the reader affirm a complete mental model in order to alter, enhance, and/or understand the program.

1.6 *The Propaganda on Programming Literately*

Little research has been performed on the effectiveness of LP in real-world scenarios. Although research suggests [71, 20] suggest that LP is an effective means for software development, and can help to reduce the learning curve required to understand programs, the wider programming community has been reluctant to adopt LP. This in itself does not suggest that LP is an ineffective programming medium. It does, however, convey the need for supporting tools and perhaps more concrete research on its effectiveness as (1) a software development medium and (2) a medium of comprehension.

1.7 *Summary*

In this chapter, we have defined literate programming; at both the physical and semantic level. We have considered its importance to the problem of software comprehension. The rest of this dissertation is focused on delivering and realising our enhancement on the LP paradigm. Our work in this area evolves the concept of theme-based literate programming (TBLP).

Before introducing TBLP, and its physical realisation in Chapters 5 and 6 respectively, we consider existing literate and documentation tools — Chapter 2 focuses on common literate tools, Chapter 3 focuses on XML-based literate tools — in order to determine both their strengths and weaknesses. Chapter 4 examines these weaknesses.

From the analysis of LP’s weaknesses and shortcomings, we propose, in Chapter 5 a new set of models for literate programming, and thus, establish the paradigm of theme-based literate programming. We also introduce a framework for theme document development. Chapters 6 and 7 demonstrate and discuss the implementation of our TBLP model.

Finally, Chapter 8 presents a styleguide for theme-based literate programming.

Our work on TBLP extends that presented in [39]⁶.

⁶ Accepted

Chapter II

A Review of Literate Programming Applications

In this chapter, we introduce the reader to literate programming and documentation tools and the functionalities that they offer. The process of literate program development was discussed in Chapter 1. Understanding this process aids one’s appreciation of the approach to developing literate programs that the tools covered in this chapter enable.

A range of literate programming applications exist (an up-to-date list can be found in [49]). There have been many attempts to improve upon Knuth’s prototypical **WEB**. We first examine **WEB** and then investigate the offerings made to the literate programming paradigm by these other tools. Specifically, **Noweb**, **Nuweb**, **FunnelWeb**, **CLiP**, **Jaba**, **Leo** and **Spider** are investigated. We also investigate documentation tools. Namely, **Javadoc**, elucidative programming, and **Perlpod** are examined.

In choosing a subset of literate programming tools, the following factors were considered: originality of approach, contribution to LP, specific architecture, and functionality offered. We remain aware that other good LP tools do exist.

Knuth wrote in his introductory paper on literate programming [45] that LP is not suitable for the masses. Specifically, he targeted the computer scientist as a potential user of literate programming. Many of the tools that were based on **WEB** focused on a different user: the programmer. This should be kept in mind whilst reading the comparisons of available tools. Tools may make “improvements”, but many of these improvements are a simplification of **WEB**. Through simplification, however, functionality can be reduced.

WEB is introduced first because it archetypically encompassed many of the functionalities available from other LP tools. Thereafter, no order of

significance is given to the ensuing discussion of each tool.

2.1 WEB

WEB was developed by Donald Knuth in 1983 [44] following several prototype systems since 1981. WEB itself is also considered a prototype tool. It is used to document programs written in the Pascal programming language and uses \TeX ¹ as its document formatter. Knuth, however, suggested that potentially any combination of document formatting language and programming language could be used. Although prototypical, WEB contains many of the concepts common in today's LP tools. Anecdotally, it was developed before the term became a popular one for the internet.

Knuth intended the editing of a literate program to be the sole means of program development. To enforce this intention, he made the by-product of WEB's tangling process (the program source code) essentially unreadable; identifiers are shortened and altered to upper-case equivalents, all comments are removed from the program source code, and entries of the relative positions of the source code in the literate program are added. This unreadable code helped enforce what Knuth saw as one of the ideals of literate programming — the literate program provided the sole means of access to program development. As such, the programmer is forced to develop the literate program without referring to the raw program source code at all. We do not adhere to this point of view and address it in Section 1.5.

WEB supports the use of 'CHange' files. These are used to facilitate the update and maintenance of a program. The idea, which has since been disregarded in LP tools, warrants special mention. A CHange file remains external to the literate program. Updates to program source code may be entered into this file. Upon tangling and weaving, these changes are reflected in the ensuing documents. Multiple CHange files may exist and may be used conditionally. It is therefore possible to develop a single program that can be adapted to run under several different environments by using multiple

¹ As an aside, Knuth also wrote the programs for \TeX and METAFONT entirely in WEB, eventually publishing them in book form [46], [47]. These are probably the largest literate programs ever published.

CChange files. Supporting tools such as webMERGE exist, which aid the merging of CChange files with the literate source.

Brown and Childs [11] realised CChange files’ importance when developing their prototypical GUI (graphical user interface) environment for LP. They considered their use as ‘good’ practice, as did Sewell [77].

WEB’s functionality is driven by the use of directives (27 in total²). Of these directives, WEB provides two levels of macros. We define these as:

1. non-parameterised
2. parameterised

Both macro types may be referenced as abbreviated identifiers (see Section 1.4.1). Non-parameterised macros are akin to the ‘chunk’, as discussed in Section 1.4.1. Parameterised macros possess the added ability to process an argument, and present this argument anywhere in the body of the macro. In offering program macros, WEB provided support to compensate for shortcomings in the Pascal programming language [41, 45]:

... it is impossible to have a PASCAL array whose bounds are ‘0.. $n - 1$ ’, or to write ‘ $20 + 3$:’ as the label of one of the cases in ‘*case $x + y$* ’: WEB’s numeric macros make it possible for TANGLE to preprocess such constraints [45].

Being Pascal-aware (possessing syntactical and semantic knowledge of Pascal) allows WEB to:

- pretty-print and type-set program source code (code chunks), and
- automatically index and cross-reference identifiers in code chunks.

Being language specific, however, prevents the programmer from developing programs in other programming languages. Also, because T_EX is the document formatter, no other formatter may be used.

²This abundance of directives, however, drove the development of the Noweb LP tool (Section 2.2)

Pretty-printing is performed automatically, and although there is some basic functionality to override aspects of this pretty-printing, it is otherwise impossible to alter the nature of the layout and presentation.

WEB's restrictions on the programming language and document formatter prompted the development of a family of LP tools. Some were based directly on WEB and used a fixed formatting and programming language. WEB directly affected the development of CWEB, and later FWEB, (each of which cater to an increasing variety of programming languages — C and (later) C++, Fortran, and Ratfor). Each implementation, however, was restricted to a set of programming languages.

Although WEB's output is limited to L^AT_EX-based documents, the latest CWEB supports output to PDF (portable document format), and navigatable cross-referencing is achieved through hyper-links.

Some programmers were intolerant of language restrictions. This sparked the creation of language-neutral, or language-unaware, general purpose literate programming tools, such as Noweb, FunnelWeb, and Nuweb.

2.2 Noweb

Noweb [70] is a popular LP tool. Its utilisation, along with CWEB, in GUI tools such as Leo (covered in Section 2.7) and LyX (see Section 2.6), is perhaps, a testament to its popularity. It was developed by Norman Ramsey and derives its name from an attempt to overcome some of what Ramsey considered to be shortcomings of the WEB family [68] — hence 'no web'. Noweb addresses the following WEB deficiencies:

- language dependencies,
- a complex array of directives — too much effort was required to master WEB,
- unadaptable pretty-printing, and
- the inability to use L^AT_EX constructs.

Fundamentally, Ramsey’s argument was that **WEB** makes the exploration of literate programming difficult [67]. He did not dispute the effectiveness of **WEB** as an LP tool, but rather as an explorative aid.

Noweb provides a solution to these shortcomings by providing programming language independence, a smaller set of directives, and no enforced pretty-printing of source code. Perhaps coinciding with the attempt to simplify and minimise the set of directives, **Noweb** offers an extremely extensible pipeline architecture.

Noweb’s pipeline architecture follows the UNIX pipeline model. The pipeline approach allows the output of one process to be treated as the input of another process. This architecture allows ‘expert’ users to create *filters* through which **Noweb** code is passed and altered. Such filters may take the form of specific language cross-referencing agents (Chapter 5 discusses the development of a set of **Noweb** filters as an attempt to enhance the LP model; Appendices D.9 – D.11 contain the source code for these filters). Pretty-printing is also enabled through use of a filter. Indeed, Pretzel [30] is a tool that has been specially adapted to offer such functionality³. This pipeline architecture provides an extremely flexible manner in which to add functionality to **Noweb** whilst keeping the directives used in literate program source code minimal.

A filter may be developed in any language (awk and Perl are examples), so long as it is executable on the necessary operating system, and obeys the relatively simple syntactic constraints set by **Noweb** [66]. **Noweb** distributions come complete with a standard set of filters, two of which are:

disambiguate: allows abbreviated chunk identifiers.

elide: removes a code chunk from the literate document.

³ Pretzel itself is a tool that can be used not only in **Noweb** implementations, but as a general purpose pretty-printer. Given a grammar and layout rules for that grammar, Pretzel generates a module that is able to process the input source code file and output a pretty-printed L^AT_EX version. With some difficulty, it is possible to extend Pretzel to cater to formatting languages such as Troff. Implementations currently exist for C, Java, and Pascal.

Noweb permits identifier cross-referencing; it provides a framework and sample filters for a wide range of languages. This cross-referencing can be performed either manually (through use of the `@_def` directive), or automatically via a filter (that automates the inclusion of the `@_def` directive). A language-independent heuristic is used to locate occurrences of known identifiers. Using such a heuristic can produce falsely-found identifiers, while others may not be found at all. For example, **Noweb** will consider the two identifiers `var1` and `var1-temp` to be the same, which is incorrect.

\LaTeX is generally used as the document formatting language; however, Troff, HTML, or \TeX are also supported. Although a diverse set of formatting languages may be used, the set is still fixed, thereby rendering the use of a markup language such as XML impossible.

Noweb inherently supports multiple languages because it has the ability to specify which code chunk is the root chunk when tangling the **Noweb** file.

Noweb does not support multiple input files. However, this limitation can be circumvented utilising the document formatter's file inclusion capabilities. For example, use can be made of \LaTeX 's `\include` directive to include other **Noweb** literate program segments. This method has restrictions, however. Although it is possible to generate identifier references and indices that span multiple documents (using the `noindex` command), it is not possible to reference chunks from different files. A chunk's scope exists solely within the file in which it is defined. Therefore, multiple file support is very limited, and is purely enabled or disabled by the formatting language used.

Noweb cannot weave documentation that spans across several separate, linked documents. For example, HTML output is constrained to one page — cross-linking among pages is not possible. The scalability of this approach is not feasible for large programs. Tools such as \LaTeX2HTML , however, can be used to convert \LaTeX documentation to HTML, providing a method to circumvent the problem.

Noweb is available as a cross-platform (both *NIX and Microsoft Windows) distribution, but requires the installation of the Icon programming language.

Future improvements planned for **Noweb**⁴ are:

- the ability to cross-reference among documents,
- easier porting and installation, and
- improved performance.

2.3 *Nuweb*

Nuweb⁵, developed by Preston Briggs [9], receives its inspiration from a number of LP tools: **WEB**, **FunnelWeb**, and **Noweb**. Simplicity is its emphasis.

Whereas **Noweb**'s extensibility lies in its command line options and filters, **Nuweb** provides, as do most other tools, extensibility via in-source directives. **Nuweb** uses directives to perform such operations as source code output (tangling) to multiple files (utilising the '@o' directive). Although multiple file tangling is allowed, multiple file weaving is not; it is impossible to split woven output into separate, interlinked files. **Nuweb** is programming language independent and utilises L^AT_EX as its document formatter.

Source files are output by **Nuweb** only if the newer version differs from the existing version. This process can be overridden, but can save significant time spent on compiling source code that has not changed since the last compilation.

Nuweb allows the programmer to manually define identifiers. As with **Noweb**, a heuristic-based approach is then used to find occurrences of this identifier. However, **Nuweb** does not perform cross-referencing of identifiers, so identifier definitions can only be used for indexing purposes. Like **Noweb**, this heuristic is not infallible. For example, the variable **var** in the expression **++\$var** is not recognised and thus not included as part of the index. Code chunks are automatically cross-referenced.

The output of indices of file names, macros, and identifiers is optional and can be included anywhere in the document.

⁴<http://www.eecs.harvard.edu/~nr/noweb/plans3.html>

⁵ which is itself a very good written example of LP

Nuweb's relatively simplistic design takes the form of one monolithic C program and allows it to be ported easily to different operating systems. As such, it is not extensible beyond its own fixed functionality. For example, **Nuweb** does not offer pretty-printing, and there is no facility to attach a pretty-printing module (unlike **Noweb**'s extensible pipeline architecture, for example).

Nuweb allows multiple input files to a maximum nesting of ten. Included files are simply expanded within the parent file. This allows cross-referencing among files.

nutweb [35] is an improvement on **Nuweb** and enhances its output capabilities by enabling the output of \TeX , as well as \LaTeX . It also provides parameterised macros, abbreviated identifiers, and the referencing of macros. Additionally, it allows the layout of side-by-side code and documentation. **nutweb** was developed by John Hurst of Monash University, Australia. **xLP** was also developed by John Hurst and is an extension of **nutweb** that converts XML-oriented **Nuweb** documents and outputs documentation in HTML. **xLP** is discussed in Section 3.1 on page 45.

2.4 *Funnelweb*

FunnelWeb [90] introduces its own typesetting language. Using this language, the programmer is able to disassociate himself from a third-party document markup language and use the macros provided. With this neutral approach, the programmer is then able to select either HTML or \TeX as the output document set.

FunnelWeb alleviates, but does not solve the 3-syntax problem, by providing its own translatable set of typesetting commands. The 3-syntax problem requires the author to utilise (1) a programming language, (2) a documentation language, and (3) an LP language (discussed further in Section 4.1.2) and thus, arguably, places undue overhead on the LP author.

FunnelWeb's typesetting commands are translated to either HTML or \TeX as selected by the programmer. This set is limited, however. It provides only a subset of the typesetting commands offered by the other layout languages; for example, the inclusion of graphics and figures is not supported. It is

possible to combine **FunnelWeb**-specific formatter commands with those of other formatting language commands. The ability to bind to a particular typesetter has a negative effect on portability to other typesetters, and thus counters the attempt to keep **FunnelWeb** markup-neutral.

The **FunnelWeb** typesetting language contains a limited set of macros that allow:

- new page headers,
- sectioning commands, and
- emphasis and literal commands for in-text typesetting.

Unlike in **Noweb** and **Nuweb**, macros can occur anywhere throughout the literate program. That is, **FunnelWeb** allows the development of macros within documentation chunks by allowing the programmer to insert documentation-specific comments — a functionality in many other LP tools. The ability to write comments allows the author to make informal notes that are not presented to the reader of the tangled or the woven document.

Like **Nuweb**, **FunnelWeb** has nested file support to a maximum depth of ten, and source code may be output to multiple files.

FunnelWeb macros (which can accept a maximum of nine arguments) make use of the ‘@’ character to delimit macros. Because macros may occur in documentation, in typesetting commands, and in chunk-related macro definitions, a **FunnelWeb** program tends to present a rather littered look (especially to the untrained eye).

If the programming language coincidentally makes regular use of the macro delimiter symbol, **FunnelWeb** allows the programmer to define his own delimiter symbol to be used throughout the program. This helps reduce the clutter that occurs when the delimiter symbol is also a commonly used symbol in the program and documentation source. This approach can, however, detract from the comprehensibility of the literate program — a programmer used to reading a **FunnelWeb** program delimited by the standard @ symbol may find the # symbol, for example, difficult to read. Another downfall to this approach is that the code might not be reusable. If a program uses the

default @ symbol, and imports a file written for another literate program, which has been delimited using a # symbol, the programmer will be forced to convert one of the files (which can have further repercussions upon the reusability of the literate programs).

As default behaviour, macros (code chunks) can be defined and used once, and once only; this contrasts with WEB, for example, which allows the use of a chunk as many times as necessary. This default behaviour, which allows for accurate error messages, may be overridden by the author by explicitly stating that a particular macro may be used many times (@M), or not at all (@Z). Macros can also be overridden, so that a particular macro can be defined more than once, by adding up to five '@L' symbols onto the end of the macro. The macro with the fewest number of '@L' symbols is used in the tangling and weaving processes. Code chunks are additive.

Pretty-printing and identifier cross-referencing is unavailable in FunnelWeb. FunnelWeb's linear numbering of chunks in its literate documents can make cross-referenced chunks difficult to find. Furthermore, code chunk definitions are encased in parentheses, which can give the reader the impression that each chunk is correctly scoped and functionally independent.

FunnelWeb is well documented. A complete set of documentation⁶ sets exist for three target audiences — tutorial, developer, and reference.

2.5 CLiP

CLiP [87, 86] (Code from Literate Programming) is an attempt to free literate programming from specific document formatters, thereby enabling programmers to use commonplace document editors such as word processors⁷ and text editors to develop literate programs. The text editor used must be capable of saving files in a text-based format (most editors are).

CLiP, like FunnelWeb, allows a user-definable token set to delimit documentation from CLiP parsable code.

CLiP is essentially a tangling agent. No explicit weaving takes place; therefore, the literate program itself is also the final literate document. This

⁶ See www.ross.net/funnelweb/ for an internet-based version.

⁷ The benefits of writing program source code in a word processor are debatable.

is an interesting approach because it has the derived benefit of solving the 3-syntax problem. One might argue, however, that CLiP has a distinct disadvantage because it is impossible to add formatting to documentation chunks in any way. This is unfortunate because word-processors such as Microsoft Word offer strong WYSIWYG (what you see is what you get) support. In addition, source code cannot be pretty-printed.

Navigation within the document can be difficult due to CLiP's inability to support hyperlinks within the document. Programmers cannot utilise the hyperlinking abilities of word processors such as Microsoft Word because they require the markup to be saved with the final document, which would disrupt transferability between text editors. Emacs users may find its bookmarking system is useful to overcome this problem.

One interesting feature is the ability to suppress comments when tangling CLiP. This can be beneficial when extracting data files from literate code, for example.

An extension to CLiP, CLiPPrep⁸, provides macro support.

2.6 LP Integrated Development Environments

Several attempts at developing LP IDEs (integrated development environments) have been made. Some were developed as explorative aids to programming, others to effect visualisation techniques, and still others to facilitate LP.

Applications range from applying GUI extensions to existing command line systems. For example, Cockburn and Churcher [20] utilise **Noweb** as their tools' LP language to incorporate novel visualisation techniques such as fisheye views [28], degree of interest, and holophrastic [81] chunk display. **Jaba**⁹ [19] also supports direct-manipulation such that common LP instructions like chunk referencing are automated via the GUI. Other techniques, such as Østerbye [58] and Trygve Reenskaug and Anne Lise Skaar [74], utilise a programming environment, such as Smalltalk, to develop Smalltalk-based LP IDEs (a programming environment within a programming environment).

⁸<http://www.ddj.com/ftp/1997/1997.06/literate.zip/>

⁹ Jaba only allows literate programming at the chunk level

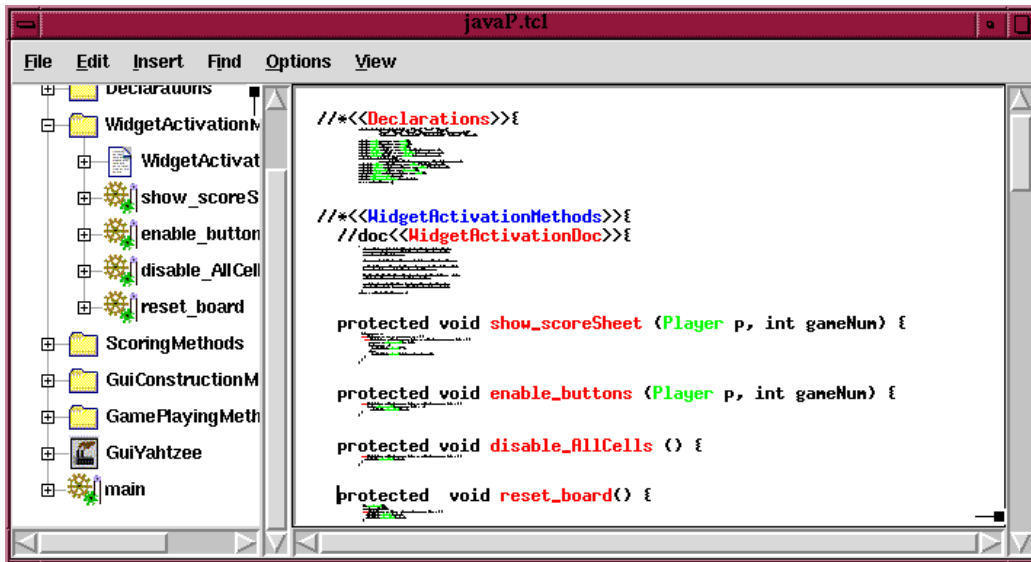


Figure 2.1: The Java LP editing environment. The left window displays a tree view of the literate program (in woven scope). The right pane demonstrates chunk holophrasing.

Most tools support, to varying degrees, navigation through hyperlinked chunks. Thus, a chunk’s definition and implementation are cross-referenced. Brown and Childs [11] WEB Interactive Environment cross-referenced and therefore hyperlinked identifier definitions and uses. The degree of navigation allowed is commonly dependant on the IDE’s support for the programming language.

Many tools also present a separate hierarchical view of the literate program, often as a tree-based representation of the tangle order of the document. The text-based view displays the literate program, and therefore, the psychological ordering of chunks.

Other attempts at IDE development are truly exploratory, such as that of Gurari and Wu [32], who utilised pre-determined fonts to determine a chunk’s type. This technique could be used to develop a comment-style chunk by simply marking up the chunk with a non-conforming font type, such that it is not processed in either the tangle or weave operations.

Extensions to common editing environments, such as Emacs, exist; en-

vironments for **Noweb**, **Nuweb**, and the **WEB** family facilitate literate programming. The `noweb-outline.el` package generates outlined, hierarchical views of the tangle order of a literate program, for example.

LyX [2], the L^AT_EX word processor (since release 1.04) supports **Noweb**-based literate programming¹⁰.

A significant and relatively successful contribution to LP IDEs is **Leo**, which remains under active development (version 3.2 released in August 2002).

2.7 Leo

Leo [72] is a GUI-based LP tool. It supports **Noweb** and **CWEB** literate programs. Literate programs are presented as trees of nodes. These nodes form what is aptly called an outline of the literate program. Each node in an outline displays either a chunk's name, or the name attributed to the outline node. The hierarchical view offered by an outline helps to establish an overview of a literate program. Each outline node is edited and viewed in the separate text editor as illustrated in Figure 2.2 on the next page.

Aside from developing new nodes, outlines may also be composed by copying and cloning nodes. Copying an outline node permits the author to edit the copied node without affecting the original node's content. Conversely, a cloned outline node's edits affect all other clones.

The concept of cloning is unique to **Leo**, and allows multiple views of a literate program to be composed. Given that a clone's edits affect all other clones, literate programs may suffer the penalties of decontextualised edits (see Section 6.3.4).

Leo decouples the hierarchical representation of a literate program from the nested hierarchy of chunks. Thus, a chunk `<<a>>` may reference chunk `<>`, but may contain outline node `c` as its immediate child in the outline.

Nodes may be moved about (up, down, left, right) the outline tree as necessary. Nodes are viewed and edited as separate entities of a literate program.

¹⁰ <http://www.sad.it/~jug/lyx/lyxdoc/Extended/> documents LyX LP support

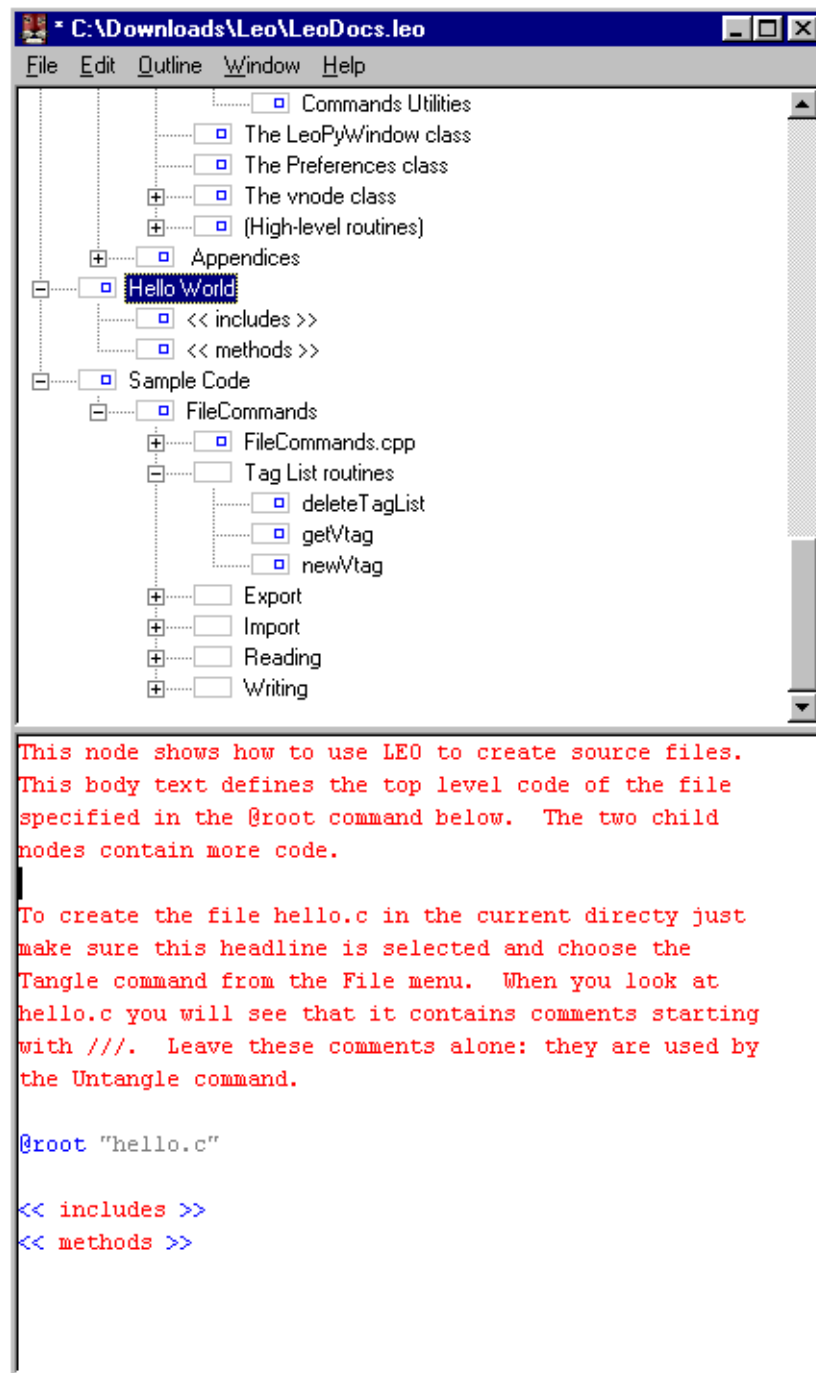


Figure 2.2: The Leo LP editing environment. The upper window displays the outline view. The lower pane displays the content of an outline node.

Leo’s interface, in which nodes may only be edited individually, eliminates temporal, “chatty” writing, and helps maintain the mapping between documentation and code chunks — both chunk types are forced to represent the same unit of abstraction. The lack of an overall low-level view of the literate program, however, makes the scope of program, and indeed, documentation artefacts difficult to ascertain (unless of course the entire literate document is published). For example, variable `i` in a code chunk is presented in an outline node. It is not immediately obvious what the scope of this variable is.

Leo insists that code chunks be defined within the scope of the chunk that they are referenced in. This means that a chunk’s implementation may not appear anywhere throughout the literate document/outline, which is a requirement of literate programming (see Section 1.2.3). Although a chunk may be cloned and included elsewhere in the literate program, the method of development puts an emphasis on the traditional development of the tangled structure and not literate program development.

2.8 Spider

Spider [69, 65] was written by Norman Ramsey (also the author of Noweb). Given a language description, Spider creates instances of a tangle program and a weave program. These program instances are then able to specifically tangle and weave any literate documents composed with any programming language.

Spider is an `awk` program that uses Silvio Levy’s WEB for C, (CWEB), with the C interpreter removed and replaced by Spider. This approach to LP, however, did not allow the creation of multi-language literate programs due to restrictions inherited from the WEB family. Although potentially any language is supported using Spider, *arbitrary* language support is not possible; the language must be predefined.

Production on Spider has been frozen and Ramsey has since developed the more successful Noweb.

2.9 *Documentation Tools*

In-source documentation tools allow the embedding and extracting of documentation or comments within the source code. The tools presented in this section are not considered literate because they do not satisfy the requirements laid out in Chapter 1, Section 1.2.1. Namely, they either do not have granularity of expression, or they do not possess the ability to psychologically order code for the reader's benefit (multiple orders of exposition).

The tools covered in this section generate static documentation. The document development process must be rerun in order to maintain current documentation. Literate programs undergo tangling and weaving upon compilation, thus ensuring the continual update of documentation and source code.

This section is not a taxonomy of existing documentation tools, however we have selected three tools of particular interest. We highlight their features and functionality in order to contrast them with the literate programming tools covered in Sections 2.1 — 2.5.

2.9.1 *Javadoc*

Javadoc was influenced by Knuth's work on **WEB** [27]. As the name suggests, it is a Java documentation utility. It uses comments embedded in the program source code and knowledge of the Java programming language to generate HTML pages that describe the class structure of a Java program.

The output of **Javadoc** is a set of highly interlinked web pages, whereby the reader may dynamically explore the inheritance structure of a class, its methods — overridden and overloaded, and a method's parameters and return values.

Javadoc provides an effective mapping to Unified Modelling Language (UML) class diagrams. Using HTML hyperlinks, it is possible to traverse the class structure of a given program or set of classes. The consistent layout and formatting of a **Javadoc** document adds to this mapping.

Javadoc generates its documentation by looking for predefined tags embedded in comments. Documentation tags include:

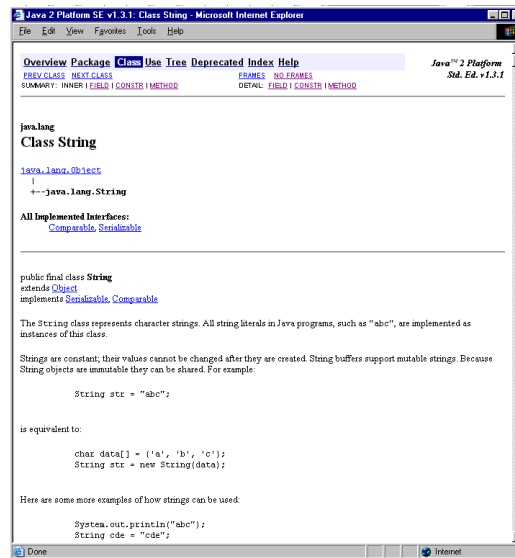


Figure 2.3: Javadoc formatted API documentation of the `java.lang.String` (Java 1.3) class.

- **@see:** adds a hyperlinked “See Also” entry to the class
- **@param:** parameter definitions
- **@exception:** descriptions for exceptions
- **@return:** methods return value
- **@author:** the author of the document
- **@version:** the document version

Javadoc, for all its power of expression, is a lightweight documentation system. It is simple to use and encourages a structured approach towards documentation. Combined with a styleguide [53], documentation consistency can be readily achieved. The styleguide also suggests a specific order of tag layout.

Javadoc is aimed at providing API (application programming interface) specification documentation. Specific documentation techniques are encouraged¹¹.

One differentiating feature of **Javadoc** is the use of inheritance of documentation artefacts. A subclass inherits superclass documentation if subclass documentation does not exist. Documentation is also transferred to overloaded methods, or to classes that implement a method in an interface. That is, documentation used for a method `m()` is also used for the method `m()'`, where `m()'` overloads `m()`.

Documentation comments are only recognised when placed immediately before class, interface, constructor, method, or field declarations. The scope of documentation is therefore fixed; the ability to document method-level statements is unavailable. Although limiting, this approach does add to the uniformity of documentation, and avoids the problems of defining rules for method-level programming abstractions of documentation. It is difficult to generate consistent rules for method-level documentation, which in turn creates difficulties in the layout and presentation of the **Javadoc** HTML output.

limited scope

As noted by Cockburn [19],

Javadoc is a post-hoc documentation strategy that requires that the class has been developed into a syntactically correct (and presumably complete) class specification.

This contrasts with LP, which promotes a pre-emptive documentation approach. Moreover, literate documents may be generated with syntactically incorrect code chunks.

Java Doclet Technology

The Java Doclet documentation technology allows programs that “specify the content and format of the output of the **Javadoc** tool” [52] to be written in Java. The programmer is able to customise the output of **Javadoc**. Using Java code and an API to **Javadoc**, the programmer may create custom func-

¹¹<http://java.sun.com/j2se/javadoc/writingdoccomments/index.html>

tions to extract custom tags, manipulate them, and then output Java-based documentation.

The Doclet technology is intended as an extension to the **Javadoc** tool. As such, users should adhere to the platform of recommended practices in the **Javadoc** specification. Although it is possible to extend the Doclet class to generate within-method documentation, this approach does not form part of the **Javadoc** recommendation.

2.9.2 *Elucidative Programming*

Friendly [27] wrote, describing possible future directions of the **Javadoc** tool: “...programmers prefer a graphical user interface for literate programming tools. Perhaps **Javadoc** should be integrated into this browser/editor so that programmers can do side-by-side editing of source and comments...”. Elucidative programming incorporates such an idea.

Elucidative programming, developed by Kurt Nørmark [56], is a documenting technique that, unlike the other tools in this chapter, stores documentation separately from source code. Although the documentation is not embedded in the program source, through the use of customised development environments, documentation and source code are developed physically side-by-side. A *prevenient* approach is encouraged, whereby documentation is developed before program source code. Tool support for elucidative programming includes an elucidative Emacs mode and an elucidative graphical user interface system aptly named ‘the elucidator’.

Some of the important concepts of literate programming are encompassed by elucidative programming; particularly the need to “address explanations that maintain program understanding and clarify thought behind the program”, contrasted with the **Javadoc** tool, for example, which solely provides API documentation. The target audience and nature of documentation are somewhat different than Knuth intended when LP was brought unto the world. He envisaged a work of literature that read from cover to cover — a novel-like work. Elucidative documentation practice considers program developers the target audience who use the documentation primarily as reference material (as do most other tools; both literate and ‘illiterate’). Although

program documentation should flow and remain consistent, emphasis is not necessarily on providing an interesting and entertaining read.

Elucidative tools must be language-aware, because documentable program units are equivalent to an abstraction in the programming language. A programming language abstraction, for example, is a package, class, method, function, or procedure; therefore, providing a fixed **granularity of expression**. Hence, like **Javadoc**, the language awareness of the elucidative programming tool is able to create hyperlinks between these programming language constructs and the supporting documentation.

To support mutual navigation between the code and documentation documents, hypertext anchors and links are included in both files. Although program abstractions delimit each documentation unit, it is possible to refer to units internal to a program construct. For example, one can explicitly refer to a particular statement within a method by embedding “source markers” in the method.

It should be noted that elucidative programming does not meet the LP requirement of allowing **multiple orders of exposition**, however, abstract representation is facilitated.

2.9.3 *perlpod*

The acronym **Perlpod** [3] (Perl Plain Old Documentation) is descriptive of the functionality it offers and the documentation it generates. The intent of **Perlpod**, according to its designer, Larry Wall, is simplicity, *not* power.

Perlpod allows the inclusion of directives (or ‘command paragraphs’) that the perl compiler ignores when compiling the program. These directives allow the inclusion of documentation within the program source.

Using directives means that a documentation section is devoid of the idiosyncrasies of the documentation tool. As an example, it is possible for programmers to include verbatim source code in the documentation without special treatment. Moreover, the source code itself is unaffected by the use of the documentation tool. Special treatment in **Noweb**, for example involves escaping any occurrence of the ‘<<’ characters by prepending them with an @ symbol. **FunnelWeb** also suffers from such issues, and is more susceptible

than other literate tools. For example, consider the following Perl expression:

```
@array = qw(john bill mary sue);
```

FunnelWeb requires¹²:

```
@@array = qw(john bill mary sue);
```

The difference is subtle, yet requires adjustment of the code presented to the compiler. This makes primitive operations such as cut and paste, perhaps from a literate code chunk to a conventional source code file, less straightforward.

The output of **Perlpod** is simply a linear representation of the documentation chunks. Program source code is *not* extracted by **perldoc** (the documentation extraction utility), unless it is specifically included as part of a documentation chunk; thus, source code must be duplicated in order to form part of the documentation.

Perlpod is designed primarily as an API interface. It follows a less structured approach than **Javadoc**. **Perlpod** does not offer Perl syntax recognition abilities and therefore cannot output, as **Javadoc** does, a list of methods of a class, for example.

Perlpod does not distinguish between formatting and content. For example, one of **Perlpod**'s directive sets is the unordered list construct:

- over
- back
- item

over begins the list and **back** ends the list. **item** creates a new item in the list. The **over** directive takes a numeric argument that tells the formatter how many spaces from the left hand margin the list should be placed¹³. The

¹² if the default macro delimiter is used

¹³ the default value is four

distinction between formatting (the indent of the list items) and content (the items of the list) is blurred.

Customised formatting commands allow the programmer to **use bold face**, *emphasise*, **use fixed width typewriter font**, and so on.

Perlpod allows the generation of dedicated documentation towards a particular typesetting system (through use of the **begin** and **for** directives), in the form of formatter specific sections. Thus, the author may stipulate which document formatters may process a section of documentation. This is a convenient alternative to embedding markup within a perl file, which would not be interpreted correctly by an unsuspecting document formatter.

Supporting utilities in common perl distributions¹⁴ allow the transformation of Perlpod to HTML, manpage, L^AT_EX, or text format. Using a pipeline approach, one utility, **podselect**, prints selected sections of documentation to standard output, thus enabling an author to generate reader-specific documents. This is similar to Noweb's **elide** filter, but also allows the exclusion of the documentation.

Contrasted with Javadoc, Perlpod allows an author to have a direct influence on output presentation. Though this is possible using Doclets, Javadoc largely separates formatting from content.

Perlpod's simplicity is a testament to its success as an API documentation medium. Perhaps this is because Perlpod is the only promoted perl documentation tool. CPAN (Comprehensive Perl Archive Network — [1]) is an example of a well-maintained, well-documented repository of Perl source code. One noticable feature is the conformance to a documentation styleguide, such as [50].

2.10 Summary and Comparison

We have discussed the prominent LP tools available. Others exist, but our selection is based upon the degree of difference in the approach adopted by the respective tools.

¹⁴ e.g., standard install with Linux Redhat 7.2 and Activestates' Microsoft operating system installs (www.activestate.com)

Table 2.1 summarises a comparison among the LP tools covered in this chapter.

WEB is Knuth’s archetypical literate tool. Despite some of its facilitative features, such as pretty-printing of program code and automatic indexing and cross-referencing, it is a complex tool that has fixed dependencies on Pascal as the programming language and **T_EX** as its document formatter. Derivatives of **WEB**, including **Spider**, were developed to enable support for other languages; however, these tools also suffer from their dependencies on programming languages and the use of **T_EX** as their sole formatter.

The trade-off between fixed language and language-neutral tools is the ability to cross-reference and index program-level abstractions. While **WEB** and its immediate derivatives are programming language-aware, tools such as **Noweb**, **Nuweb**, **FunnelWeb**, and **CLiP** are not.

Noweb was Ramsey’s attempt at solving many of the issues facing literate programming with the use of **WEB**. **Noweb** provides support for pretty-printing (via filters), cross-referencing, and identifier indexing. **Noweb** has relatively few directives, however, programmers can make use of its pipeline architecture, through the use of filters, to extend its functionality¹⁵. This is an effective method of separating document processing from literate source development.

Nuweb is the simplest of the tools discussed.

FunnelWeb contains its own abstract typesetting macros. These macros are translated to the appropriate document formatting language. **FunnelWeb** and **CLiP** (through **CLiPPrep**) offer powerful additive macro capabilities, whereby macros that accept multiple arguments may be defined. **WEB** provides a limited, one-argument form of this¹⁶.

CLiP is different because the layout and formatting is determined by the document formatter chosen (e.g., Microsoft Word). **CLiP** relies upon the document formatter to perform functions such as cross-referencing and indexing of macros and identifiers.

Many of the LP tools have adopted the use of macros in the tradition of

¹⁵ it is possible to generate filters that accept user-specified directives

¹⁶ although this limitation may be overcome through the use of nested macros

<i>Feature/LP tool</i>	WEB	Noweb	Nuweb	FunnelWeb	CLiP
Abbreviated identifiers	yes	no	no	no	no
Additive chunks	yes	yes	yes	yes	no
Automatic identifier matching	yes	yes	no	no	no
Cross-referencing of chunks	yes	yes	yes	yes	no
Multi-file cross-referencing of chunks	no	no	yes	yes	no
Identifier matching feature	yes	yes	yes	no	no
Typesetting language independence	no	no	no	yes	yes
Output formats	T _E X	L ^A T _E X, HTML	L ^A T _E X	T _E X, HTML	any text editor
Multi-file input	no	no	yes	yes	no
Multi-file tangle output	no	yes	yes	yes	yes
Multi-file weave output	no	no	no	yes	no
Programming language independence	no	yes	yes	yes	yes
Parameterised chunks	yes	no	no	yes	no ^a
Chunk index	yes	yes	yes	no	no
Identifier index	yes	yes	yes	no	no

Table 2.1: Literate programming tools and their supported features.

^a available in CLiPPrep

WEB. WEB's macros were a system to bypass the shortcomings of the Pascal language.

LP development environments also exist. Notably, the **Leo** tool uses outlines to illustrate program structure. Using this method, a literate program is able to be presented in multiple views. The Glasgow Literate Programming Project¹⁷ (not examined in this chapter) also promoted the concept of multiple views (termed 'ribbons'), which was later discarded, however. **Jaba** is a Java-based LP environment. Important features it incorporates are the visualisation techniques of fisheye view, degree of interest, and holophrastic chunk displays.

Many documentation systems exist are not literate-enabled. **Javadoc**, based on the concept of literate programming, is a succesful documentation tool for the Java programming language, which generates API documentation. **Perlpod** also allows the embedded use of documentation within the code source, however, remains largely language unaware. Elucidative programming stores both documentation and code files separately and requires tool-based support to develop programs. Instead of producing a novel-like representation of a program, elucidative programming's attempt is to promote the explanation of programming abstractions to aid as a reference source.

In Chapter 3, we look at the XML-based literate programming tools. We then analyse the common deficiencies of LP tools, and the model these tools are based upon, in Chapter 4.

¹⁷ <http://www.desy.de/user/projects/LitProg/glasgow/top.html>

Chapter III

A Review of XML-Based Literate Programming Applications

In this chapter, we investigate LP tools that combine XML (Extensible Markup Language [22]) with literate programming. LP with XML is largely a prototypical domain, and relatively few tools that combine these technologies exist. We examine these prototypical tools as serious attempts to develop LP environments; we acknowledge, however, the probability that these implementations will be altered and enhanced.

Most XML tools fundamentally follow the same paradigm as the more traditional, non-XML LP tools presented in Chapter 2. The use of XML in LP simply transfers functionality between paradigms; that is, the functionality afforded by XML is not fully exploited by current XML LP tools (albeit some tools exploit its functionality more than others).

The LP tools in this chapter are presented in no particular order.

3.1 *xmltangle*

Jonathan Bartlett wrote *xmltangle*, version 0.6, in an attempt to capture an LP environment using XML, without being constrained to a DTD (document type declaration¹). His application consists solely of a tangle process — unsurprising, given its title. For this reason, documentation chunks are DTD independent — a DTD of choice may be used, however.

Code chunks are also DTD independent; however, they utilise XML processing instructions for chunk directives. For example, chunk names use the `<?lp-section-id?>`, `<?lp-section-id-end?>`) processing instruction.

¹ <http://www.w3.org/TR/REC-xml#dt-doctype>

Other processing directives exist such as chunk references, output files, code chunk implementation and formatting instructions. This method of code chunk processing avoids the need to mark up a code chunk and its content, and is unique to `xmiltangle`. `xmiltangle` uses a SAX parser to process XML documents (see Section 7.2.1 on page 179) and is developed using the Python programming language.

The use of processing instructions, instead of DTD compliance, means that the DTD(s) used for documentation chunk development do not need to be extended, or indeed used. The repercussions of this are that the tangling process of `xmiltangle`'s literate programs can only be elegantly performed by purpose built tools; this is what `xmiltangle` does. Technologies such as XSLT cannot elegantly transform processing instructions and their textual content, as they do elements. The tangling process especially requires a customised tool.

The use of processing instructions defeat the use of XML as a markup language. Processing instructions are used predominantly to pass information to applications in a way that escapes most XML rules. Moreover, DTD use (instead of processing instructions) for code chunks has several benefits:

- validity checks may be performed on document content.
- editor support such as that of Emacs² may be used to facilitate XML document development.
- syntactic consistency within documentation chunks, which are marked up in XML, is maintained.
- indexing and cross-referencing of identifiers can be implemented as a separate, extended, process; thus, code chunk source is marked up to reflect this processing.

The weave process relies upon the author's selection of DTD to markup documentation appropriately and utilise methods of transformation at his

² psgml-mode <http://sourceforge.net/projects/psgml> offers DTD-based XML editing functionality

discretion, such as XSLT. Again, processing and transformation is thwarted due to the difficulty in transforming the code chunks (processing instructions and their content).

The latest `xmltangle` (0.6 July, 2002) is a complete rewrite of the original (0.1) and has fixed many of the original's shortcomings. Indeed, it was the original version that `xml-lit`'s implementation provided remedies for.

3.2 `xml-lit`

`xml-lit` [76] was developed by Rafael Sevilla in August 2001. It uses Clark's `Expat` XML parser³ to tangle and weave XML-based literate programs. `xml-lit` is a monolithic executable file comprising both a tangling and weaving agent. It is compatible with version 0.1 of `xmltangle`.

The feature differentiating `xml-lit` from `xmltangle` is the use of DTD support for its code chunks and the use of a namespace for this DTD.

Following version `xmltangle`'s (0.1) DTD, `xml-lit` insists that each code chunk is accompanied by the name of the source document where it will also be tangled. Although this facilitates multi-file output, this tight coupling means that a code chunk is unable to be output to a file other than that initially stipulated. This largely defeats the purpose of allowing multi-file output. The example listing presented in Figure 3.1 on the next page is taken from the `xml-lit` distribution. It illustrates that the code chunk will be output to the file `gnomovision.c`. The referenced chunks *must* also be output to the same file.

A code chunk's identifier (its name) is unique, and thus, cannot be attributed to another chunk. Code chunks are not additive, therefore.

Free text within a `fragment` element is woven along with the referenced chunk's name and a reference number, as a cross-reference to the chunk's implementation. Thus, a chunk reference's display name can be different from its reference name. This is unique to `xml-lit`.

The weave option produces an XML document; however, further parsing of this document by an XSLT stylesheet, for example, is hindered due to the

³<http://www.jclark.com/xml/expat.html>

```

    <programlisting>
      <xml-lit:code xml-lit:filename="gnomovision.c">
int
main(void)
{
    <xml-lit:fragmap xml-lit:name="localvars">
      Local variables
    </xml-lit:fragmap>
    <xml-lit:fragmap xml-lit:name="maincode">
      Main code
    </xml-lit:fragmap>
}
  </xml-lit:code>
</programlisting>

```

Figure 3.1: A code chunk is defined within the **fragment** element. Code chunk references are made via a **fragmap** element.

non-marked up nature of the resulting code chunks and code chunk references. For example, the `literate` source excerpt of the previous example produces the following when woven:

```

<programlisting>
--Code fragment from file: gnomovision.c--
int
main(void)
{
  (localvars) [1]:
    Local variables

  (maincode) [2]:
    Main code

}
</programlisting>

```

Note that the content of the `programlisting` element is free text.

Sevilla uses the DocBook DTD as a documentation option for his literate programs. Any DTD may be used, however.

3.3 xLP

xLP [37], written by John Hurst of Monash University, is based on `nutweb`. It extends `Nuweb` (discussed in Section 2.3 on page 26). It requires the literate program to be written in `nu(t)web` source. Thereafter, it can be translated into its XML equivalent.

The DTD for the `nutweb` XML source is hard-wired in the program source code, and is therefore difficult to alter.

xLP comes with AXE (AJH's⁴ XML Engine) [36] which is a custom-built conversion program to translate XML to either HTML or \TeX . An interface to the AXE program was also developed [82]. AXE bears similarities to XSLT; however, it is not an XML-based language, but is arguably simpler to use than XSLT.

3.4 LPML

LPML [89] (literate programming markup language) is an ambitious XML LP tool. Many of its proposed features are unimplemented; the tool's documentation stresses that it is in 'pre-alpha' status.

Despite its incomplete status, LPML offers some noteworthy features:

- the ability to generate a new HTML page by using a top level (`item`) documentation chunk. Of the literate tools in Chapter 2, only `FunnelWeb` (Section 2.4 on page 27) offered this functionality. This is an important feature, and one deemed necessary for literate programs that deal with large programs.
- the ability to have `variants`⁵ of code chunks. This feature allows a code chunk to receive multiple definitions and conditionally tangle a chunk

⁴ the author's initials

⁵ The variants feature is *unimplemented*.

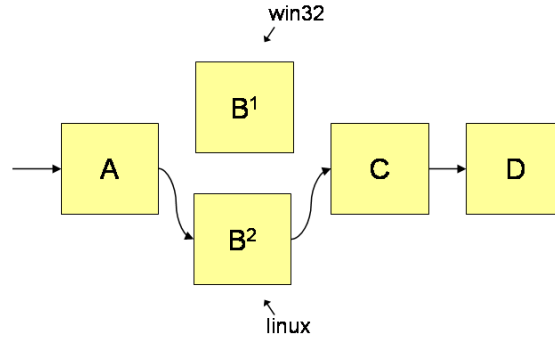


Figure 3.2: The variant tangling process.

variant based upon the variant adopted by the literate document. This technique bears great similarities to the functionality offered by WEB’s CChange files (see Section 2.1 on page 21).

The variants feature enables operating system-specific chunks to be written (e.g., Linux and Win32). Upon tangling the source code, the author may stipulate that the Linux variant is output. Figure 3.2 depicts this process. Chunks `<<A>>`, `<>`, `<<C>>`, and `<<D>>` all form a literate program. The Linux variant is stipulated by the author. Thus, the Linux variant of chunk `<>` (B^2) is tangled; the Win32 variant of chunk `<>` (B^1) is excluded.

- the nesting of code chunks within surrounding documentation.

A **piece** is a code chunk. The estimated DTD in Figure 3.4 shows that **piece** elements (code chunks) are children of **item** elements (documentation chunks). This rule deviates from other tools’ treatment of code and documentation chunks where the neighboring sibling relationship of code and documentation chunks is regarded as an atomic unit. Most LP tools associate a documentation chunk with a code chunk, physically locating them as a sibling pair. By treating code chunks as

children (or nested elements) of documentation chunks, LPML enables *surrounding* documentation — documentation both before and after a code chunk.

Figure 3.4 shows an edited excerpt as an example of surrounding documentation (written using LPML). Note how the `piece` code chunk is nested within the `item` documentation chunk. Two code chunks are nested within one documentation chunk. The body of the documentation chunk begins before the first code chunk, and continues following the end of the first and before the beginning of the second code chunk.

Code chunks must exist within a documentation chunk; they inherit their name from the documentation chunk that contains them. Code chunks are additive.

Code chunks are implemented by reference using the `insert` element. In the woven document, this appears as a reference from chunk reference to chunk implementation; the reverse-reference (cross-reference) from chunk implementation to its uses is not displayed.

xmLP HTML literate documents are the sole product of the weave process. Indexing and cross-referencing of identifiers (in code chunks) is not available (although mentioned as a potential enhancement).

The implementation of LPML, and some of the practices suggested in its supporting documentation, contravene a number of XML standards. Others foster a confusing development environment. The following list describes these violations.

badly structured XML documents: Badly formed documents⁶ may be tangled and woven. This inhibits LPML’s integration with XML technologies such as XSLT, which require a well-structured XML document.

LPML makes no use of the standard `Expat` libraries commonly used for parsing XML. A by-product of the acceptance of badly formed XML documents is that element content is able to include symbols such as

⁶ <http://www.w3.org/TR/REC-xml.#sec-well-formed> defines what constitutes a well-formed document

```

<item name="scan" label="Initial scan">
  The intial scan of the input file is pretty straightforward.  It
  simply reads everything and
  ...
  gather everything we need for tangling.  First, let's set up some
  globals we'll be using.

  <piece>
    @items = ();
    ...
    $formatname = '';
  </piece>

  The way I'm doing this is that I'm effectively using the name of
  the current item, and the
  ...
  nothing in the code which formally precludes that.
  <piece>
    while ([[INPUT>)
      {
        <insert name=".handle_tags"/>

        if ($piecename ne '') {
          ...
        }
      }
    </piece>
  </item>

```

Figure 3.3: An excerpt from the LPML implementation. A documentation chunk may contain a nested code chunk. The ommitted sections are indicated by the ellipses.

‘<’ (‘less than’ operator) and ‘>’ (‘greater than’ operator) rather than substituting them with ‘<’ and ‘>’ or using CDATA⁷ sections.

Another result of bypassing common XML libraries is the reliance upon parsing the XML document with regular expressions to extract elements, attributes, and content. Regular expressions are not recommended for XML parsing and processing: issues such as multi-line matching and greedy matching of symbols become problematic. As a consequence of the use of regular expressions, two XML elements cannot not occur on the same line.

peculiar attribute-naming requirements: LPML insists on the use of “x.subsection” (where x is the name of the parent item) as the **name** attribute of an **item** (documentation chunk) element. Conversion using standard tools such as XSLT may not be possible without unnecessary contortion (string manipulation). This protocol also means that an **item** chunk is coupled to its parent **item**. Reuse of documentation chunks is therefore impossible.

unconventional element ordering: An **item** is essentially a documentation chunk. Although **items** might not be nested, a hierarchy is established by the use of the **item**’s name attribute. An **item** is a child of another **item** if its name attribute contains the intended parent’s name (in the form **xxx.yyy**, where **xxx** is the name of the parent **item**, and **yyy** is the name of the child **item**). A parent **item** causes a new HTML page to be generated. A child **item** begins a new section in the HTML page.

Note that in Figure 3.4, the approximated DTD shows that the **item** element cannot be nested within itself. This means that sub-items are not nested inside the **item** they are logically a child of.

The **object** element indicates the file to which proceeding code chunks (chunks that occur after an **object** element) will be output. The code

⁷ character data — data that is not markup

chunks are not nested in the `object` element. It would be more intuitive, especially for later processing of the XML document (perhaps by XSLT), to include the code chunks to be output to a given file as children of the `object` element.

XML's extensibility is not exploited by LPML. The fixed nature of the elements used (they are hard-wired into the Perl program source code) means that the implied DTD cannot be altered easily. For example, the interpretation of an `item` (which can generate a new HTML page, or an HTML heading) is hard-wired into the LPML script. A programmer must alter the source code in order to alter its action, rather than enabling customised event handlers (in SAX tradition) or postprocessing on a resulting XML document before its translation to HTML.

The `format` element is a parameterless macro allowing the definition of a template file that an `item` element may adopt as a document layout framework. Certain predefined, hard-wired attributes are used to include content in the woven document. For example, `##body##` is replaced by the body of the woven document, allowing a template HTML page including this body to be developed. However, user-defined attributes are not possible.

3.5 litXML

litXML was developed by Vincent J. Carey ⁸. litXML was developed for use with statistical software. It is also an exercise in developing a LP tool that takes advantage of XML and its related technologies, such as XSLT.

This software has one unique feature that differentiates it from other *multiple presentation formats* tools; *multiple presentation formats*. Multiple literate documents, oriented *presentation formats* towards different audiences, can be generated from the same source file.

It is rare that only a single level of documentation suffices for statistical computing contributions. The code itself must be documented in detail, but that is rarely of interest to users. User level code in the form of a standalone document is often desirable. To this must be added

⁸ The package itself is downloadable from [16].

```

litprog
  format      :      name
  html        :
    head      :
      title   :
    body      :
      hr      :      width
      p       :
      a       :      href
      h2      :
      center  :
        hr    :      width
        nbsp  :
        table :      width
          tr   :
            td :
              font : size
            br    :
            a     :      href
  object        :      language, itemname
  item          :      format, display_class, labelname
  code          :
  p            :
  a            :      href
  i            :
  ul           :
    li         :
      code     :
      a        :      href
      i        :
      br       :
      b        :
      code     :
  piece        :      add-to
  insert       :      name

```

Figure 3.4: Estimated DTD of LPML: Extracted from `lpml_alpha.xml` with the `xmlStat` tool. Elements are presented in nested formation on the left and their respective attributes are presented on the right.

the various types of on-line documentation that can be called upon in the statcomp environment. — <http://www.biostat.harvard.edu/~carey/Aboutlit.html>

Although the capabilities to output this reader/user specific documentation are rudimentary, this is one of the few tools that incorporates this useful concept. Essentially, chunks are conditionally woven based upon a their `modname` value and their lexical ordering in the source file. Chunks are thereafter marked up depending on the target audience. Transformation is performed using an XSLT stylesheet.

litXML supports two types of code chunks. One may be referenced by other code chunks, while the other may not. It is utilised as a top-level code chunk to stipulate the output file of the tangled source. The use of two code chunk types is slightly confusing.

Pretty-printing, identifier cross-referencing and chunk cross-referencing are not supported; however, they may be implemented by extending the existing XSLT stylesheets.

Although no explicit DTD was provided with this tool, we generated Figure 3.5 using our `xmlStat` tool. Note that code chunks can be nested as children of any element. Note also that code chunks contain an attribute (`lang`) for the implementation language of the code chunk — enabling chunk-specific markup.

3.6 `xmlLP`

Anthony B. Coates developed `xmlLP` (version 1.0 [18]), a well-constructed LP tool — one of the more polished tools examined in this chapter, both in design, and implementation — in the mould of `FunnelWeb`. It mirrors the macro support available in `FunnelWeb` (Section 2.4 on page 27). Warning errors are generated by the accompanying `xmlPvalidate.xml` XSLT stylesheet, which is also faithful to `FunnelWeb`'s error messaging.

The weaving and tangling processes are completed by `xmlPweave.xml` and `xmlPtangle.xml`, respectively. DocBook or XHTML are the suggested markup languages for documentation chunks. The weave stylesheets are

```

article
  hr :
  section :
    code : modname, lang
    fragment : id
    fragmentRef : id
    fragmentRef : id
  title :
    font : color
    footnote :
    url :
  p :
  footnote :
  subsection :
    ol :
      li :
        code : modname, lang
        title :
        p :
        footnote :
        ul :
          li :
appendix :
  subsubsection : name
  title :
  p :
  br :
  ul :
  code : modname, lang
  title :
  p :
  img : src
title :
abstract :
  par :
  ol :
author :
  footnote :
date :
  month :
  day :
  year :

```

DTD-independent, and are used to generate XML documents, essentially copying documentation chunks unchanged and adding indexing attributes to code chunks to aid cross-referencing.

Although multi-file output is not included in the XSLT 1.0 standard⁹ (it is recommended in the XSLT 2.0 working draft¹⁰), the XALAN XSLT processor [25] allows additional processes to be created to enable multi-file output (through the `lp:file` tag). This method differs from `litXML`'s (Section 3.5), which requires separate XSLT documents to output multiple documents¹¹.

`xmlP` (like `FunnelWeb`) does not support identifier indexing and cross-referencing. Additive code chunks are allowed. The DTD works well and provides for intuitive use. Unique to `xmlP` is the use of the `name` element, instead of an attribute, to identify a macro element (`lp:macro`).

3.7 DBLP

After experimenting with literate programming through SGML and DSSSL [92], Mark Wroth then developed DBLP [93].

DBLP uses DocBook [55] as the basis for its DTD specification. DocBook is a set of DTD specifications used predominantly for the markup and preparation of technical documents. Compliance to DocBook's rich set of markup promotes the portability of technical documents and is thus well suited to literate programming. DBLP incorporates extensions¹² to allow for:

- formatting of cross-referencing macros (denoted by the `xref` element and the `xreflabel` attribute of the `programlisting` chunk).
- literal characters to be output (e.g., `greaterthan` is translated to '>').

The `programlisting` element from the DocBook specification was extended to allow multi-file output and additive macros.

⁹ <http://www.w3.org/TR/xslt>

¹⁰ <http://www.w3.org/TR/xslt20/>

¹¹ Carey, however, uses the multiple output feature for a different purpose; weaving different literate documents.

¹² Effort was expended in minimising changes to the DocBook DTD.

DBLP also utilises Norman Walsh’s DocBook DSSSL stylesheets¹³. With slight modifications, they also conform to LP requirements.

The system contains two DSSSL documents that perform the tangle and weave operations. The weaving process supports HTML, but can also utilise DocBook transformation stylesheets to produce other formats.

3.8 Summary

Several XML LP tools have been investigated. Many of these tools are still in their prototypical stages of development. This reflects the relatively recent emergence of XML and its related tools. Nevertheless, each tool has been evaluated for its viability as an LP candidate.

Several of these tools do not deviate greatly from their more conventional (non-XML) counterparts, covered in Chapter 2. `xmlLP`, one of the more complete implementations of the XML LP tools we studied, greatly resembles `FunnelWeb`, and `xLP` is an extension on `Nuweb`.

Importantly, however, are the `LPML` and `litXML` tools’ proposed concepts of chunk variants and multi-level documentation, respectively. Variants allow conditional tangling, while multi-level documentation allows conditional weaving. We extend these concepts in our proposed model in Chapter 5.

XML offers extensibility to define, and then redefine, a document’s data structure. It also facilitates ease of transformation, due to its well-formed syntax and the use of tools such as XSL — hence it is becoming the preferred choice of LP representation. Some tools, however, forgo this extensibility. `LPML` hard-wires the XML DTD into the LP tool itself. This has repercussions on the ability of the programmer to alter the DTD, and therefore also on the functionality offered by the LP tool. It also affects, in the case of `LPML`, the ability to output the literate document in different formats. `xLP` also uses a fixed DTD, which makes alteration of the markup used for documentation chunks difficult. Stylesheets can be utilised to alleviate this problem. Fixed DTDs are inevitable to some degree, especially to represent the literate programming model, and this is necessary to provide instructions on code chunk definition and referencing. Tools such as `xmLTangle` and `xml-lit`

¹³<http://www.docbook.org/>

often implement such a code chunk model. Through use of XML-based technologies (such as XSLT) for document transformation, as well as supporting DTDs, LP development becomes more extensible. `xmlLP` is a good example of this adaption.

DocBook is a common choice for documentation markup. DBLP illustrates that with slight adjustments, DocBook may also be used to facilitate tangling. Most literate tools use HTML as the woven format. HTML is a convenient markup language and HTML browsers are commonplace and functional, warranting their use as document viewers. The DocBook DTD can be readily processed into several other formats, using pre-existing XSLT (or DSSSL documents in the case of DBLP) stylesheets (e.g., pdf, rtf, \LaTeX).

None of the tools covered are language-specific. They all aim for language independence. Language-specific tools, such as `WEB`, allow identifier indexing and cross-referencing, as do generic tools such as `Noweb` and `Nuweb`, through heuristic means. This important and useful feature of LP tools can greatly facilitate the understanding and navigation of literate documents. Unfortunately, none of the XML LP tools contain these useful features. Only LPML makes mention of the *intent* to support such functionality.

Also missing from the XML LP implementations are paramaterised macros. In the case of `xmlLP`, this is a design decision; Coates has opted to exclude paramaterised macros from `xmlLP`, but suggests that they may be included in future versions. LPML approaches this functionality to a small degree through use of its `format` element, which is used for the sole purpose of creating documentation templates.

Chapter IV

Literate Programming's Limitations

In this chapter, we draw from our examinations of literate programming applications in Chapters 2 and 3, to discuss the limitations of literate programming. The examination of these tools highlighted that LP's shortcomings predominantly extend from two areas:

1. the LP model and
2. a tool's implementation of the LP model.

It is important to recognise that an LP tool's limitations are not always reflective of the LP model: the shortcoming may be implementation specific. It is, however, the model-centred shortcomings that pose the greatest barrier to LP's acceptance in the software development community. We firstly address, in Section 4.1, common implementation-specific shortcomings of literate tools. In Section 4.2, we examine the shortcomings of the LP model. Finally, in Section 4.3, we present the philosophical issue of documentation support and how the LP model can affect it.

4.1 Application Specific Shortcomings

4.1.1 Debugging

Literate programming's most criticised shortcoming has long been the disparity between reported line error and the occurrence of the line in the literate program. For example, a reported syntax error on line 32, for example, could very well be found in the fifth line of the chunk `<<generate sorted list of client names>>` which is located on line 2334 in the literate program. This problem is caused by two factors:

1. the psychological ordering of the code chunks differs from the computer-oriented order, and
2. added documentation between code chunks.

To locate the offending line of source code in a literate program, it is common for a programmer to find the offending line in the tangled source code file, gain an appreciation of the surrounding context of this line, and search the literate program for the line's occurrence — a clumsy process indeed.

Certain literate tools, such as **Noweb**, **Nuweb**, and **FunnelWeb**, allow line number indicators at chunk boundaries to be emitted. These can be used by the C preprocessor, for example, to indicate the occurrence of the offending statement in the literate program. The number of languages that accept line directives is limited however, and thus this method is not reliable for all programming languages.

The **Leo LP** tool, whilst promoting the understanding of the overall program — well-named headlines and the tree-like representation of chunks is effective — can exacerbate the debugging problem due to a node's headline not necessarily mapping to a chunk (documentation or source code).

4.1.2 The Three-Syntax Problem

Commonly, three languages (hence 'three' syntax [24]) must be learned and applied to develop a literate program. These are:

1. the programming language, e.g., Java,
2. the formatting language, e.g., \LaTeX , and
3. the literate program's language, e.g., **Noweb**.

The language of documentation, English in many cases, is excluded from this list, however, forms a vital role in comprehension and communication. Notational design languages such as UML add more complexity.

These languages arguably place undue overhead on the programmer and require beginning programmers to become gain general semantic knowledge of each language. Conversely, using conventional programming methods only requires knowledge of the programming language.

FunnelWeb provides support that alleviates the three-syntax problem by introducing a customised set of documentation chunk formatting instructions. The richness of these instructions is limited, however. Elaborate formatting requires the specific use of the selected document formatter’s commands.

GUI tools provide environmental support to alleviate the burden of the specific syntax of each language. The **Jaba** (see Section 2.6 on page 30) tool enables direct manipulation [78] of chunks via its menu selections — this also promotes syntactical correctness, which allows the programmer to focus on chunk manipulation, rather than its representation. **Leo**, however, affords direct manipulation to manage tree node development, however, in doing so, introduces a fourth syntax. It requires the direct textual entry of the other three syntaxes.

Solving the 3-syntax problem does not rid authors of the necessity to conform to the each target processor’s requirements. Whereas the three-syntax problem can be alleviated by introducing the use of XML, it is still necessary to understand and conform to the semantic model of each target language or process:

1. the LP tool’s processing requirements (chunk referencing must be resolved by adhering to the chunk model),
2. the document formatter’s processing requirements (the XML document must be valid and verifiable), and
3. the programming language’s processing requirements (the source code must compile).

4.1.3 Monolithic Files

Output

The weave process of most LP tools generates one monolithic document. While this method is satisfactory for small programs, larger program readability may suffer. Monolithic documents fail to convey the semantic grouping of sections (such as chapters and their contained sections) that printed media are able to. They also fail to exploit the cross-referencing, and hence, navigation, that could occur between multiple documents. Furthermore, HTML browsers don't perform well with large documents.

Exceptions to the norm are **FunnelWeb**, its XML counterpart, **xmlLP**, and **LPML**, which provide the ability to generate multi-file woven output.

Noweb uses **L^AT_EX2HTML** [23] to weave HTML documents from **L^AT_EX**-based literate programs. **L^AT_EX2HTML** supports the segmentation of large **L^AT_EX** documents into smaller files, thus solving the monolithic file problem. The conversion operation from **L^AT_EX** to HTML is largely fixed, however. It does not allow the author to adapt the translation of **L^AT_EX** formatting rules.

In addition to monolithic output, literate tools do not provide an elegant facility to either include or exclude documentation or code chunks from the weave process; the **litXML** tool, which facilitates conditional processing, is an exception (examined in Section 3.5 on page 54).

Most tools facilitate multiple file output during the tangling process. Although this was not possible with Knuth's architypical **WEB** system, later adaptations, such as **CWEB**, incorporated this functionality, which has become a de-facto standard. Tools such as **Noweb** support this by allowing user-specified root chunks. Thus, the author is able to commit the tangle operation multiple times — each time with different root chunk¹.

Input

Most LP tools limit the use of multiple input files, if this support exists at all. Although certain tools, such as **Noweb**, can be coerced into including external files by using the document formatter's (**L^AT_EX**) document inclusion

¹ Makefiles are commonly used to facilitate this

(`\include`) command, this practice does not marry well with the **Noweb** cross-referencing heuristic (see Section 2.2). Other tools, such as **Nuweb** and **FunnelWeb**, allow the input of multiple files, but restrict the number of nested files². **Leo** supports the use of multiple input files.

The constraint of single file input can counter the concept that some methodological implementations (Java as an object oriented programming language for example) physically impose upon file structure. Classes are often contained in separate files: merging multiple classes in the confines of one file can detract from the modular distinction that OO imposes upon classes. LP can therefore distort the modularity that other methodologies attempt to impose, and therefore impose its own perspective on software development³.

By virtue of their use of XML, LP tools such as **xmLtangle**, **xmlLP**, **xml-lit**, and **litXML** can exploit XML capabilities⁴ to allow multi-file input. Multi-file input is reliant upon a conforming XML parser to resolve the inclusions, thus **LPML** (see Section 3.4 on page 49 lacks this ability.

4.1.4 *Tangling Creates Tangled Code*

WEB was developed with the underlying assumption that program development would only occur using an LP tool. It therefore tangled program source code such that it was purposely unreadable, and therefore uneditable in its raw state. Such an assumption is not unreasonable. Firstly, LP is not a widespread development methodology. Insisting on its use therefore, by insisting on an environment within which a program is developed, might create an even greater decrease in its utilisation. Some programmers may prefer to program in a traditional manner, and rather than be forced to program literately, abandon LP altogether⁵.

² **FunnelWeb** and **Nuweb** allow a maximum of 10 nested files — a limitation that may hinder large software development projects

³ While this is not necessarily bad in all cases, we believe that the author should be free to decide which perspective he takes.

⁴ XML entity references or use of the **XInclude** (<http://www.w3.org/TR/xinclude/>) standard (although this is not fully not part of the XML 1.0 standard (<http://www.w3.org/TR/REC-xml>)).

⁵ and therefore remain illiterate

Secondly, insisting on LP’s utilisation assumes that LP is a superior programming methodology. This may not be the case for all individuals, however⁶.

Thus, we prefer that equal opportunity LP systems are developed whereby unintrusive support, such that both programming literately or use of traditional programming means is provided (as suggested by Cockburn and Churcher [19]). Facilitating equal opportunity functionality will alleviate the problems of scoping (discussed in Section 4.1.5). Command-line based LP tools are unable to offer this functionality, due to their combined chunk and processing model (discussed further in Section 4.2 on page 70. *Leo* provides equal opportunity development through the use of its ‘untangling’ capability; however, this is provided so that the author may alter the tangled file externally to the *Leo* IDE. Our new chunk model, presented in Chapter 5, facilitates “equal opportunity” development.

4.1.5 *Scoping*

A code chunk’s implementation encapsulates a programming abstraction’s semantic and logical scope. Difficulties can arise in determining this scope because chunks may be composed by reference, therefore chunk implementation can occur in a physically distributed manner.

The following documentation chunk and code chunk serve as an example (extracted from `helloworld.nw`, in Appendix C.2 on page 283):

```
@ The greeting is printed using the printf function again, this time
   with a \texttt{\%s} format to insert the name.
<<print greeting>>=
printf("Hello %s", buffer);
```

The intention of the chunk is obvious, especially after reading the leading documentation. The scope of the variable `buffer` is not obvious. Because the programmer lacks a complete view of the program’s scope it can be difficult to identify exactly where the variable is defined, how it has been altered before this chunk, and how other code chunks use this variable.

⁶ although these individuals’ colleagues may suffer when maintaining his software

Scoping problems also affect the detection of mismatched parentheses for example.

Identifier cross-referencing in the woven document reveals the location of identifier definitions and uses throughout the document (illustrated in Figure 1.6 on page 14). The tangled source enables the realisation of the scope of program artifacts because of its flow-based representation of the program source code. In its unwoven state however, this information is difficult to ascertain manually — it is in this state that literate programs are presented for development. Editors that are LP-enabled and language-aware only are able to reveal this information — only **Jaba** (see Section 2.6 on page 30) facilitates such functionality.

4.1.6 Object-Oriented Limitations

Existing LP tools are unable to elegantly represent the OO concepts of overloading and overriding because representing methods with chunks that possess the same name are treated as additive chunks (see Section 1.4.1 on page 15). Not only is treating two distinct chunks as additive an incorrect representation in the woven document, the tangled source of this literate program would place these chunks in the same physical location i.e., in the same class file. This may be suitable for overloaded methods of the same class, however, overridden methods will need to appear in distinct classes, and therefore, different physical regions (separate Java class files, for example).

Identifier locating heuristics of tools such as **Noweb** and **Nuweb** are not amenable to object oriented approaches where same-named identifiers may exist, yet represent different programming abstractions. (Appendix C.3 illustrates an example where an overloaded method is ambiguously referenced and is discussed further in Section 4.1.7.)

4.1.7 Primitive Cross-Referencing

Code chunk implementations and references are (commonly) cross-referenced in the woven literate document. This is imperative to facilitate navigation through the documentation. Many tools support cross-referencing between chunks.

Cross-referencing between identifiers, however, is not fully supported by all tools. Only language-specific tools that have an intimate knowledge of the programming language such as Knuth’s **WEB** (and its derivatives) and **Jaba**, are able to generate accurate cross-references to program artifacts.

Language-neutral tools, such as **Noweb** and **Nuweb** produce less accurate identifier references. Both tools utilise heuristic-based (and therefore sub-optimal) identifier matching algorithms. Appendices C.3 on page 285 and C.4 on page 287 illustrate the limitations of a heuristic-based algorithm (**Noweb**’s and **Nuweb**’s respectively). Specifically, Appendix C.3 illustrates the limitations of identifier locating heuristics in OO languages: **startEngine** is an overloaded method. Both **startEngine** methods are invoked by the **beginExcursion** method of the **Driver** class. The identifier algorithm is unable to differentiate between the two methods, however. To do so requires language specific knowledge.

Although source code may be altered to accommodate a literate tool’s heuristic shortcoming’s, this is an undesirable approach. Ramsey comments that such errors may not necessarily be ‘bad’ in light of the time and effort saved in generating a perfect indexer.

4.1.8 *Limited Output Formats*

Most LP tools adopt a fixed representation mechanism because they tie their document output to a specific formatting language — **L^AT_EX**, **T_EX**, or **HTML** in most cases. Although **L^AT_EX** and **T_EX** are powerful document preparation systems, transformation to other formats thereafter is difficult⁷. This is due largely to the formatting-based, rather than semantic-based markup that **L^AT_EX** uses.

Furthermore, should the author choose to mark his literate program up in **HTML** for example, transformation thereafter to a **L^AT_EX** format commonly ends in unattractive results.

Document output of the literate program to an arbitrary document formatter is thus limited commonly to either **HTML**, or **dvi** (device independent) file (which can be processed further to generate **ps** (postscript) and

⁷ **L^AT_EX2HTML** is a conversion utility that translates **L^AT_EX** to **HTML**

pdf (portable document format) files).

XML-based LP tools such as xLP, xmltangle, xml-lit, and xmLP (see Chapter 3) generate an XML document that may be transformed, given the appropriate translation stylesheet, into any document format⁸.

4.1.9 *Static Documentation*

WEB’s roots stem back to 1981. T_EX was used to generate literate documents — static, printed documents. Practicalities, and environments, have changed somewhat, since 1981. Software is large. The evolving nature of software can render static documents quickly obsolete, or even incorrect. Referencing a static document for software information can, therefore, be misleading and incorrect. To overcome this limitation, methods of dynamic representation must be developed to reflect the current status of a literate program. No such literate tools exist.

4.1.10 *Disparity between Document Editing and the Formatted Document*

Although Sewell [77] suggests that LP reduces visual complexity in the literate document, Nørmark [56] notes that this emphasis on the literate document, and the beautification of it, is uncomplemented by a comparatively ugly formatting environment. Ramsey and Marceau [71] note that the “difficultly reading the source and the marked difference between source and listing complicate editing”.

Some LP tools place great emphasis on the final document’s presentation. Whilst benefiting the reader of the literate document, there is often a proportional cost paid — the programming environment is then littered with the LP tool’s primitive constructs and document formatting language’s syntax.

XML LP tools exacerbate this problem. Although XML can be utilised to make the documentation chunks and the literate tool’s constructs (thus alleviating the three-syntax problem — Section 4.1.2 on page 62), XML is

⁸ forgoing the weave process of xml-lit makes translation to different document formats easier — not as one would expect.

not easily processed by humans because of its verbosity. The XML literate program is often dissimilar to the comparatively beautiful literate document.

Code Interference

The syntax of a literate tool can cause the distortion of source code when there is a conflict between LP symbols and programming language symbols.

For example, the following Perl code

```
print << EOC;
```

in Noweb, must appear as:

```
print @<< EOC;
```

in order to avoid the double angle brackets being interpreted as chunk delimiters.

Oddly enough, Simon Cozens' **webPERL**⁹, a perl-specific tangle program utilises the '@' symbol — the commonly used **array** symbol — as the chunk delimiter.

4.2 Model-Centred Shortcomings

Figure 1.2 on page 6, expressed the LP chunk model as defined by Knuth. This model has been widely followed by most literate programming applications since. It shall be referred to as the traditional literate programming

traditional LP model.

model The traditional model constrains the development of literate programming. Although the model supports simplicity, it is significantly restraining, we believe, such that it limits the application of LP to the code development phase of the software development life cycle (SDLC). The main problems associated with the fixed model are:

Processing model and chunk model are combined: psychological flow is limited to only one flow.

⁹ webperl1999

Asymmetric processing model: the model is unable to create higher-order documentation chunks, and

Fixed chunk model: the chunk model's supports only documentation and code typed chunks,

Chapter 5 on page 82 proposes a new chunk model that solves these limitations. We firstly consider why the existing LP model requires replacement.

4.2.1 One Psychological Flow — Limited Readership

LP is not reflective of the multi-dimensional process of software development, nor the multiple views that may be required of a software system.

In the domain of 'Separation of Concerns', Harrison, Ossher, and Tarr termed the inability to represent a problem or program in any manner other than that dictated by initial decomposition of the problem, as inherently enforced by the programming language, as the "tyranny of dominant decomposition"¹⁰ [59]. Essentially, Knuth partially overcame this with the introduction of literate programming; however, the LP approach is restricted to only one other representation.

*Separation of
Concerns*

The readership audience of an LP is potentially large, as is its development audience; it would be unreasonable to suggest that one single document is well suited to all audience's requirements. Some common queries of a software system are oriented towards its design, design decisions, requirements, maintenance, development process of code, use of patterns, API, specific use of variables, and testing of classes and methods. Should a literate program contain all the required information in this list, it is impossible that its chunks are ordered such that all audiences are satisfied. A literate program written to explain the program's development would not make a good API reference, for example.

The traditional model of LP allows the programmer to produce a psychological ordering of chunks. For example, Figure 4.1 shows a literate program

¹⁰ Recent advances from this domain of 'separation of concerns' have given rise to technologies such as AOP and Hyperslices (discussed in Section 5.1.1 on page 85).

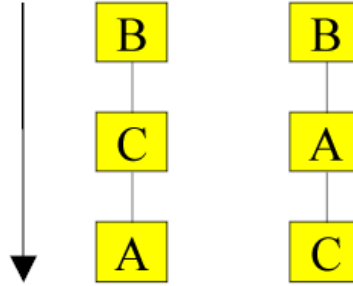


Figure 4.1: Chunks <<A>> <>, and <<C>> are presented in two psychological orders: <>, <<C>>, and <<A>> and <>, <<A>>, and <<C>>.

presented abstractly by three chunks <<A>>, <>, and <<C>>. The psychological ordering of these chunks is enforced by the lines connecting each square box. <>, <<C>>, <<A>> is determined as the best psychological order for human understanding. But what if you desire to present these chunks in the order of <>, <<A>>, <<C>>, for example? We are effectively asking for two documents to be created from one source web. This is not possible in the traditional model. Short of replicating the literate program and altering the sequence of chunks, the literate model does not support such operations.

Leo, through its outline approach, uniquely supports multiple views. All other tools do not.

4.2.2 *Asymmetric Processing Model*

Inability to Express Higher-Order Documentation

We argue that LP is largely limited to the software construction phase of the SDLC and is commonly utilised for ‘documenting-in-the-small’ [85] because of the fixed chunk and processing model employed by all existing LP tools.

The fixed model of LP makes it impossible to create relationships between documentation chunks. As illustrated in Figure 1.2, documentation chunks have an implied association with code chunks. This association is implied because documentation chunks are commonly used to explain a code chunk that immediately follows (in lexical order) in the source web. In contrast, code chunks may be nested, and thus form relationships with other code

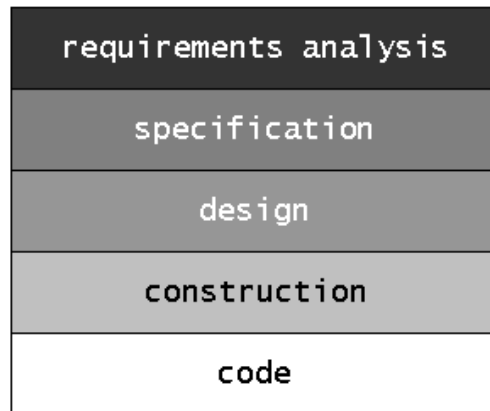


Figure 4.2: A layered representation of the software development life cycle.

chunks. The inability to develop nested documentation chunks causes, what we term, an “asymmetry of processing”. This is because the recursive processing of code chunks is different to the linear processing of documentation chunks.

*asymmetry of
processing*

The asymmetry of processing effectively constrains literate programming to the construction phase of the SDLC. The lack of nesting support for documentation chunks prevents the development of higher-level (or higher-order) documentation.

Figure 4.2 presents a diagram of the common phases in the SDLC. Source code appears as the first, or bottom, layer¹¹. Construction sits above source code in the second layer. Literate programming commonly represents the construction layer as documentation chunks. Thus, the construction and code layers form the literate program as documentation and code chunks, respectively.

While it is possible to represent any two layers of the SDLC with LP, there is a limit to expressing only two. Commonly (almost always in fact), it is the case that the construction documentation and source code layers are

¹¹ It is fully recognised that this may not necessarily be the case; although the phases are commonly presented in a circular fashion that implies continual updating, the layered model is more pertinent for our illustration, and does not detract from the cyclic nature of the SDLC.

represented. It is possible, for example, to represent the analysis documentation, driving the specifications analysis layer, for example. It is not possible, however, to elegantly include an entire third layer and create second-third layer nesting relationships.

Attempting to incorporate three or more layers into a literate program *convergence of layers* creates the phenomenon that we term as the *convergence of layers* — two or more layers are forced to be represented by one chunk type. This has negative side-effects such as:

Promoting undue emphasis on documentation: it is more difficult to incorporate two layered abstractions in the same chunk whilst maximising comprehensibility.

Psychological ordering loses its dominance: because the varying parts of a documentation chunk, which attempt to describe aspects of an abstraction, naturally fit into more than one psychological order, the emphasis, or direction of documentation can be lost.

Reverse engineering is complex: there is no clear manner to distinctly (and automatically) separate these layers once compounded.

There is also an assumption that there exists a one-to-one mapping between all layered abstractions — that one code chunk is represented by one documentation chunk, for example. This is often not the case; it is probable that one documentation chunk will influence the development of many code chunks (as depicted in Figure 4.3 on the facing page). Only the LPML tool (Section 3.4 on page 49) facilitates the relationship of one documentation chunk to multiple code chunks (by allowing surrounding documentation). No other tools support this.

Thus, as Cordes and Brown proposed the use of LP to contain design documentation [12]¹², we assert that it is not possible to elegantly do so unless (1) only design documentation will form the documentation chunks and (2)

¹² In a later publication [21] however, Cordes and Brown recognise that LP “is a technique to be used for system implementation.”

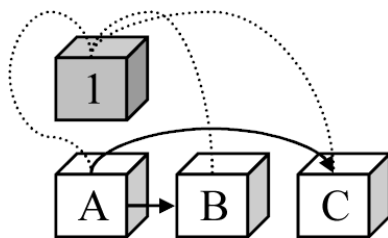


Figure 4.3: A one-to-many mapping commonly exists between high-level SDLC chunks and code chunks. Chunk <<1>> has a ‘describes’ relationship with chunks <<A>>, <>, and <<C>>.

a singular perspective is employed — multiple perspectives will document overlapping sets of chunks, and thus require multiple psychological orders.

To allow high-level documentation, documentation nesting must be facilitated by the literate model. This is an impossibility using existing LP tools’ fixed chunk model.

Poor Reuse

We argue that the difference between the weave scope and tangle scope is debilitating to LP. Given LP’s asymmetry of processing, it is impossible to perform the same operations in the weave process as one is able to in the tangle process. Whereas it is possible to reuse a code chunk in the tangled source by referencing it multiple times, it is not so in the woven document — a code chunk’s content cannot be displayed more than once.

Documentation chunks are also unable to be reused in the woven scope. Exceptions exist however. **Leo**, through its cloning mechanism facilitates documentation reuse. Noteworthy is the **Javadoc** utility that provides inherent documentation reuse by automatically promoting a documentation section to an inherited artefact (e.g., method, class) unless documentation has already been provided.

Code and documentation chunk reuse from multiple (external) webs is also not facilitated with any elegance (**Leo** excluded) — although multiple documents may be included, it is not possible to include specific parts of a document only. Namespace support does not exist, either.

The disparity between weave and tangle is enforced by the impossibility to include, and therefore reuse, documentation chunks in the tangled document. Thus, if it is desired to include a documentation chunk in the program source code as a comment, the author would be required to copy and paste the documentation chunk into the code chunk and mark it up in appropriate comment delimiters¹³ (we illustrate how this may be performed with our document development framework, in Section 7.2.2 on page 181).

Fixed Hierarchical Chunk Model

The difference between the features of a code and documentation chunk is the chunk's type. The difference in functionality of each is a one-way association from a documentation to a code chunk. The chunk model is thus has a fixed hierarchy. This is illustrated in Figure 1.2.

The reverse relationship, as illustrated in Figure 4.4 on the next page, such as a code chunk that implements a unit test and generates test output data in the form of a documentation chunk, is a scenario likely to be useful in an XP environment. The code chunk can be viewed as hierarchically superior in this case. Such representation is not elegantly possible with the current LP chunk model.

4.2.3 Fixed Chunk Typing Mechanism — Real World Overloading

Multiple chunk types are required for realistic software development. Current programming requirements not only blur the distinction between code chunk and documentation chunk, but also begin to render the chunk as a type-overloaded entity.

Internet web pages, for example, are commonly developed through a combination of HTML markup, a client-side scripting language (e.g., Javascript), and a server-side scripting language (e.g., Perl). The code chunk is overloaded with different code types. It is impossible to (elegantly) distinguish between the three different code types. Literate tools commonly imply that two types

¹³ Utilising the pipeline architecture of tools such as **Noweb** to develop filters that present documentation chunks in the tangled source is one solution to this problem. This is not, however, an architecture that all LP tools employ.

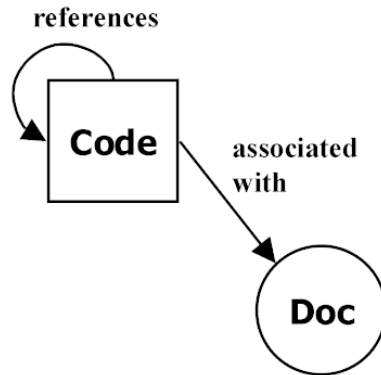


Figure 4.4: A code chunk has an association with a documentation chunk. The LP model cannot elegantly represent such a relationship.

of chunks exist only — as this example illustrates, this is not always true.

The documentation chunk is broadly unspecific. It is impossible to specify the type of documentation contained in the chunk. A documentation chunk could contain requirements analysis prose, a class diagram, or simply an elaboration of thought. The distinction between code and documentation chunks is blurred: is a UML notation segment documentation or code? Unit tests in XP are considered as documentation (because they provide a framework and direct the development of source code), however, are implemented as source code.

4.2.4 Refactoring — Chunk Version Control

LP development, despite its coercive nature to require intensive thought, is not immune to revision and enhancement. Literate tools do not facilitate chunk-level version control, such as a chunk versioning scheme that facilitates the conditional tangling and weaving of specified versions of chunks. Figure 4.5 on the following page illustrates version-based processing.

The existing LP processing model is, thus, anti refactoring: it is not amenable to the refactoring practices common in extreme programming [5] (XP), for example, whereby code is continually reviewed, refined, and simplified. LP tools do not support the conditional processing of documentation,

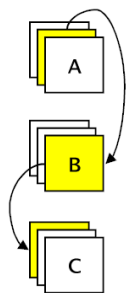


Figure 4.5: Three versions of chunks `<<A>>`, `<>`, and `<<C>>` exist. An elegant model would allow conditional processing of chunk versions, both in the tangling and weaving operations.

code, or atomic (code and documentation) chunks. Commonly, the order of chunk presentation in the woven document is dependent on the the lexical ordering of chunks in the source web; commonly *all* chunks are woven. The tangle process does not posses chunk versioning ability, either. Thus, it is impossible to tangle or weave one of multiple versions of a given chunk into the resulting document. The `WEB`¹⁴, `Noweb`¹⁵, `litXML`¹⁶, and `LPML`¹⁷ tools offer limited functionality, however do not perform all required operations.

Undo Factor

The atomic chunk — the combination of a documentation and code chunk to represent the same abstraction — is often reluctantly undone, or refined, to represent the abstraction (more) correctly — especially if the alteration requires that *both* documentation and code chunks are edited.

Chunk versioning functionality would appease this problem (by allowing chunk rollbacks), but not completely solve it, however. There is no apparent

¹⁴ `WEB` provides, through use of `CChange` files, the ability to replace code chunks only — there is an assumption however, that the LP is near production quality. Thus, a `CChange` file contains patches that may be integrated into the program source. Versioning, on the other hand, treats software development as a process whose quality increases incrementally. Versioning enables each stage of this process to be captured.

¹⁵ through use of its `elide` filter to exclude code chunks from a woven document, or the development of a filter to elide an atomic chunk (see Appendix D.10)

¹⁶ `litXML` facilitates conditional weaving

¹⁷ `LPML` proposes the use of code chunk variants.

method to support equal opportunity alterations such that an alteration in the documentation chunk would be reflected in the program source code (code chunk)¹⁸.

Altering one code chunk may render other code and documentation chunks invalid. Thus, the navigation to affected chunk implementations throughout the literate program can, relative to conventional programming methods, be a large overhead.

4.3 *Literature versus Documentation*

In addition to the tool-based and model-centred limitations of literate programming, we present this philosophical issue that focuses on the application of LP documentation practice.

We disagree with Knuth's position that a literate program should read like a book from cover to cover. Not all programmers can match the literary prowess of Knuth's expository and entertaining literate programs. Furthermore, many readers will refer to the program as a reference source, looking for explanation about a particular algorithm or variable or the interaction between two classes, not necessarily a complete understanding of the program proper. Others have also raised these concerns [80, 56, 72].

We recommend that reference-style documentation is written. The focus of documentation should be on software development rather than literary excellence. Temporal, novel-like documentation detracts from the issue of software quality. Cohesive units of code are better reflected by cohesive units of documentation, thus helping authors refrain from writing temporal, and therefore, highly coupled programs.

In Appendix A.1.1, we examine why documentation is necessary. Many documentation types can exist, and, in Appendix A.1.2 on page 237, we assert that particular types of documentation practice encourage alternative perspectives on software development, thus resulting in better quality software. Existing LP tools, apart from *Leo* (allbeit in an inelegant manner), do not support a varying approach to documentation of software systems and

¹⁸ Systems, such as *TogetherJ* do exist however that map edits to UML diagrams to Java source code, however, UML is structured and maps well to program source code.

thus encourage a monotone style of writing — one approach, and therefore perspective, is likely to be applied throughout the software’s development. They do not allow the elaboration and inter-connection between different sets of chunks required by such higher-order documentation (see Section 4.2.2 on page 72).

4.4 Summary

We have discussed several limitations of existing literate programming tools. We categorise these limitations into either implementation-specific, or model-centred limitations. We consider model-centred limitations the most serious of the two because they limit LP’s adoption and advance into common-place software development environments. The fixed model approach adopted by existing LP tools, not only is restrictive to some current methodologies, however, restricts LP’s future use also.

Of the implementation-specific limitations, we have discussed and demonstrated that the cross-referencing and indexing functionalities supported by literate tools are inadequate to cope with OO languages (for example). The concepts of overloading and overriding tax the identifier matching heuristic, and also prove that the coupling of the chunk naming mechanism with a chunk’s identification reduces LP’s amenability to OO.

LP’s use of chunk references to develop in a physically distributed manner produces scoping problems of code chunk identifiers; debugging is made difficult because compiler reported line numbers do not correlate with the LP source code and editing program artefacts that directly affect source code in a physically separate areas can cause consistency problems.

Input file restrictions that necessitate the development of source code in singular (or a limited set of) files can thwart the perspective imposed by methodologies, such as OO, which encourages modular development, by imposing a more global perspective. Furthermore, monolithic output of a literate document can misrepresent the modular nature of a literate program. Fixed output formats prevent the use of, and conversion to, other document formatters.

Other limitations are LP tools’ use of delimiting symbols that conflict

with programming language source, causing the escaping of relevant symbols in the source code. And finally is the disparity between beautiful literate documentation presentation and comparatively ugly literate source.

We also examined the limitations of the LP model employed by existing literate tools. Chief amongst these model limitations is the limited psychological order that literate programs may be illustrated in. This is due to the LP's combined processing and chunk models.

The asymmetric processing model that prevents the expression of higher-order documentation and also leads to poor chunk reuse, is caused by the code and documentation chunks possessing different functionality — the code chunk may be nested, the documentation chunk may not.

Furthermore, the fixed LP model's documentation and code chunks do not facilitate the representation of the diverse array of current notational, documentation, code, and multi-media types necessary for software development.

LP currently does not elegantly support a versioning system, however, this would be useful functionality to offer to methodologies such as XP, for use in refactoring, for example.

Finally, we discussed the philosophical issue of 'the manner of program documentation' and concluded with our recommendation that novel-like literate programming is to be discouraged because this approach detracts from the emphasis of good quality software development, and creates temporal, and therefore, coupled documentation chunks. The inability to express higher-order documentation is limiting to the perspectives that a programmer can approach software development, through documentation, from.

Importantly, as we illustrate in the next chapter, our generic model and document development framework facilitate multiple psychological orders, multiple chunk types, and higher order documentation. We also provide an elegant and workable chunk version management system.

Chapter V

Theme-Based Literate Programming

The tutorial is made up by a single scenario, which is elaborated into an increasingly complicated synchronisation scenario. We would therefore like to be able to extract several different programs from the tutorial, without having to state each in full. We have found no good method of doing this in **Noweb**, and have in the tutorial a full example program for each variant. This is not satisfactory, and we believe that a specialised literate tool is necessary to produce tutorials and other example based instruction material. — Kasper Østerbye [58]

In this chapter, we present the work that is the basis for this thesis — the development of a new chunk model to overcome many of literate programming’s shortcomings.

We discussed the shortcomings of LP in Chapter 4 and concluded that the chunk model supported by existing LP tools (examined in Chapters 2 and 3) is LP’s most limiting factor. To summarise the major shortcomings of the traditional LP model:

- only one psychological order may be woven and
- there is no support for the expression and implementation of higher-order documentation.

We stress that the model-centred shortcomings remain LP’s most serious drawbacks. All other problems are implementation-specific and can be solved via altered implementation or tool extensions.

Our new models comprise the concept of theme-based literate programming (TBLP). TBLP, through the enhanced literate model, solves many of

LP’s shortcomings. TBLP has the advantage of facilitating *multiple view* presentation and editing, and also facilitating high-level documentation. Our development framework allows the inclusion of processes at any stage throughout the theme-development process, thus allowing extended tool support. We extend our contribution to TBLP by physically demonstrating our chunk model and processing model in Chapters 6 and 7 via a tool-based archetypical implementation.

This chapter is organised as follows: firstly, in Section 5.1, we provide motivation for TBLP. Section 5.1.1 provides an overview of relevant research in the area of separation of concerns and similar areas of multiple-view representation. We then discuss, in Section 5.2, the chunk model and processing model required to support TBLP. We show that separating the chunk model from the processing model (Section 5.2.1) allows multiple ordering of chunks. Attempting to allow the development of higher-order documentation and solve the problem of documentation chunk reuse, Section 5.2.2 considers nested documentation chunks as an option.

We arrive, in Section 5.2.2, at a generic chunk model, which allows the development of arbitrarily typed chunks, and also allows any chunk to be nested. Section 5.2.3 discusses the representation of this generic chunk model. Sections 5.2.4 – 5.2.4 discuss chunk storage and the affordance of chunk-based version control that the new chunk and processing model allow.

The theme model is discussed in Section 5.2.5. The effect of the generic chunk on the processing model, and the processing model’s relevance to theme composition are then presented in Section 5.2.6. Intriguing, is that the new processing model requires a blend of the traditional weave and tangle operations. Finally, the issues of TBLP theme document development and multiple distributed webs are considered in Sections 5.3 – 5.4.

In Appendix B we present an example of how stages of software development, as a holistic process, may be represented using themes.

We stress that the model produced is elegant, powerful, and importantly extensible. *Our solution is a generic chunk (one that is not specifically a documentation chunk or a code chunk, but one that may be attributed any type), which is able to reference any chunk and likewise be referenced by any*

chunk, and therefore one that offers equality of processing to all chunks. The generic chunk enables freedom of expression, yet specific exposition.

5.1 Theme Weaving

LP allows the presentation of a program in a psychological order (the woven order). The ability to present multiple woven orders is beyond the capabilities of most existing LP tools.

Imagine, firstly, that LP does not exist. Now imagine a tool that allows the presentation of source code in the order in which it is developed (much like existing LP tools). This may be in a top-down, bottom-up, or nucleus-centred approach. Another author, preferring to program in the traditional order dictated by the compiler, might forgo the psychological ordering abilities of LP, but would still like to develop method-level granular chunks — for the purpose of threading a commonly themed set of chunks (methods in this case) across multiple classes to highlight a significant aspect.

Imagine the ability to present this program to multiple audiences; to add documentation to a program and order both the documentation and code to the individual requirements of each audience so that its presentation enhances the comprehension of each audience. Imagine the added ability to create connections between the program’s architectural design and its source code, and exploit these connections to illustrate to a human how the system architecture conforms to a specific design pattern. Imagine informally annotating source code, architectural specifications, and requirements analyses, such that these annotations are non-existent in the customer-deliverable documents, but can be viewed by the system developers. Audio and video recordings of customer interviews could be included as part of the requirements analysis documents. Imagine documenting an informal thought process and illustrating its progression throughout the entire software system’s development. Imagine that the program’s source code forms part of a pedagogical book; you present the source as a down-loadable internet document, however, you also include part of this source in the book as an example listing.

Imagine that compiler messages are linked back to the respective code chunks and are stored, along with chunk versions, to provide historical track-

ing and reasoning for syntactical alterations made to the program source. Also, at all stages of the program’s development, unit tests are prepared and executed upon the existing source code. The test results are presented as a document that is linked to the problem source.

Now try weaving all of these documents using an existing LP tool, and one web only. It *may* be possible to achieve one of these tasks — but certainly not all. Although it is possible to write a new web and weave a document for *each* of these specific tasks, this is a brute-force, inelegant approach. The limitation of one psychological ordering in the traditional LP model is, indeed, a serious drawback.

Each of these different views/uses of the software system is considered a theme. All of these imagined themes can be delivered by theme-based literate programming.

A theme, in TBLP, is defined as ordering of information, or chunks, towards a specific audience. Two themes may exist which include exactly the same chunks, for example, however order them differently for each audience type in order to enhance comprehensibility. Another theme may contain only a subset of these chunks.

5.1.1 The Need for a Tool of Abstraction

Advances in programming languages and development methodologies have not always been well met by LP tools — OO (see Section 4.1.6) is an example of a software development methodology that existing LP tools largely fail to elegantly accommodate. As future advances in methodologies are likely to alter in the aspects and perspectives they offer the programmer, so must LP, as a development and representational tool, facilitate the expression of these multiple perspectives and methodologies.

One area of software engineering attracting much recent attention is ‘separation of concerns’ [59] (the concept of concerns — identification, encapsulation, and manipulation of those parts of software that are conceptually and pragmatically relevant — was introduced by Parnas [63]). Two emerging implementations that encapsulate this concept are Hyperslices[59] and Aspect Oriented Programming (AOP) [43]. *HyperJ* [57], an implementation of

Hyperslices, allows the development of software, written in Java, in multiple perspectives. **AspectJ**, AOP’s implementation, is similar, but composes software through program transformation. AOP affects and manipulates code, whereas Hyperslices composes programs through inclusion or preclusion of programming abstractions.

Although **HyperJ** allows the development of software emanating from any stage of the software development life-cycle (from a requirement specification, for example), assumes that there exists a definitive mapping from requirements specification to system architecture to source code development. Mens [51], however, asserts that there does not always exist a direct mapping of elements from the architectural to the code level. He highlights the need for media that allow the development of software from these immediately unmappable abstractions. TBLP provides an elegant framework with which to represent and develop an equality-based separation of concerns. TBLP, unlike the **AspectJ** and **HyperJ**, is not limited to programming level abstractions, and can be used to contain abstractions that do not map explicitly to programming level abstractions.

Essentially, **AspectJ** and **HyperJ** focus on expression of software systems towards the computer. This is a fixed approach and suffers from what we *inequality of* term as “the inequality of concerns” — the inability to express a ‘concern’ to *concerns* any given audience. We contrast this with “equality of concerns”, which allows the expression of a theme towards any audience — be that computer(s) or human(s). TBLP’s focus is, following Knuth’s premise for LP, to express the multi-dimensional aspects of software systems *towards the human*¹. Moreover, Batory asserts that refinements are all-affecting; source code is but one affected artifact; “documentation, formal properties, performance models, and so on, change”. TBLP provides a framework for the consideration of all of these artifacts.

Other attempts at providing multiple perspectives have also been made. Morgensen, Tylvad, and Vestdam recognised the limitation of the existing

¹ The side-effect of AOP and Hyperslices is enhanced human comprehension due to a flexible approach to the separation of concerns. In contrast, the side affect of TBLP is quality program source code due to the ability to express to human audiences multiple separation of concerns.

LP model, and through their **DocSewer** [33, 88] tool, facilitated multiple ‘threadings’ of documentation and code abstractions (hence Doc-Sewer, we assume).

Specifically, **DocSewer** allows the documentation and extraction of the following *fixed* abstractions:

- relationships in class diagrams
 - class
 - method
 - class relations; generalisations
- source code

Thus, there is a finite granularity of programming abstraction that can be documented. Furthermore, software development in a psychological order is impossible. The author is forced to document in an annotated fashion after the programming artifact has been developed.

Although both TBLP and **DocSewer** attempt to solve the problem of limited psychological orders, both approaches differ and give different results. **DocSewer** is language and methodology specific. TBLP, in contrast, is language and methodology unspecific.

TogetherJ² is an IDE that facilitates the development of object oriented program source code. Development of source code is reflected by graphically presented UML notations such as class diagrams, sequence diagrams, and use case diagrams. The equal opportunity interface allows direct manipulation of the UML notations to be reflected in the respective program source artefacts. Although **TogetherJ** presents a powerful and advanced programming environment, programmers are governed by the tyranny of dominant decomposition. They are constrained to development with UML notation (some may see this as an advantage). Furthermore, the representation offered is layer-based. Cross-sections of these layers to combine relevant artefacts of source code, class diagrams and sequence diagrams, for example, are unable

² www.togethersoft.com/

to be displayed. Also, the interaction between layers assumes a one-to-one mapping between programming abstractions. **TogetherJ** offers, through its API, an extensible development environment.

Wikis [7] are interactive web sites that allow any page to be edited by any visitor, via a customised markup language. Authors are free to edit or annotate sections of prose and create new pages. A Wiki page is commonly used to discuss a user-instigated theme. Themes evolve through user-contributions. Further themes may be derived, and therefore, branched off by developing new pages with referencing hyperlinks. Hyperlinks not only facilitate navigation, but also (1) encourage linking to related pages, and (2) avoid replication of prose on a page.

Although Wikis are not commonly utilised for software development, their allowance of multiple perspectives to emanate from a given subject, and their affordance for meta-notation i.e., allowing authors to document existing documentation, provide a partial solution to the model-centred shortcomings of existing LP tools — limited psychological ordering and no support for higher-order documentation, respectively. Although Wikis facilitate the development of ‘themes’, their solution is partial from an LP perspective; they do not allow the reordering of documentation segments, nor do they allow documentation segment reuse.

5.1.2 Theme or Psychological Order?

We define a theme to be a psychological ordering of chunks. A theme, however, also implies that there are many possible orderings of chunks that might be composed from one or more webs (see Section 5.5 on page 113). A theme is advantageous because it can be aimed specifically towards a given audience. As the audience changes in ability and interest, the theme can be altered in an attempt to satisfy the audiences’ ability to comprehend.

The traditional psychological ordering of chunks assumes a fixed audience (there is a limit of one psychological order) and that the audience is interested in all parts of the program (all chunks are presented in the woven document). Themes, on the other hand, allow a more natural abstraction because they do not require that all chunks are woven (or tangled); a subset only can be

presented, and in any order.

5.2 Multiple Themes: *Theme-Paths and Chunk-Nesting*

A traditional web is composed of ordered associations of documentation and code chunks. The presentation order of these chunks in the woven document is dictated by their sequential ordering in the web.

We can view these implied sequential interconnections between chunks as an acyclic, directed graph of nodes and edges. Figure 5.1 on the following page shows four themes imposed on the one web. Each theme is a path that moves from chunk to chunk, thereby generating a graph of nodes (chunks) and edges. Two types of chunks exist — documentation chunks (square shaped nodes) and code chunks (circular shaped nodes). The traditional woven order is illustrated by the non-filled arrow.

Allowing the composition of multiple themes from a singular web means that it must be possible to define a path that threads together a specific set of chunks to form a psychologically correctly ordered theme. For example, an alternative theme composition is demonstrated by the dotted lines in Figure 5.1 that thread documentation and code chunks in a non-traditional (non-lexically ordered) manner. The grey line illustrates code chunk versions of “C1” that are ordered in a theme document. Theme composition can therefore be considered an ordered collection of chunks.

Chunk composition differs to theme composition, however. Chunks are nested structures, not sequentially linked entities. This is depicted by the filled arrowed lines in Figure 5.1, which represent the tangle order.

Chunk ordering is therefore distinct from chunk composition. Chunk ordering refers to the ability to determine a chunk’s position in relation to other chunks. This is the process of theme composition. Chunk composition, on the other hand, refers to the ability to build the content of a chunk with (1) free text and/or (2) references to other code chunks.

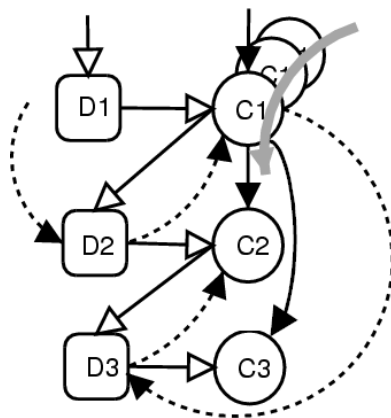


Figure 5.1: Four orders/themes, indicated by the four different line types, are generated from the one web.

5.2.1 Processing Model: Separation of Content and Ordering

Having determined that we require multiple themes to extend from a single web, we now propose a processing model that facilitates the elegant development and composition of multiple themes.

Figure 5.2 on the next page illustrates the traditional processing model used by current literate tools. This model makes it impossible to alter the ordered placement of chunks to generate multiple woven documents. This is because chunk implementation and chunk ordering are combined; the processing model relies upon the lexical order of chunks in the source file to govern the chunk's placement in the literate program — hence, the limitation of one psychological order.

The processing order is shown more definitively in Figure 5.3 on the facing page (as a lower-level abstraction of Figure 5.2). It typifies the tangling and weaving capabilities of the traditional processing model. Documentation chunks are represented as numbered circles and code chunks are represented as alphabetized squares. From the web, it is possible to tangle the source by starting at the root chunk, which in this case is `<<A>>`, and following the directed arrows to the code chunks that are referenced. The resulting tangled source is shown as the nested chunks `<<A>>`, `<>`, and `<<C>>`. The woven document is simply a reflection of the ordering of the chunks as they

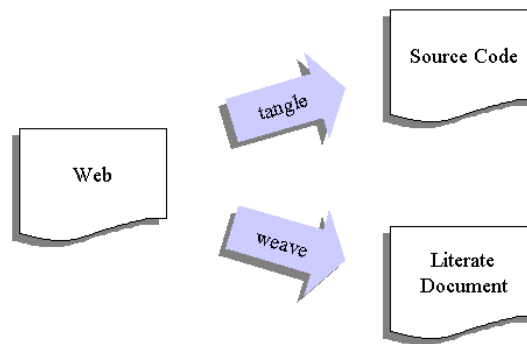


Figure 5.2: The traditional processing model.

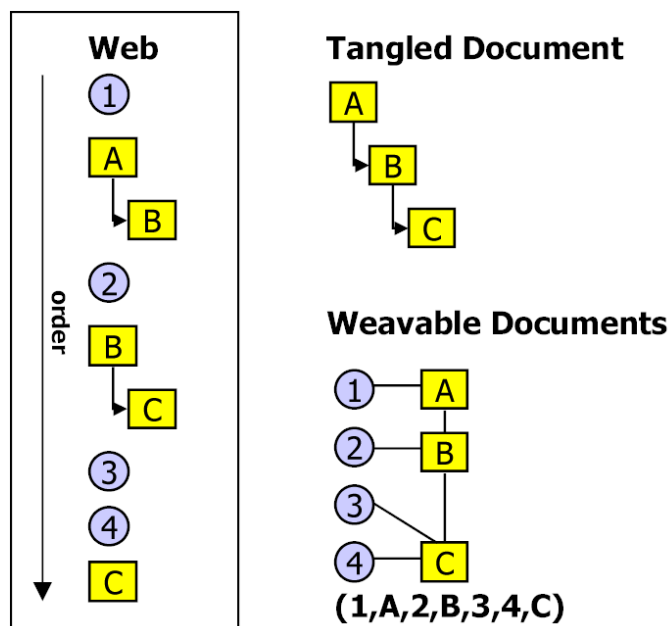


Figure 5.3: A web offers only two chunk orderings.

occur in the web. Note that chunks <<1>>, <<A>>, <<2>>, <>, <<3>>, <<4>>, and <<C>> appear in the same order in both the web and the woven document.

The ‘displacement’ model (described in detail in Appendix D Section D.3 on page 299) illustrates that it is possible to separate a chunk’s implementation from its order of inclusion in a woven document. Chunks may be presented in multiple orders and are not confined to appear in their lexical ordering in the source web; they may therefore be ‘displaced’. Specifically, to effect the displacement model, we implemented a filter for the **Noweb** LP tool. The **Leo** LP tool also allows multiple views of the same literate program by facilitating the development of multiple ‘outlines’. The **Glasgow Literate Programming Project** also initially provided what Will Partain termed as ‘ribbons’. The implementation was later discarded because “No-one uses it, and it’s pretty clunky in this system, both conceptually and implementationally.”³

The displacement model enables multiple psychological orders to be woven from one web, or repository. It is more pertinent to use the term *repository* to describe the storage of chunks as separate from their theme-ordering.

Figure 5.4 on the facing page illustrates that the separation of a chunk’s ordering (the “Chunk Ordering” files) from its implementation (the repository) enables authors to generate multiple woven documents. Note that the web in Figure 5.2 has now been replaced by (1) a repository of chunks and (2) a set of “Chunk Ordering” documents; each of which contains the order in which chunks are to appear in a theme. Thus, a weave operation now involves:

- the processing of a “Chunk Ordering” document to determine the order of chunk appearance,
- the extraction of the relevant chunks from the chunk repository, and
- the presentation of the theme document.

³<http://www.desy.de/user/projects/LitProg/glasgow/top.html>

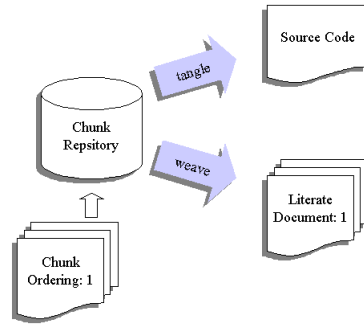


Figure 5.4: The displacement model separates between chunk implementation and ordering, thereby allowing multiple weave operations to be executed on the one literate program.

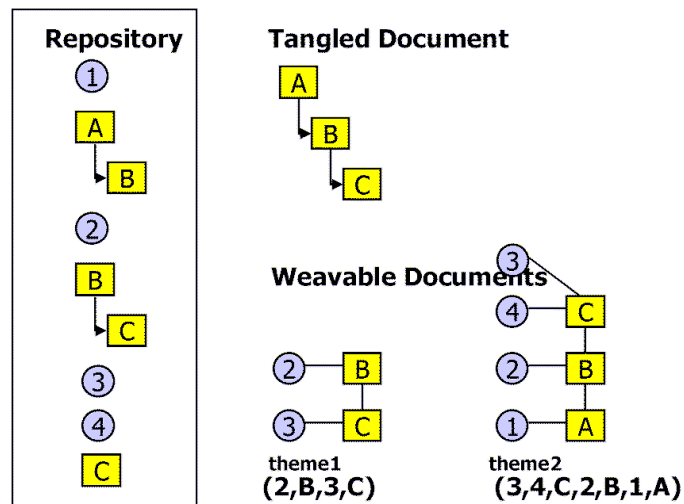


Figure 5.5: The displacement model enables multiple documents to be developed from a repository.

Figure 5.5 on the page before illustrates how the separation of content and ordering affects LP at the chunk level. The repository contains the same chunks as the web in Figure 5.3, however, it no longer maintains an implied ordering of chunks; any chunk’s implementation may appear anywhere in the repository. The lexical ordering of chunks in the repository is now a redundant feature. The tangle process remains the same.

It is now possible to generate any number of woven documents, containing chunks displayed in any order, which include or exclude any atomic chunk (combination of documentation and associated code chunk). Because the content each chunk is contained in the repository, the chunk ordering files need only contain a list of references. In the example in Figure 5.5, two documents have been woven:

1. Documentation chunks <<2>> and <<3>> are presented with their associated code chunks, <> and <<C>>, respectively.
2. The code chunks are presented in the order <<C>>, <>, and <<A>>, as opposed to <> and <<C>> in the last ordering. Note that chunk <<C>> now has both documentation chunks <<3>> and <<4>> associated with it.

Asymmetric Weaving

Two further implications arise from the employment of the displacement model:

- Not all chunks must be output from the repository.

Theme one presents only code chunks <> and <<C>>. Theme two presents chunks <<C>>, <>, and <<A>>. This contrasts positively with existing tools, which commonly output all chunks from a web to the woven document (a shortcoming explained in Sections 4.1.3 on page 64 and 4.2.4 on page 77).

- Theme-specific atomic chunks.

Although code chunk <<C>> appears in both themes, theme one associates only documentation chunk <<3>> with it, whereas theme two associates both documentation chunks <<3>> and <<4>>. This implies that a documentation chunk is marked up with the theme(s) in which it may be included. (This was the method employed by the ‘switch’ model, which we implemented and describe in Appendix D.2.1 on page 294). Thus, the documentation chunk is not *totally* devoid of knowledge of its inclusion in a woven themed document. This shortcoming is a result of the inability to reference a documentation chunk. Documentation chunk referencing is discussed in more detail in Section 5.2.2 on the next page.

If we were to generate these two woven documents using the traditional method, two separate webs (like that in Figure 5.3) would need to be developed. The separation of chunk ordering from the repository thus allows *chunk reuse*.

The Tyranny of Dominant Decomposition is Broken

The separation of chunks’ content and ordering allows LP to address separation of concerns (introduced in Section 4.2.1 on page 71) *on the programming level*; it is therefore amenable to technologies such as AOP and Hyperslices (Section 5.1.1 provides further explanation).

The ability to present a program in multiple orders allows an order of decomposition and expression to be undertaken that is not dictated by the programming language or compiler. Although traditional LP allows a psychological ordering of chunks, it offers only one such order of presentation. The new processing model, which separates content from ordering (allowing chunks in a repository to be referenced from “Chunk Ordering” documents), allows the development multiple psychologically ordered documents from one repository. Multiple concerns are now able to be represented; the following sections, however, show how the literate model may be further improved to allow the elegant and scalable implementation and representation of these concerns.

5.2.2 Enhancing Chunk Composition

Having enhanced the processing model’s ability to separate content from ordering allows the development of multiple themes from a single repository, we will now examine enhancements to the chunk model that enable higher-order documentation and more flexible chunk development.

The traditional code chunk model allows chunk composition via the referencing of code chunks. A tree-like structure is formed, and the traversal of this structure, in a depth-first manner, is called the tangle process. This tangling functionality is available specifically to the code chunk, but not to its documentation counterpart thereby disallowing documentation chunk reuse. (a shortcoming discussed in Section 49 on page 75). In Section 5.2.2 we solve this shortcoming by facilitating the composition of nested documentation chunks.

In order to elegantly facilitate higher-order documentation, in Section 5.2.2, we propose a *generic chunk model*; an enhancement to the (nested documentation) chunk model. A generic chunk allows chunk to be arbitrarily typed. We also consider the repercussions, both positive and negative, that this advance has on literate programming.

Nested Documentation Chunks

The code chunk provides a working nested model that we can transfer to the documentation chunk. Figure 5.6(a) on the facing page specifically shows that a code chunk may reference, and hence be composed of, many other code chunks. Existing LP tools are unable to compose documentation by reference. We bestow these features upon the documentation chunk, as shown in Figure 5.6(b) on the next page.

Combining the ability to nest documentation chunks with the ‘displacement’ processing model (introduced in Section 5.2.1), we are able to generate an immense array of documents from a repository ($\sum_{i=1}^n \binom{n}{i} i!$ combinations of chunks may be generated, assuming (badly) that no chunk appears more than once in the output document). For example, the two literate documents generated from the repository in Figure 5.7 on the facing page. The repository contains code chunks (rectangles) `<<A>>`, `<>`, and `<<C>>`, and

documentation chunks (circles) <<1>>, <<2>>, <<3>>, <<4>>, and <<5>>. Theme one contains documentation chunk <<1>>, which is associated with code chunk <<C>>. Note that the repository defines chunk <<1>> as composed of chunks <<2>> and <<3>>. Both chunks <<2>> and <<3>> are displayed in the literate document. Documentation chunk <<2>> appears again; however it is now associated with code chunk <>. This illustrates documentation chunk reuse — the lack of which is a shortcoming of current LP tools.

Theme two in Figure 5.7 is equivalent to tangled source code and contains the nested code chunks <<A>>, <>, and <<C>>. Note that theme **one** contains documentation chunks that are *tangled* to produce the traditional *woven* output.

In order to perform a tangle operation on a documentation chunk, each documentation chunk must contain a referencable unique identifier. We recommend a unique, automated, chunk identification (**chunkID**) naming scheme, which is distinct from the descriptive label (**name**) given to a code chunk. We find, in Section 7.4 on page 193, that the traditional chunk naming mechanism is no longer feasible for the generic chunk model we next propose.

The Generic Chunk

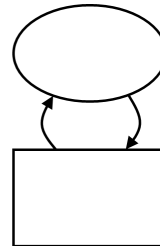
Having provided the documentation chunk with “nestability”, we now look to enhance the chunk model’s extensibility. We propose a generic chunk model and present a case for the generic chunk by defining the key restrictions of the current nested chunk model (as covered in Chapter 4). These are:

- a fixed hierarchical chunk model,
- an asymmetric processing model,
- an unrealistic set of chunk types, and
- a uni-directional association between chunk types.

These limitations of the traditional chunk model highlight three important requirements of an enhanced chunk model:

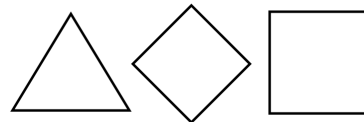
1. **multi-directional association between chunks,**

A specific chunk hierarchy should not be enforced by the model. The association, or nesting, of any chunks should be left to the author's discretion.



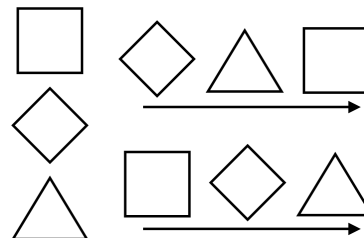
2. **limitless set of chunk types,**

A chunk's type should depend on its content and intended use. The author should be free to select an appropriate type for each chunk. We envisage that the chunk's name will indicate the role it is used for.



3. **a separate, non-discriminatory processing model.**

Chunks may appear in any document, be it tangled or woven. The processing model should process multiple chunk types, as dictated by the author.



These requirements are realised by a generalised chunk model⁴ — a *generic chunk* that is able to be arbitrarily typed and nested is the elegant solution that the rest of this dissertation is based upon, and represents an entirely new chunk model allowing programmers, as document authors, the freedom of arbitrary expression, yet specific exposition.

⁴and the people rejoice, the streets come alive, peace reigns, and software engineers look knowingly — they have just seen a great light

A high-level representation of the generic chunk is presented in Figure 5.8 on the facing page. A chunk may reference, and be referenced by, zero to many other chunks. Chapter 6 discusses the composition of the generic chunk in detail. We recommend, based on our findings, that a chunk contain the following (minimum set of) attributes:

type: The chunk’s type should indicate the nature of the chunk’s content.

name: The chunk’s name should be descriptive of its content.

chunkID: This unique identifier allows it to be referenced by other chunks and allows its inclusion in themes.

In addition, two further attributes will comprise the chunk’s attribute set because of the functionality that they offer to the author:

version: A chunk belongs to a version hierarchy (explained further in Section 5.2.4 on page 107).

variant: This allows demonstration by difference. It allows a chunk to receive multiple implementations dependant on external factors, such as operating system (e.g, Win32 or Linux source code chunk) or language (e.g., English or Greek interface tutorial chunk) or display requirements (e.g., a segment of an XSLT stylesheet that formats cross-references receives many different implementations — the author is able then to trial a set of them before committing to one). It is usually the case that two chunk variants would not be included in the same theme document. Variant use is demonstrated in Section 6.3.6 on page 164.

The traditional chunk model’s simplification to a generic representation has positive implications, including, in no particular order, the following major ones:

user-defined chunk types: The author may attribute to a chunk any type necessary to represent its content. Authors can now definitively type all

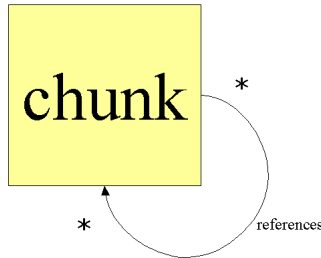


Figure 5.8: A chunk may reference, and be referenced by, zero to many chunks. A chunk may not reference itself.

manner of chunks. These types can follow a notation convention such as that suggested by UML — sequence diagrams and class diagrams can be developed by aptly typed chunks, for example.

chunk reuse: Because any type of chunk in our model can be nested and therefore referenced, it is possible to reuse any chunk by *referencing the same chunk* twice (or more) in the same theme. As a consequence, TBLP allows authors to display any type of chunk’s content multiple times within the same literate document (Section 49 on page 75 further explains why reuse is not possible in the traditional LP model).

theme-based chunk formatting: A chunk, throughout a theme, can receive specific type-dependant formatting. For instance, a construction documentation chunk is marked up in 10 point, Times, serif font for presentation in a user-story theme; however, it would sensibly occur as ‘raw’, unformatted text delimited by programming-language specific comment syntax when included in the source code presented to a compiler. Existing LP tools do not offer this functionality⁵. For example, an excerpt of the `wc.nw` literate program:

⁵ Although it is possible to develop a filter in `Noweb` to generate such a tool. Other tools that offer macro support allow the output of documentation chunks in the source code; however, not the individual treatment of the chunk’s content.

@

The `[[status]]` variable will tell the operating system if the run was successful or not, and `[[prog_name]]` is used in case there's an error message to be printed.

```
<<Definitions>>=
#define OK                0
    /* status code for successful run */
#define usage_error      1
    /* status code for improper syntax */
#define cannot_open_file 2
    /* status code for file access error */
<<Global variables>>=
int status = OK;
    /* exit status of command, initially OK */
char *prog_name;
    /* who we are */
```

could be presented as the following source code using TBLP (note that the documentation chunk is now presented as a source code comment):

```

/*
 * The [[status]] variable will tell the operating
 * system if the run was successful or not, and
 * [[prog_name]] is used in case there's
 * an error message to be printed.
 */
int status = OK;
    /* exit status of command, initially OK */
char *prog_name;
    /* who we are */

```

TBLP facilitates the visual differentiation of distinct chunk types. A requirements specification chunk can be presented in an emphasised font and receive a framed border, for example. The supporting architectural specification can be presented as a partial class diagram with supporting documentation to elaborate design decisions. This occurs by transforming chunk content to markup languages such as XMI (XML Metadata Interchange), which can be further processed to generate UML diagram notation. These processes can be integrated with the TBLP development process as described further in Section 6.2 on page 117.

Traditional LP development cannot elegantly present and differentiate between such hierarchical layers of documentation (higher-order documentation). The generic chunk facilitates the elegant distinction of layers of abstraction.

multi-formatted themes: Each theme in our model can be formatted in any number of ways according to rules imposed by each of a given set of stylesheets. As an example, it is possible to generate a source code theme to present to the compiler. The same source code theme can also be formatted using a different stylesheet, and presented as an HTML document for internet browsing.

This is a benefit of an extended processing model (described in Chapter 6, Section 6.2). Our processing model utilises XML and supporting technologies such as XSLT to separate content development from formatting. The TBLP processing model thus considers theme development and theme formatting as two distinctly separate operations.

positional chunk formatting: A chunk can receive specific formatting given the combination of its type and its relative position to other chunks of a given type and position, in the chunk hierarchy.

Thus, It is possible, for example, to apply differentiated formatting amongst nested code chunks, in order to explicitly illustrate chunk nesting. Section 7.2.2) demonstrates how this is achieved.

Such visualisation methods can be extended to provide useful abstractions literate themes. Chunk relationships, chunk types, chunk (content) size, the number of chunk references made from/to a chunk (reminiscent of Henry and Kanfura’s fan in/fan out [34]), or combinations thereof, can be represented using colours, shapes, movement, and distance, in a virtual environment, for example.

centralised processing model: Each chunk contained within a theme can undergo a mixture of the traditional weave or tangle operations, thus solving the biased processing model (see Section 49 on page 75). Our new processing model is discussed further in Section 5.2.6 on page 108.

the need for a literate aware tool: TBLP treats chunk development and theme development as two separate processes. Themes consist of an ordered list of nested references to chunks that exist in the repository. The author of a literate theme must organise and define the nature of these references (discussed in Section 5.2.5), whilst simultaneously utilising the repository, a separate document, to define chunks that are referenced.

Themes, as collections of nested references, are difficult to visualise. Furthermore, the continuous movement between literate themes and

the repository introduces a tedious sequence of actions. Traditional command-based system functionalities employed by existing LP tools do not offer suitable functionality for TBLP development.

Tool support alleviates these shortcomings associated with a manual implementation system. As chunks are manipulated and re-oriented within a theme, and implemented in the repository, a TBLP tool should offer dynamic update of a theme's display. Whether such a tool distinguishes between theme development and repository (chunk) development outright, or hides this distinction such that chunk development is combined with theme development (a theme is presented as though it contains free text) is a subject for further research.

Future research should also address whether such a tool is presented in the form of an IDE or integrated with existing IDEs (such as Microsoft's .NET development environment⁶) or editors (such as Emacs). Database development environments, such as Jade⁷, are also promising options. We propose that TBLP needs to be a supported process rather than an imposed one. Thus, modularised IDEs that are adapted through programmable API's, will be prime candidates for TBLP adaptation.

We next discuss the storage issues of the generic chunk and their implications.

5.2.3 *Chunk Representation*

It is common for chunks to be collated in a common physical proximity, as a group, in order to describe the same abstraction. These chunks, which may be of any type, form an atomic chunk. Thus, an atomic chunk is composed of a set of chunks that may each be of a different type, which describe the same abstraction. Each of these chunks may be further composed of a set of variously typed chunks representing a specific abstraction. We utilise the composite pattern [29] to express this relation, as shown in Figure 5.9 on the next page.

⁶ <http://www.microsoft.com/net/>

⁷ <http://www.jade.co.nz/>

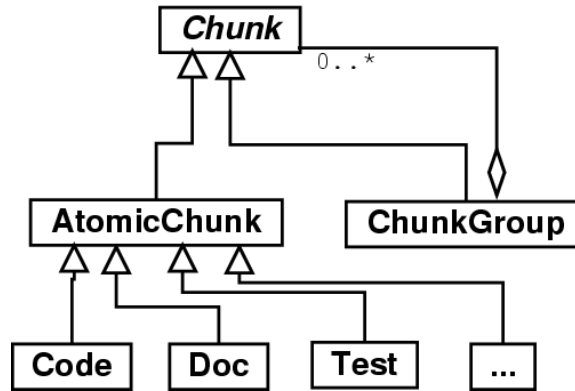


Figure 5.9: The new chunk model.

Whether this atomic unit is explicitly represented (by a chunk), or implied (by grouping chunks in close physical sibling-like proximity), as has been the case in existing literate tools, is a matter for further research of the chunk model’s representation.

Furthermore, an extended chunk model, likely to result from further research, would facilitate sibling relationships such that chunks that do not compose the body of the parent may be included as fostered siblings of child chunks. Another candidate method of providing this functionality is to allow the definition of parent-child relationships, thus allow the inclusion/exclusion of chunks based on theme-specific instructions. It may well occur that an enhanced theme model will suffice.

5.2.4 The Repository of Chunks

Every chunk is stored in a repository. The ordering of chunks in the repository does not affect chunk or theme development. A repository may be thought of as a database of chunks. Any chunk may reference any other chunk in the repository. Recursive references are disallowed, however.

A chunk’s storage in a repository is distinct from its inclusion in a theme. Thus, a chunk may exist in a repository without being included in a single theme. Furthermore, a chunk can be implemented without necessarily existing in a theme document.

Granular Revision Control

As chunks are progressively altered and improved, providing a mechanism that maintains each chunk's state at each stage of its development produces a powerful revision control system. Thus, if all chunk revisions are stored in the repository and are referencable (and therefore considered as chunks outright) all manner of historical theme representations are enabled.

In this manner, TBLP improves on traditional, file-based revision control systems. Using the chunk model, it is possible to compose themes from any version of any chunk. Themes may be rolled-back to stages of specific chunk development for example. Historical themes that show a chunk's progressive development are easily compiled to generate theme document.

In order to facilitate chunk-based revision control, the chunk must contain a **version** attribute. Also useful, although not critical, is a **time** attribute that specifies the time a new version of the chunk was created. Chunk versions are discussed further in Section 86 on page 162), while Section 7.5 on page 195 provides details of their implementation and necessity.

5.2.5 Theme Model

A theme is an ordered collection of references to chunks in the repository. Note that a theme itself does not contain free text. Chunks, which are defined in the repository, are threaded together as references to generate a literate document. A theme document bears similarities to the traditional web source. As with the traditional web, a theme document contains lexically ordered chunks. Each of these chunk references can reference a set of chunk references also. This can continue recursively to create hierarchies of chunk references. Nested references are reflective of a chunk's composition as it exists in the repository.

For each chunk reference, the theme model maintains a set of display attributes that dictate the chunk's appearance in the theme document.

The manner in which chunks within a theme are processed is dependent on the processing model. We discuss the requirements of the amalgamated weave and tangle processing model in the next section.

5.2.6 Processing Model: Blending Weave and Tangle

As discussed in Section 49, the traditional LP processing model is unable to equally process all chunk types in both the weave and tangle operations; it is impossible to tangle documentation chunks into the tangled source. Thus, the tangled scope and woven scope differ. Our TBLP chunk processing model overcomes this drawback by allowing both the execution of the weave and tangle operations in one theme document.

The generic chunk allows any chunk type to be nested. Therefore, what was previously considered a woven document, which was effectively a sequential translation of the web to the literate document, can now also undergo tangling operations to present nested documentation chunks. *The ability to nest all chunk types makes the process of tangling pervasive throughout the weaving of a document*, and the reverse is also true.

The amalgamation of the weave and tangle operations creates the need for discretionary control of the display of chunks in a theme. It is not always pertinent to display a chunk's content in occurrence in a theme, neither is it always pertinent to display a chunk's name each time it occurs in a theme.

partial weave and partial tangle Thus, documents may undergo both *partial* tangle and *partial* weave operations. These operations are dependent on the instructions contained in the theme document.

Thus, the author may stipulate the presentation of each chunk's occurrence in the theme document using this a simple rule-set. It may be feasible to further decouple these rules such that a chunk's content is hidden, but the referenced chunks it contains are displayed. This would allow something similar to a table of contents to be developed as a theme rather than achieve the same result via document post-processing. Perhaps future research will implement this functionality.

Theme Composition

top-level chunks A theme is composed of an ordered collection of nested chunk references. We name such an ordered collection⁸ 'top-level' chunks.

⁸ we borrow from XML terminology

We can think of this collection of chunk references as having a null root (the theme beginning). The restriction on this root chunk, however, is that it cannot contain free text. This maintains the difference between theme development, and chunk development, conceptually clear and distinct. Each chunk reference refers to a chunk that exists in the repository.

The Blending of Weave and Tangle

In this section, we emphasise the difference between LP and TBLP by comparing processing models. Although theme composition is the process of ordering a collection of chunk nested references, further instructions about the processing of these chunk references must be provided.

The traditional processing model consists of a tangle and weave operation. Tangling consists of recursively displaying a nested code chunk's content. A (simple) weave operation presents the content of a chunk and outputs the names of the referenced (nested) chunks contained therein.

In contrast, TBLP essentially combines the recursive traversal of a nested chunk performed by a tangle operation with the ability to either display a chunk's content (as do the tangle and weave operations), or include the chunk's name (as does a weave operation). Inclusion of cross-references in

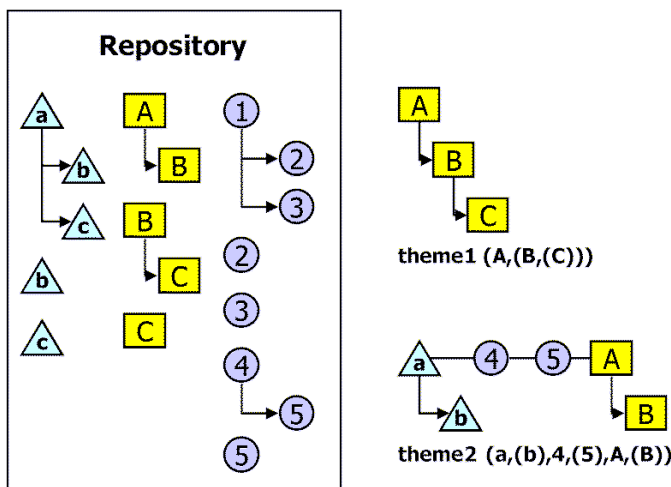


Figure 5.10: Multiple chunk types exist and all may be nested.

the generated document is also an option. Note that the display of a chunk's name, content, and cross-reference details may be performed even for nested chunks.

As an example, Figure 5.10 on the preceding page illustrates theme and chunk composition. The example illustrates that some chunks require something akin to a tangle operation, while others require something akin to a weave operation. Other combinations might also arise. The repository contains three chunk types, denoted by the triangular, rectangular, and circular shapes. The repository illustrates that chunk `<<a>>` references chunks `<>` and `<<c>>`; `<<A>>` references `<>`, which references chunk `<<C>>`; and `<<4>>` references `<<5>>`.

The theme composed from the repository orders chunks `<<a>>`, `<<4>>`, `<<5>>`, and `<<A>>` as the top-level chunks. Chunk `<>` is the only nested chunk of `<<a>>` included; chunk `<<c>>` is excluded from the theme document. Chunk `<<4>>` is included without including its nested chunk `<<5>>`. Chunk `<<5>>` is included as a top-level chunk. Chunk `<<A>>` is included, with chunk `<>`, however, chunk `<<C>>` is not included even though it is referenced by `<>` in the repository.

It is insufficient, however, to merely indicate a nested chunk's inclusion (or not) in a theme document. A chunk's presence may mean many things, including:

- a reference to the chunk's implementation elsewhere in the document, or in another theme.
- a display of the chunk's content.

The author must therefore also indicate each chunk's display properties. Combined with each reference, the following instructions pertaining to the chunk's display in the output document must also be given:

chunk content: Should the free text/content contained within the chunk be shown?

chunk name: Should the descriptive name used to label the chunk be displayed in the literate document?

```
[
  [a name=no cross-reference=no content=yes
    [b name=no cross-reference=no content=yes]
    [c name=no cross-reference=no content=no]
  ]
  [4 name=no cross-reference=no content=yes
    [5 name=no cross-reference=no content=yes]
  ]
  [A name=no cross-reference=no content=yes
    [B name=no cross-reference=no content=yes
      [C name=no cross-reference=no content=no]
    ]
  ]
]
```

Figure 5.11: Display instructions are required for each chunk reference in a theme document.

chunk cross-references: Should the chunk’s uses in this or other themes be displayed?

An example set of possible display instructions (in notation form) for each chunk is shown in Figure 5.11. The top level chunks are effectively presented as a list of nodes with accompanying instructions. A list is delimited by the ‘[’ and ‘]’ characters. A nested chunk is presented as a member of a parent chunk’s sublist. Thus, the set of theme-specific instructions can be accurately represented as a list of lists.

An example of a traditional tangle process, as depicted by theme **theme1**, in Figure 5.10, would require the following theme instruction set:

```
[
  [A name=no cross-reference=no content=yes
    [B name=no cross-reference=no content=yes
      [C name=no cross-reference=no content=yes]
    ]
  ]
]
```

The root chunk, <<A>> is the only top-level chunk included in the theme document. Chunk <<A>>'s content is displayed; its name and cross-reference details are not. Each nested chunk that composes <<A>> is recursively processed — only the free text is displayed.

An example of the traditional weave process, as illustrated by theme `theme2` in Figure 5.10 is as follows:

```
[
  [b name=no cross-reference=no content=yes]
  [4 name=yes cross-reference=yes content=yes
    [5 name=yes cross-reference=yes content=no]
  ]
  [c name=no cross-reference=no content=no]
  [5 name=yes cross-reference=yes content=yes]
]
```

Note that chunks <> and <<c>> act as documentation, and are therefore associated with chunks <<4>> and <<5>>, respectively. Also note that the name and cross-reference details are not displayed for the documentation chunks. This remains consistent with the display generated for traditional woven documents. Note that chunk <<5>> appears as a nested code chunk of <<4>>; however, its content is not displayed — only its name and cross-reference details are. The cross-reference details displayed in the theme document will allow navigation to the place in the document where chunk <<5>> is defined.

5.3 TBLP Development Emphasises Expression over Development

TBLP separates content from ordering (see Section 5.2.1 on page 90) provides a clear distinction, unlike LP, between chunk development and theme development. Although chunks may still be developed in a psychological order, TBLP also facilitates the development of themes by reusing existing chunks. Emphasis is placed, therefore, on expression of concept rather than development of concept.

Although expression and development will never be completely separated (and nor should they, we must be mindful of LP's promotion of thinking before developing), development with a mind for reuse of chunks will not only create more cohesive and loosely coupled chunks, but also allow the author to concentrate on theme-expression.

5.4 *Equality of Concerns*

All documents are generated from themes. Just as a document that follows a psychological order of expression is considered a theme, so is the source code that is fed to the computer. The new processing model treats all documents as equal products. The source code is no longer relegated to being a 'tangled' source. All documents are viewed as being of equal import, and all documents are considered 'literate' to a specific audience.

This leads to an interesting phenomenon of TBLP — it is possible to program in the traditional order dictated by the compiler, thereby avoiding the need to program in a 'literate' fashion — after all, source code is just another theme. Thus, any view that may be generated by a theme, is also one in which a document may be edited.

5.5 *Multiple Distributed Webs*

Although the focus of our discussion has been oriented towards a single repository and the composition of themes from this repository, we stress that multi-web TBLP is key to solving issues of source reuse — documentation, formalised notations such as UML, and source code⁹ are all examples of potentially reusable chunks.

Essentially, multi-web TBLP allows the composition of chunks from many sources to create one theme. The concepts described within this chapter are applicable to distributed and multi-web literate programs. Although not demonstrated in the TBLP implementation chapter (Chapter 6), there is scope certainly for consideration and development of distributed webs during

⁹ we do not refer to modular reuse such as replacing the need for shared libraries

further academic endeavour — we therefore place this in the future work basket (discussed further in Section 9.2).

5.6 Summary

We have developed a generic chunk model that allows non-discriminatory processing, thus developing the paradigm of, what we term, theme-based literate programming.

We derived the generic chunk model by allowing the separation of chunk content and their ordering in a theme document. We were thus able to develop multiple themes with chunks psychologically ordered appropriate to audiences' comprehension. We also allow for each nested chunk reference in a theme, the author to stipulate whether the referenced chunk's name, content, and cross-reference details are displayed.

TBLP solves many of the shortcomings associated with traditional tools; certainly most of the model-centered shortcomings. Specifically, in this chapter, we have illustrated that TBLP, through the generic chunk model (allowing arbitrarily typed chunks) allows the distinction of all types of chunks (SQL, test source, unit test, user story, UML, requirements analysis, thoughts on requirements, specification, are all examples of possible chunk types) — solving the indecision of chunk categorisation. Also, hierarchical, multi-layered chunks may be composed (to represent all phases of the SDLC for example). Thus, the ability to develop multi-typed, multi-layered chunks presents a viable solution to the tyranny of dominant decomposition.

In Chapter 6, we demonstrate an implementation of our TBLP model.

Chapter VI

Implementation of the Theme-Based Literate Programming Model

In theory, there is no difference between theory and practice. But, in practice, there is. — Jan L.A. van de Snepscheut

This chapter describes the practical implementation of the theme-based literate programming model as discussed in Chapter 5. The new processing model and chunk models lead to the design of a new data model, and this chapter is concerned with the realisation of these models as physical proofs of concept.

The sequence of explanation in this chapter follows the initial stages of the TBLP development process — the later stages are presented in Chapter 7. Firstly, in Section 6.2, we explain the development process, and then address the implementation and support of its distinct stages.

The initial development stage of TBLP includes chunk and theme composition (Section 6.2.1) The prototype tool, Context Based Development Environment (CBDE) facilitates this composition (Section 6.3). The CBDE was developed as a test framework for both TBLP environment development and the conceptual advance of TBLP programming.

The CBDE implementation's architecture and design is discussed in Section 6.4.

Throughout this chapter, prose that is presented in an enclosed box is used to demonstrate present or future design decisions of the CBDE's development.

6.1 *Why XML?*

XML is a popular and pervasive markup language used for data presentation and storage. It is selected as the representation medium for TBLP because it provides:

ubiquitous support: XML is supported pervasively throughout the world wide web, open source, commercial, programming, and document authoring communities. It is fast becoming the language of choice for information storage, and data communications. Many software development environments provide XML capabilities. Main-stream database support is also growing (<http://www.rpbouret.com/xml/XMLAndDatabases.htm>). Support, in the form of extended libraries, is available from many programming languages. This support continues to grow.

XML provides technologies such as DOM, Schema, SAX, XSLT and XSLFO. These are powerful parsing, storage, and transformation agents that exist as cross-platform implementations. Markup languages such as DocBook provide ready-to-use scientific documentation markup.

Future support is likely to benefit TBLP also. Navigation between chunks and their display could be facilitated by XLink and XPointer aware browsers. Further research will determine the usefulness of this approach.

XML standards are developed by the W3C (World Wide Web Consortium¹), a vendor-unspecific entity focused on developing the technical evolution of the “web”.

Moreover, it seems clear that XML is not simply a passing fad. It’s wide acceptance has effectively etched its use into, at least, the short-term future.

extensibility: Because XML separates content and formatting, documents are easily transformed, and therefore processable and transferable between XML-enabled processes and applications. This makes XML a

¹ <http://www.w3c.org/>

strongly feasible option for interfacing to each stage in the TBLP development framework (see Section 6.2).

The implementation of the individual processes themselves become black-box, modularised entities — importantly they must adhere to the process-interface requirements². Such an approach facilitates experimentation, which is an important consideration in an academic endeavour such as this. It also promotes a plug-in approach whereby processes may be inserted and/or replaced without disturbing other processes in the pipeline.

XML enables document transformation to multiple document formats, and thus, is unaffected by the monolithic output problem (see Section 4.1.3 on page 64) that affects most non-XML LP tools.

human-legibility: XML can be written manually, or partially/fully automated through the use of conforming utilities. And although it is terse, XML is human-readable.

Furthermore, XML helps alleviate the 3-syntax problem (see Section 4.1.2 on page 62). Users need only know two syntaxes — the programming language and the relevant XML markup. Take note that authors are not confined to mark chunk content up in XML. It is quite possible to utilise other markup languages, such as \LaTeX for example.

validity and verifiability: XML documents can be checked for validity and verifiability.

6.2 *The Literate Document Development Process*

The literate document development process, as illustrated in Figure 6.1 on the following page, employs a pipelined architecture that allows the inclusion of sub-processes at any stage. The process of document development forms the basis of a framework for themed literate document development — in-

² as far as the TBLP development framework is concerned

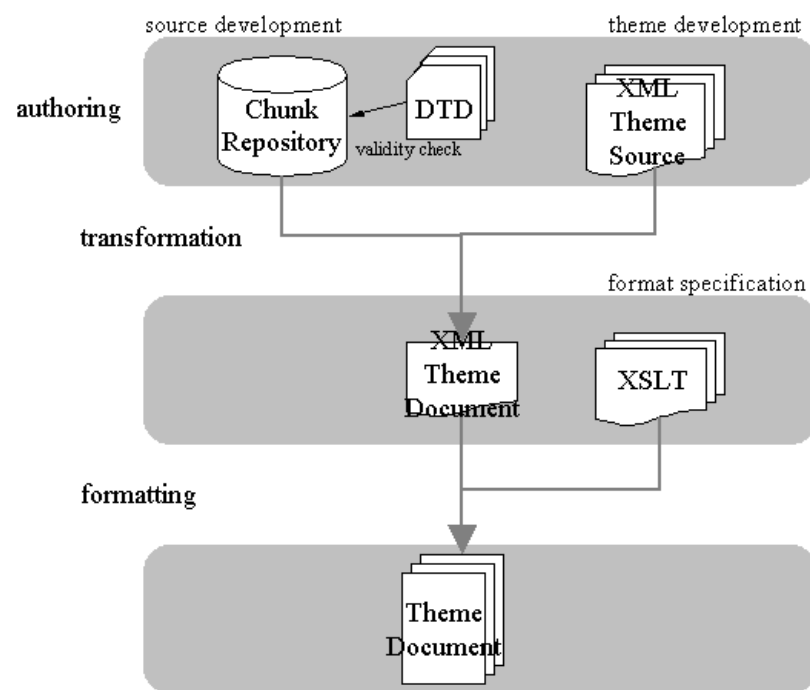


Figure 6.1: Literate documents are generated using the TBLP Development Process framework.

terconnected, navigatable, formatted documents. The distinct stages in the document development framework are:

1. **Authoring:**

- **Repository/chunk development:** individual chunk content added as combinations of (1) free text and (2) references to other chunks.
 - **DTD development:** each chunk conforms to a given DTD.
 - **XML theme source document development:** chunks from the repository are referenced. The XML theme source documents are therefore linked to the repository. Chunk references are ordered. XML theme source documents contain no free text.
2. **Transformation — XML theme document output:** the references made in the theme source document are resolved against the repository. The result is an XML document with full content, presentation, and formatting instructions.
 3. **Document formatting:** The XML theme document is parsed using an XSLT stylesheet(s) to generate a literate document(s) in the user-specified format.

There is a clear conceptual and pragmatic distinction between chunk development, theme development, document content, and document formatting. Although two or more of the distinct development stages may be integrated in future TBLP implementations (such as combining theme editing with chunk editing to allow in-theme chunk editing), we design the CBDE to enforce their separation (Section 6.3.4 explains why the current prototype doesn't support in-theme editing).

The distinction between each of these stages has many consequences. The most significant are as follows:

the separation of theme composition and chunk composition: Multiple theme documents can be composed from a single repository, as discussed in Section 5.2.1 on page 90.

the separation of chunk display and chunk formatting: Although a chunk — its name, cross-references, and content — may be marked up for future formatting, it is possible to display or prevent the display of any one of these items (explained in Chapter 5, Section 5.2.6 on page 108).

separation of content and formatting: Any XML theme document may be transformed by multiple XSLT stylesheets. This allows multiple themed literate documents to be generated, each differentiated by its formatting.

pre/post/intermediate processing: The pipeline process employed allows sub-processes (other tools or technologies) to be inserted at any stage in the pipeline. This is similar to the successful extensible approach adopted by **Noweb** and promotes the development of modularised processes and support utilities. For example, in order to validate XML documents at all stages of the development process, it is possible to include a DTD validity checker to ensure document conformance.

process replacement: It is possible to replace any stage of our TBLP development process, so long as the newly inserted process adheres to the XML-based interface requirements of its surrounding processes. For example, in order to generate PDF documents, it is possible to replace XSLT with XSLFO³ (or use XSLFO as a post-processor of XSLT's output). It is therefore possible to replace the chunk-authoring and theme-authoring environments (such as the CBDE) with other environments.

Each stage in the document development framework interfaces to the previous and next stage. These interfaces *must* be adhered to in order to facilitate process replacement and intermediate processing. These interfaces are formed by the XML expected by each process. We discuss these in the following sections.

The document development process employed by TBLP is similar to the processing employed by existing XML-based publishing frameworks such

³<http://www.w3.org/TR/xsl/slice6.html> — forms part of the XSL standard.

as Cocoon (<http://xml.apache.org/cocoon/>) and Axkit (<http://axkit.org/>). We offer the basis of a framework for document publication as well, however, we also provide a framework for document composition. Document composition occurs in stage 1 in Figure 6.1.

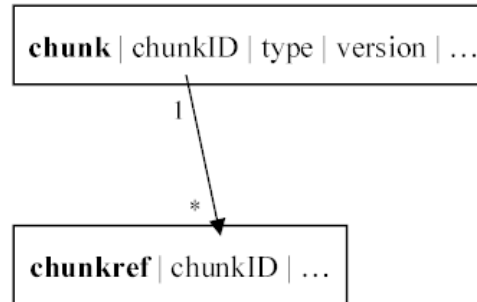
6.2.1 The Repository — Chunk Composition

Chunk and theme composition are explained in Chapter 5, Sections 5.2.2 on page 96 and 5.2.6 on page 108, respectively. Briefly summarising: the development stage involves both the population of the repository and the generation of themes that reference these chunks in the repository. A theme cannot contain free text; it can only contain chunk references. A chunk may contain any number of chunk references. Any chunk reference that is nested in a theme document must exist, as a chunk, in the repository; any nested chunk reference reflects the referenced chunk’s parent’s content in the repository.

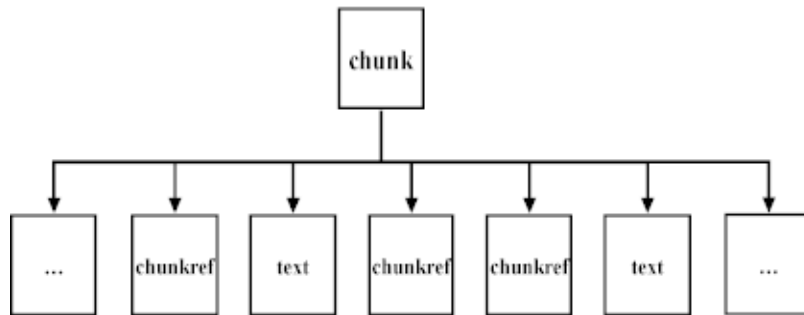
The traditional literate model discriminated between a code chunk and a documentation chunk. To provide equality of functionality to all chunks, Section 5.2.2 on page 98 introduced our new concept of a generic chunk. That is, the body of *any* type of chunk can be implemented by reference much like the body of a code chunk in the traditional chunk model. A chunk comprises the free-text that it possesses, plus indirectly, that of any chunks that it *references*, illustrated in Figure 6.2(a) on the following page. Chunk composition occurs in the repository. This is illustrated in Figure 6.2(a) and modelled using a DTD in Figure 6.3 on the next page.

The DTD stipulates that a repository document may contain any number of **chunks**. Chunks may contain textual data (`#PCDATA`) and any number of chunk references (`chunkrefs`). A **chunk** must possess a unique identifier (`chunkID`). A `chunkref` references a **chunk** by the `chunkID` attribute — a chunk reference’s foreign key (see Section 7.4 on page 193 for an explanation of its implications on TBLP).

Both elements (`chunk` and `chunkref`) have a constraint stipulating that a `chunkID` is mandatory. It is incorrect to reference a non-existent chunk.



(a) A data model of a chunk. It may reference zero to many chunks. A chunk may not reference itself.



(b) A chunk is composed of free text and chunk references.

Figure 6.2: A data model and composition representation of the chunk.

```
<!ELEMENT repository (chunk)*>

<!ELEMENT chunk (#PCDATA | chunkref)*>
<!ATTLIST chunk chunkID      ID #REQUIRED>
<!ATTLIST chunk version      CDATA #REQUIRED>
<!ATTLIST chunk chunkName    CDATA #IMPLIED>
<!ATTLIST chunk type          CDATA #IMPLIED>
<!ATTLIST chunk variant      CDATA #IMPLIED>

<!ELEMENT chunkref>
<!ATTLIST chunkref chunkID    IDREF #REQUIRED>
```

Figure 6.3: A repository DTD.

Furthermore, a chunk must possess a **version** number⁴ The **chunkName**, **type**, and **variant** attributes can contain null values — the default for these attributes implied by the DTD.

To allow nested documentation⁵, LP tools such as LPML (Section 3.4) have seen the inclusion of an additional element to the DTD, named *documentation* (or similar). Based upon this method of chunk model extension, including different chunk types would require an additional entry in the DTD for each supported chunk. An explosion of chunk element types would likewise see an explosion in the DTD ruleset — maintainability is negatively affected. Contrastingly, because our model requires that any type of chunk can be nested, we impose the same nesting ability of a traditional code chunk upon *all* chunk types. Therefore, the document author need only stipulate the chunk's *type* as an attribute of the chunk — thereby implying the creation of a new chunk type⁶. The chunk **type**, for example, may be a traditional **code** or **document** chunk. Or, it is likely to contain a more specific description, such as **test code**, **requirements specification**, or **interface bug**.

An example will prove more enlightening than the verbosity of our explanation. We employ the DTD to compile a small repository of chunks; presented in Figure 6.4.

There are seven chunks in this repository. Two are named `<<define printIterator>>`, two are named `<<initialise variables>>`, and three are named `<<loop and print>>` — each group of same-named chunks possessing chunks of type “construction documentation” (**cons-doc**) and “C source code” (**c-code**). For example, although chunks with **chunkIDs** 1 and 2 possess the same name, they are of different types, and have dissimilar implementations.

Chunk composition by reference is demonstrated in this example by chunks `<<2>>` and `<<3>>`. Chunk `<<2>>` demonstrates the traditional reference that code chunks make — it references chunks `<<6>>` and `<<4>>`. Chunk `<<3>>`

⁴ The DTD stipulates that a version may have any character data; however, we define a version number as containing numbers and delimiting decimal (‘.’) characters.

⁵ albeit, nesting to two levels

⁶ although these types may be restricted by stipulating a limited chunk type set — a useful approach to restrict the over-definition of chunk types



Figure 6.4: A repository as it is represented using the XML DTD in Figure 6.3. An abstract representation (left) shows chunk references made.

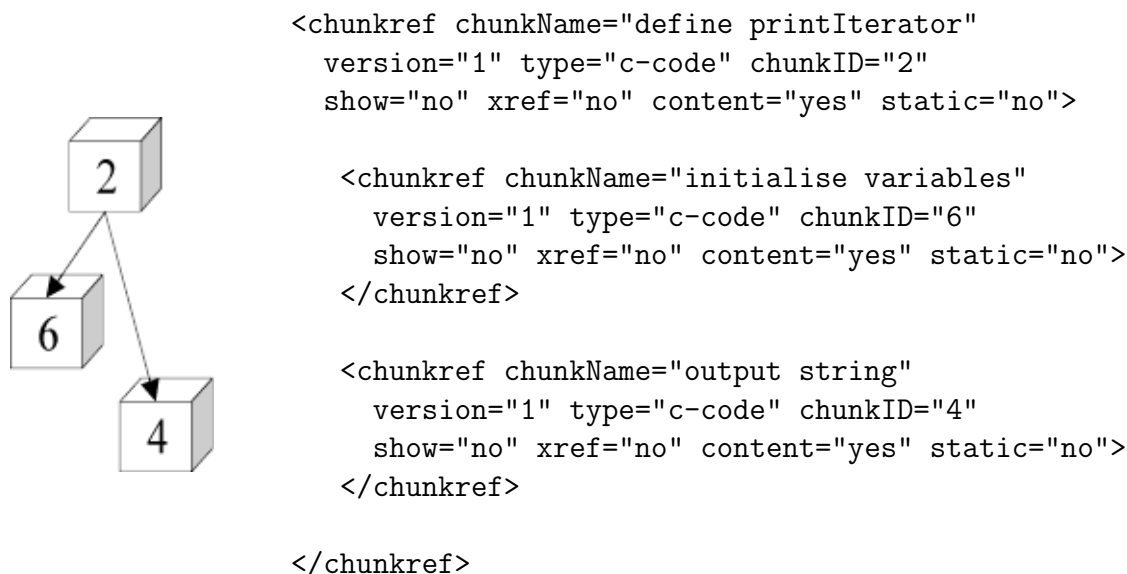


Figure 6.5: An XML theme source document contains only chunk references. No free-text exists (white-space is ignored). The accompanying abstract diagram illustrates the nested tree structure generated by chunk references.

illustrates that it is possible to compose documentation chunks in the same manner — it references chunk <<7>>.

The <<loop and print>> doc chunk illustrates that nesting may also be achieved using documentation chunks. The `chunkref` element within the <<loop and print>> chunk refers to the <<loop and print2>> doc chunk, which contains a textual description.

6.2.2 XML Theme Source Document

From the repository, let us develop a theme with content equivalent to a traditional tangled document — program source code. The theme source document contains only references to the chunks in the repository. It uses the same notation to reference chunks as the repository document does. The difference is that the theme source document is theme-specific; thus, each chunk reference also contains presentation instructions. These are the `show`, `xref`, `content`, and `static` attributes (Section 6.3.2 on page 138 discusses their functionality further). Figure 6.5 illustrates this. From this theme source, we derive the theme DTD in Figure 6.6 on the next page.

```

<!ENTITY % switch.atts " ( yes | no ) ">

<!ELEMENT chunkref ( chunkref *)>
  <!--ATTLIST chunkref chunkID      IDREF #REQUIRED-->
  <!--ATTLIST chunkref version      CDATA #REQUIRED-->
  <!--ATTLIST chunkref chunkName    CDATA #IMPLIED-->
  <!--ATTLIST chunkref type          CDATA #IMPLIED-->
  <!--ATTLIST chunkref variant       CDATA #IMPLIED-->
  <!--ATTLIST chunkref xref          %switch.atts "no"-->
  <!--ATTLIST chunkref show          %switch.atts "no"-->
  <!--ATTLIST chunkref content       %switch.atts "no"-->

```

Figure 6.6: A theme source document DTD.

The content of this theme is destined for compilation by a C compiler. Note that all the **content** attributes are set to **yes** indicating that the free text of a chunk should be output. The **xref** and **show** attributes, however, are set to **no**. This prevents the output of cross-reference details and the chunk's name together with the source code. It is pointless and damaging to output the chunk-name and its cross-reference details to a compiler.

Note that three levels of nesting are present in the theme document. Chunk <<1>> references <<2>> and <<3>>. <<3>> references <<4>>. This nesting is reflective of the nesting of the related chunks in the repository.

Within the theme source document, chunks can be associated by nesting or by proximity. This source document excerpt illustrates only association by nesting; association by proximity is illustrated in the excerpt contained in Figure 6.7 on the facing page, where a construction documentation chunk is utilised to describe a code chunk. Figure 6.8 on the next page abstractly depicts this relationship.

Although it is feasible to manually produce an XSLT stylesheet that transforms an XML theme source document and its supporting repository into an XML theme document (discussed in Section 7.1 on page 176), authoring tools are required to ease this stage of the document development process. For this

```

<chunkref chunkID="1" chunkName="define printIterator" type="cons-doc"
  version="1" show="no" xref="no" content="yes" static="no">
<chunkref chunkID="2" chunkName="define printIterator" type="c-code"
  version="1" show="no" xref="no" content="yes" static="no">

```

Figure 6.7: Association by proximity occurs when two chunk references occur as siblings and represent the same unit of abstraction.

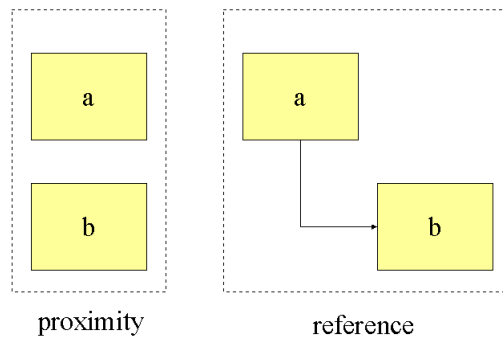


Figure 6.8: Association by proximity and association by reference. Both are valid means of association in the new LP model.

purpose, the CBDE, presented in the next section, is utilised.

6.3 The Context-Based Development Environment

The context-based development environment (CBDE) facilitates the authoring of documents associated with the first stage of the theme-based literate programming document development process. The CBDE is by no means reflective of a commercial or final end-user product. It is an archetypical, prototypical, TBLP tool designed and implemented as:

- an expression of the concept of TBLP, and
- a testbed for further development of TBLP.

TBLP provides a flexible document development environment. The chunk model allows any chunk to be nested within any other chunk. Potentially, it is a confusing environment. The dilemma is that we are enabling a state whereby we say “You can do whatever you like.” However, we continue the instruction: “But we would prefer that you did it this way.” The functionality of suggestive guidance, which lies largely in the domain of human-computer interaction (HCI), is difficult to attain, but can be adequately facilitated by the following interface design criteria (an adaptation of Cockburn and Churcher’s [20] criteria):

support visualisations of the literate program structure: Both high and low level abstractions of the literate themes and repository are necessary for LP comprehension. Moreover, support for visualisation extensions, such as virtual 3D world representations of themed literate programs, also aid comprehension and analysis of theme-specific statistics.

be an equal opportunity interface: Updates to chunks and themes should update all related interface widgets⁷ simultaneously.

⁷ the term **widget** is used commonly throughout the Tk library to describe a graphical user interface component. We will use the same terminology throughout this chapter.

have minimal syntactic requirements: XML is not suitable for manual input, nor is it suitably readable. A TBLP tool should ease the input and readability of XML-based chunks.

contain tools for literate browsing: Navigation throughout themes should be facilitated by the literate tool.

offer support for plug-in literate tools: Tools that help generate chunk content from external entities — for example, a tool that parses and converts compiler messages into chunks used by source-code testing themes should be supported.

offer non-intrusive support: All programmers will not want to use an LP approach to software development. The system should therefore not force a literate style upon the programmer⁸. It turns out that this capability is inherently supported by the TBLP model (source code is ‘just another theme’. We mention it, however, for completeness.

Our archetypical implementation serves as a starting point for the investigation of these concerns. The CBDE interface is developed as a reflection of TBLP’s two fundamental areas of functionality:

1. chunk editing — manipulation and inclusion of free text and chunk references, and
2. theme browsing — comprehension in context.

Note that the actions of *editing* and *browsing* are treated as distinct and separate operations because (1) there is a clean mapping between the processes of repository (chunk) development and theme composition (see Chapter 5, Sections 5.2.2 and 5.2.6), (2) it promotes comprehension in context (discussed in Section 6.3.4 on page 151), and (3) it forms well to the Model View Controller design pattern used to develop the CBDE (explained in Section 6.4 on page 168).

⁸ Contrary to Knuth’s approach to LP.

Thus, the CBDE allows the operations of chunk editing and theme browsing in physically separate areas. Figure 6.9 on the next page shows the text edit and theme browsing environments with Norman Ramsey’s `wc.nw` (converted and) loaded⁹. The text view widget on the lower right of the application is the editing pane. In this window, the programmer is able to edit (the content of) any given chunk. All other windows in the CBDE are used to browse theme/chunk information. Specifically, the display-oriented widgets are:

1. the repository widget (displayed in Figure 6.10) that floats next to the main CBDE window,
2. the tree view widget, and
3. the theme text view widget.

6.3.1 *The Repository Widget*

The repository widget, illustrated in Figure 6.10 on page 132, reflects the chunk repository’s contents. It presents a complete list of all chunks, each of which can be included in any theme.

Each row is dedicated to the display of the attribute values of one chunk. Specifically, the data displayed is:

- the chunk’s name,
- a drop-down list box of themes that a chunk is included in,
- a drop-down list box of chunk versions,
- the chunk’s type, and
- the chunk’s unique identification (`chunkID`).

⁹ Refer to Appendix C.0.1 on page 270 for the original literate document.

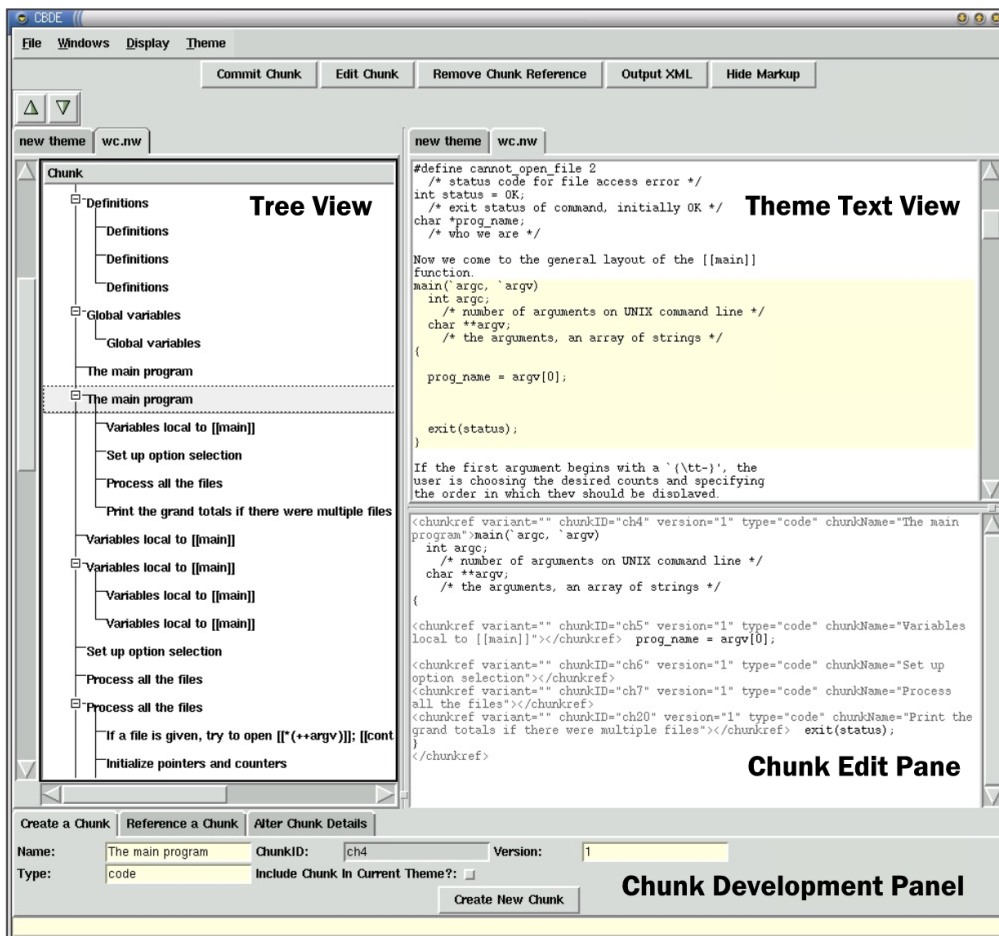


Figure 6.9: The CBDE environment with Norman Ramsey's Noweb version of `wc` loaded.

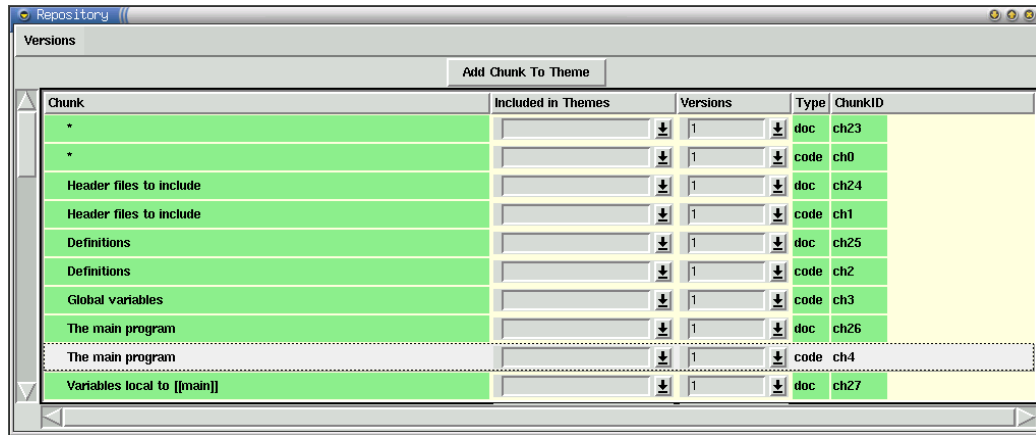


Figure 6.10: This CBDE repository, a separate window from the main CBDE interface, shows some of the chunks generated from developing the literate program in Figure 6.9 on the preceding page.

The repository widget has two modes of chunk display: (1) display each and every version of every chunk (as illustrated by Figure 6.12 on page 135), and, (2) display only the default chunk versions (as illustrated by Figure 6.10). Thus, the second mode saves on screen real estate and eases the locatability of chunks. The programmer is able to alter views by choosing the “Show Versions” option in the “Versions” menu. The version and theme list boxes allow greater functionality than the static display of data. Sections 6.3.1 and 7.6 investigate these two items respectively.

Chunk Versions

The CBDE facilitates chunk version development (as discussed in Section 7.5 on page 195). The progressive development of a chunk generates multiple versions, and each of these is stored in the chunk repository. Each version is considered a chunk in its own right (therefore possessing an automatically generated `chunkID`, see Section 7.4) and is physically grouped together with the chunk versions that comprise the same chunk version-tree in the repository widget. The order of display of chunk versions sorted in numerical order. (See Section 9.5 on page 198 for an example of chunk version hierarchies and

a description of the version numbering system.)

Figure 6.12 on page 135 illustrates an example where three chunks exist in the repository: chunks <<A>>, <>, and <<C>>. Note that chunk <<A>> has three versions; 1, 1.1, and 1.1.1, each with a unique `chunkID`; `ch1`, `ch4`, and `ch5`, respectively. `ch5` is the default version of the version-tree for chunk <<A>>. The default version is always highlighted with a green background¹⁰. A yellow background¹¹ denotes a non-default version. Either may be selected and included in a theme.

A default chunk version exists for each version-tree of chunks. It is usually the version that has been most recently edited and committed to the repository, however, any chunk version may be set as the default chunk version using the chunk development panel’s “Alter Chunk Details” tab (see Section 86).

While the repository is in default mode, it is possible to view the details of a non-default chunk version. By clicking on the chunk’s “Versions” drop-down list box, and clicking on the desired version, any chunk version can be displayed. All other versions of a chunk are then hidden.

Figures 75 on the next page, shows the selection of version 1 from the default state of version 1.2.

Default Versions

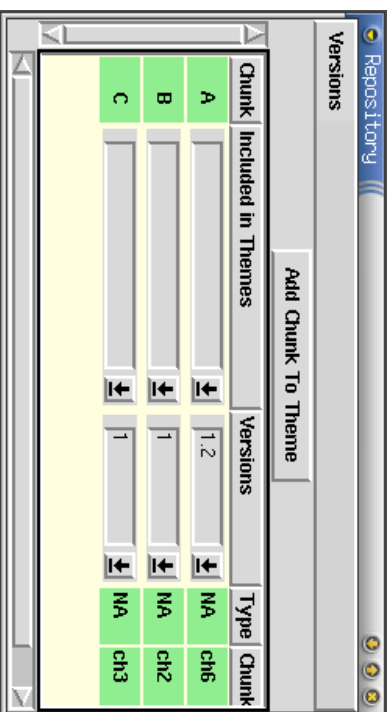
The default chunk version indicates to the programmer the development state of a chunk (because further development of a chunk should be performed by altering the default chunk) and, therefore, which chunk version should be included in a theme. Defaulting to previous versions can occur, and revision control branches can be created (in the mould of version control systems such as RCS (Revision Control System [84]) and CVS (Concurrent Versions System¹²)). Thus, it is possible to nominate a chunk version as the default of the chunk group.

Also, chunk editing creates a new chunk, with an incremented version

¹⁰ appearing as a dark shade of grey

¹¹ appearing as white

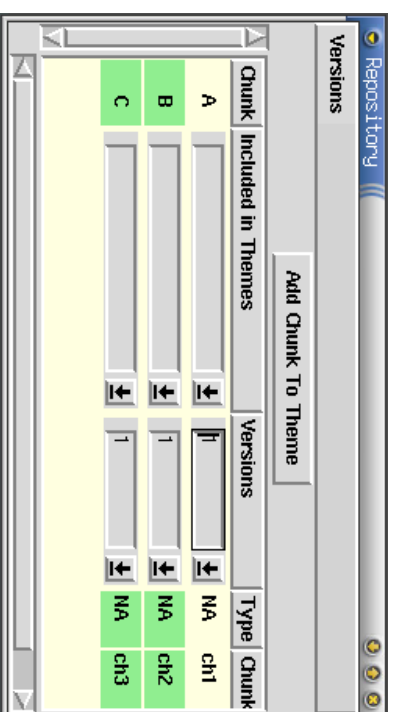
¹² <http://www.cvshome.org/>



(a) Chunk <<A>>'s default version 1.2 is highlighted.



(b) The drop-down list box reveals all available versions of chunk <<A>>



(c) Version 1 is selected. Note the non-highlighted background; version 1 is not the default chunk.

Figure 6.11: The repository widget provides the ability to hide and select versions of a given chunk.

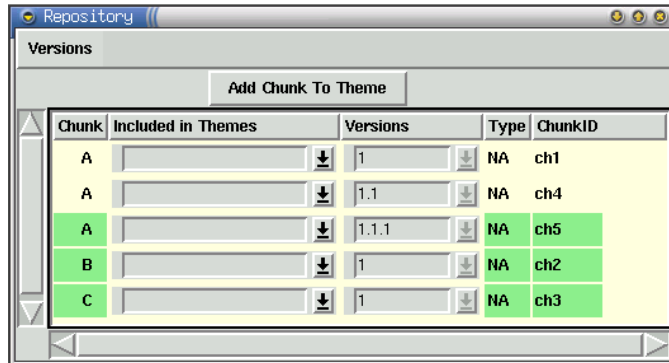


Figure 6.12: The CBDE repository showing three versions of chunk <<A>>; 1, 1.1, and 1.1.1. Version 1.1.1 with chunkID ch5 is the default chunk of the group.

number, which is automatically made the default. For instance, version 1.1 of chunk <<A>> is the default version. Altering version 1 of chunk <<A>> will cause a new chunk to be created with version 1.2 as its version number. Version 1.2 now becomes chunk <<A>>'s default version. Figure 6.13 on the following page shows the updated repository after performing this operation. Section 86 on page 162 explains how to perform chunk rollbacks using the CBDE, while Section 7.5 on page 195 discusses the need for versioning and how it is implemented.

Viewing Themes a Chunk is Included In

The drop-down list box in the “Included in Themes” column contains the list of all themes that a set of chunk versions is included in. Remember that in the default view, only the default chunk versions are displayed — all other versions in the group are hidden. Which chunk is included in the listed themes is ambiguous, so we have marked each theme with a preceding asterisk (*). Any themes in the drop-down list box without prepended asterisk’ include chunks of other versions in the group. Figure 6.14 shows a drop-down theme list box.

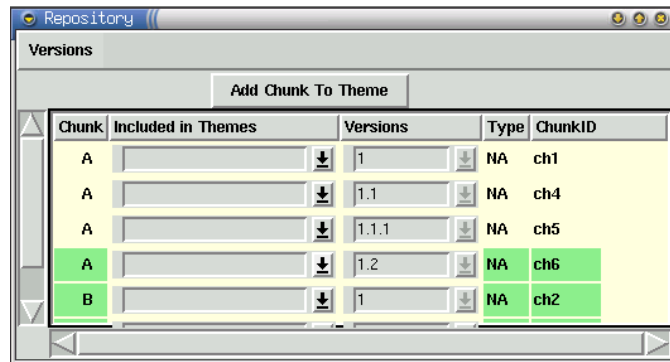


Figure 6.13: The CBDE repository showing version 1.2 of chunk <<A>> as the default.

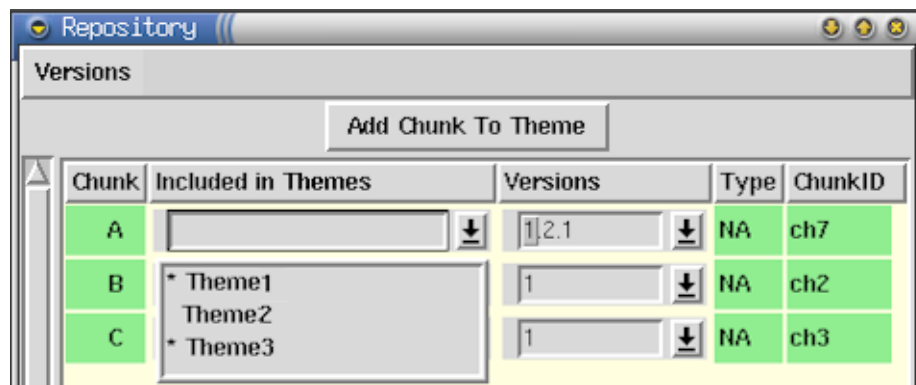


Figure 6.14: Chunk <<A>>, version 1.2.1 is included in Theme1 and Theme3, however, not in Theme2. Another chunk in the group of versions is included in Theme2.

It is a limitation of the **BrowseEntry** Tk widget that no other reasonable measures can be taken to indicate which themes a chunk version is included in. For instance, it would be convenient to use a colour scheme; perhaps blue to indicate the raised chunk is included in the theme, and grey, or a lighter shade of blue to indicate that another chunk version of the group is included in the theme. The **BrowseEntry** widget allows only textual content presented in a default font using a default size.

Clicking on any one of the themes in the drop-down list box will raise the respective theme tree (see Section 6.3.2 on the next page) and theme text view (see Section 6.3.3 on page 145) widgets, and also highlight the chunk’s occurrence in the given tree. Thus, the repository is able to satisfy such queries as:

“In which themes is chunk A referenced?”

- Clicking on chunk <<A>>’s theme list box will reveal the answer.

“In which themes is version 1.2 of chunk A referenced?”

- Selecting version 1.2 in the “Versions” list box of chunk <<A>>. Clicking on version 1.2’s theme list box will reveal the themes that reference version 1.2 (designated with an asterisk prepended to the theme name).

“Locate version 1.2 of chunk A in Theme1”

- Clicking on the theme that version 1.2 is included in will bring the Theme1 theme tree and Theme1 theme text view widgets to the fore. Version 1.2 of Chunk <<A>> will be highlighted in both widgets.

Including a Chunk in a Theme

Chunks may be selected singly or as a group using the mouse or keyboard hot-keys (pressing the space-bar when the relevant chunk is highlighted or using

the arrow buttons with the shift button held down) for inclusion in a theme. Selecting a range of chunks and pressing on the “Add Chunk To Theme” button will result in the chunks’ inclusion in a theme. Double-clicking on a specific chunk’s row has the same result.

Chunks are displayed in their order of development, however, a chunk’s `chunkID` does not necessarily indicate its relative time of development.

6.3.2 The Theme Tree Widget

The collection of theme trees is contained in a **Notebook** widget. A **Tk Notebook** widget can contain any number of pages. Each page is devoted to the display of a given theme tree. Specifically, an **HList** in the form of a scrolled tree (**ScrlTree** of the **Tree** widget set¹³) is used.

The tree view shows all chunks included in a theme as well as their nested structure. Figure 6.9 on page 131 shows an example of a theme tree; the left panel contains this tree widget. Chunk nesting (inclusion by reference) forms a tree of chunk references. As an example, chunk `<<A>>` references chunk `<>`. Chunk `<>` will therefore be displayed as a child of chunk `<<A>>`. Figure 6.16(a) on page 147 illustrates this scenario.

Alongside each chunk node in the tree exist four neighboring check-
theme-specific buttons that allow manipulation of the theme-specific attributes of a chunk:
attributes

1. show
2. xref
3. content
4. static

*The state of the **show**, **xref**, and **content** options translate directly to the chunk-display display attributes of a chunk in the literate document.* Clicking on these *attributes* checkbuttons will select (set to on), or de-select (set to off), the related option. It will also set the related XML attribute in the theme text widget

¹³ see `man Tk::Tree` in the Perl Tk documentation set

to “yes” (selected) and “no” (de-selected). Using this method, it is possible to control the display properties of each chunk referenced by a theme. The static button, which allows the author to prevent a chunk reference from being updated when edits, rollbacks, chunk version default changes occur, may also be selected or de-selected.

Sections 79 – 79 discuss these attributes in more detail.

Navigation and Chunk Selection

Each node in a tree represents a chunk reference made in a theme document. Clicking on the nodes of a tree highlights the respective chunk in the related text widget. In addition, the chunk’s details are displayed in the “Chunk Development” panel (Section 6.3.5 explains further). Figures 6.9 on page 131 and 6.18 on page 149 illustrate chunk selection. The tree view panel in Figure 6.9 shows the chunk <<The main program>> highlighted with a light-grey background. The theme text panel shows <<The main program>> chunk also with a highlighted background — this time lightyellow.

A mouse-click¹⁴ on a chunk in the tree widget automatically scrolls the theme *text* widget to the area where the chunk exists. The opposite is also the case: a mouse-click on a chunk in the text widget will automatically scroll and highlight the related chunk in the tree widget.

This function can be utilised as an effective navigation tool. Combined with the repository widget, and its ability to locate chunks in themes, (by clicking on the theme name in the themes drop-down list box), referenced chunks can be located easily.

navigation

Nested Chunk Hierarchies

In combination, the tree and text widgets can be used as effective abstraction agents. The tree widget provides a hierarchical (table of contents-style) theme representation; given well named chunks, it is possible to create a hierarchy that effectively tells the story of the literate document — in essence, a summarised view of a theme (Section 8.3.9 on page 212 describes this feature as important for the development of “good” literate programs). This

¹⁴ or press of the space-bar

feature is similar to the outlines approach used by tools such as **Leo** and Dan Schmidt’s emacs **Noweb**-outline mode (<http://www.dfan.org/real/noweb.html>)¹⁵. The textual view offered by the theme text widget (see Section 6.3.3) is referred to for content-level chunk detail. The hierarchy of chunks is not as prevalent in the theme text widget. The tree widget is an abstract replica of this hierarchy. Figure 6.3.3 on page 147 illustrates the difference between the hierarchical view of the theme-tree and the theme content browser. Even with such a simple example, one appreciates the clarity of the tree representation.

Atomic Chunk and Relationships

Chunks are currently represented as a hierarchical structure where there exists a direct parent-child relationship between chunks. Definition of explicit relationship types amongst chunks, however, are not currently supported by the chunk model and therefore, not offered by the CBDE. It is deemed pertinent to the growth of TBLP, however, to investigate an extended generic chunk model (Section 9.1 on page 222 further explains our initial thoughts about this extended model.).

Folding Tree View

Each sub-tree in the tree-hierarchy is either expanded or contracted, thus providing a folding tree view. The default is to display all nodes in the tree. By clicking the ‘-’ symbol to the left of the parent node, however, the node is set to a “closed” state. A closed state means that the chunk’s children (chunks that it references) are folded, or hidden. It is important to note that this feature does not affect the final literate document in any way, but is purely an interface display option.

¹⁵ Existing command line-based LP tools are unable to offer chunk outline views; a traditional-style literate document shows only nested chunks of an immediate code chunk. For example, it is impossible to view the grandchildren of a given chunk. Thus, tool-based facilities are required.

Note that we find this functionality to be rather unnecessary^a. It is enabled, however, due to its ease of implementation. If a programmer deems it necessary to use this functionality often, it is best to create a new theme that includes only the high-level chunks deemed important.

^a although future research will clarify

*The **show** Button*

The **show** check-button toggles the display of the chunk's name in the literate document. It is possible to prevent the chunk's name from appearing in the literate document (final output). As an example, the chunks `<<A>>`, `<>`, and `<<C>>` in are referenced in the three themes, **Theme1**, **Theme2**, and **Theme3** in Figure 6.15 on the following page. These three themes illustrate the possible results utilising the chunk display attributes check-buttons. Note that **Theme1** displays all chunk names as does **Theme2**. **Theme3** displays only the names of the top-level chunks.

Contrast this with existing literate document formatting: fixed output requires that code chunks are displayed with their chunk names at the top of the chunk implementation. Traditional literate program documents also lack the ability to associate a chunk name with the document chunk; a label for the document chunk is not supplied, and therefore not displayed. TBLP allows any variation (as described in Chapter 5) — name labels for *all* chunk types, or chunks displayed without their names, for example.

*The **xref** Button*

The **xref** button toggles the display of a chunk's cross-references in the final theme document. A chunk's cross-references appear as a list of hyperlinks (in the case of HTML documents) to the chunk's uses in all themes.

Setting the **xref** check-button to its on ("yes") state enables the output of a chunk's cross-references; to its uses in all theme documents loaded in the CBDE, resulting in the display of a chunk's cross-references in the final theme document.

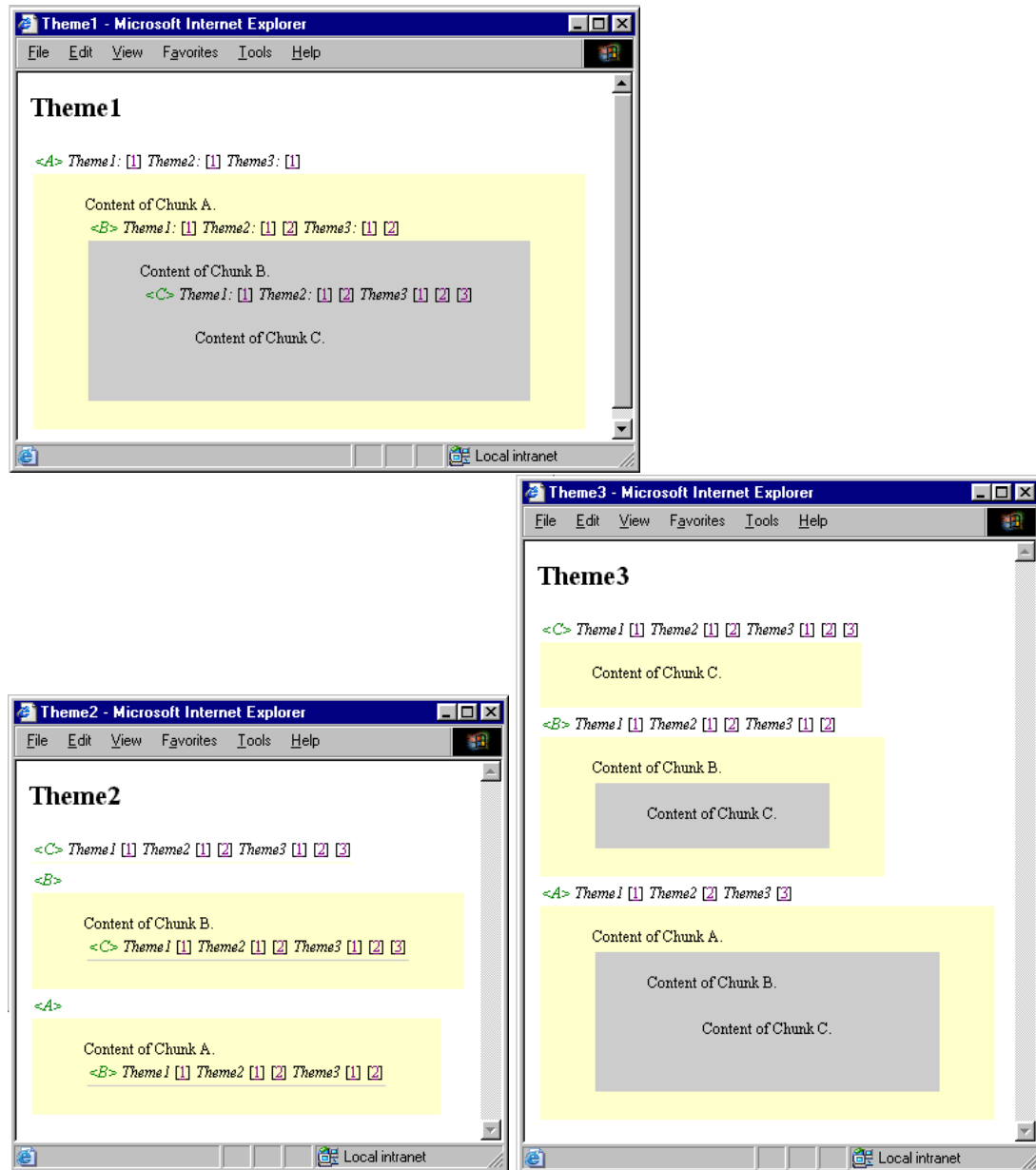


Figure 6.15: Three HTML theme documents illustrate the possibilities of representation by utilising the chunk display attributes. Note also the hierarchical (nested) representation of the chunks — these reflect the respective theme hierarchy.

This powerful feature facilitates cross-theme and intra-theme navigation — an important difference to traditional literate program cross-referencing. Traditional literate documents provide cross-references between chunk *definitions* and their *use*. In contrast, TBLP cross-referencing does not distinguish between a chunk’s definition and its use. This is because a chunk is implemented in the repository — although its content can be displayed in a theme document.

A chunk may appear multiple times within a theme, and in many different themes. Its appearance, or lack thereof, depends on the **show** and **content** attributes. Regardless of these attribute’s settings, each occurrence is cross-referenced if cross-referencing is set to “yes”. The example themes in Figure 6.15 clarify this concept. A cross-reference appears as the name of the theme that the chunk in which the chunk is referenced, followed by a numbered list of hyperlinks that link to the chunk’s specific reference in that theme. In **Theme1**, note the cross-reference details of chunks <<A>>, <>, and <<C>>; all chunks have **xref** set to “yes”. **Theme2** illustrates selective cross referencing — the two top-level chunks, <> and <<A>> do not display their cross-references. Note however that chunks have **xref** set to “no” may still be referenced. **Theme3** illustrates that only the top-level chunks have their **xref** attribute switched on.

The content Button

The **content** check-button toggles the display of the content of a chunk (free text and chunk references), effectively facilitating chunk holophrasting [81]. In this manner, chunk content can be folded in the theme view text widget.

Chunk holophrasting affects the output of a chunk in the literate document. This technique should *not* be used as a display minimiser in an effort to reduce the real estate that a chunk occupies in the text widget, which is a function of the tree-view widget.

Switched on, a chunk’s content is displayed. Switching **content** to off (“no”) will also turn off the **show**, **xref**, and **content** options of the chunk’s *descendants*.

It may aid theme development to decouple the display of a chunk's content from the display of its references. Thus, it would be possible to hide the display of a chunk's free text whilst allowing the display of its chunk references. The author is then able to develop a theme that effectively generates a table of contents.

Traditional LP code chunk references are effected in TBLP by setting **content** to “no” and setting **xref** and **show** to “yes”. This is illustrated by top-level chunks <> and <<A>> in **Theme2** in Figure 6.15. All themes illustrate the output of nested chunk content to varying degrees. Chunk <<A>> clearly illustrates the recursive display of chunk content. Nested chunk content is highlighted by different background colours.

It is interesting to contrast the display of **Theme1** and **Theme2**'s top-level <<A>> chunk. In both cases, the chunk's content is recursively displayed, however, **Theme1** illustrates that it is possible to also display each chunk's name and cross-reference information.

The static Button

Finally, the **static** check-button disallows a chunk reference from being updated to the default chunk version. Under normal circumstances, when a chunk is edited and then committed back to the repository, all occurrences of that chunk are updated to the new chunk version. This avoids the inconvenience of manually making the necessary updates. Any chunk that has **static** set to “yes”, however, will *not* be updated.

Take note that the static setting is global for every child parent chunk relation in every theme. For example, if chunk <> is nested inside chunk <<A>> and <>'s static attribute is set to “yes”, all occurrences of <> as a child of <<A>> are also set to “yes”. This avoids issues such as chunk content inconsistencies and hierarchical propagation of version updates.

Preventing a chunk's update is convenient when a theme specifically displays evolutionary or historical information about a chunk. It would be incorrect to update such a theme's chunks because it is the chunk versions themselves that are of specific interest.

The static value is set to “no” (off), by default.

Vertical Buttons

The two buttons (containing up and down arrows) located above the tree widget allow a top-level chunk to be moved before or after one of its sibling chunks.

6.3.3 The Theme-View Text Widget

One text widget exists for each theme. The theme view text widget is a more detailed low-level representation of the theme tree (see Section 6.3.2 on page 138). It reflects the unformatted content that will be displayed in the final literate document.

Note that the content of the final document is shown — not the content’s formatting instructions. Formatting instructions are contained in a separate XSLT document. It is likely that future work would facilitate the (dynamic) display of the final document.

The theme-view text widget is read-only; however, it is automatically updated to reflect chunk edits. Editing is discussed in Section 6.3.4 on page 151.

Theme/Chunk Display

Because themes are stored as XML documents (storage is discussed in Section 7.3), we have decided to also display them as XML documents. Missing, however, from the XML theme view document is the XML declaration “<?xml version=“1.0”?>” and the **theme** root element.

Chunk Attributes Each chunk reference is delimited by a **chunkref** element. The attributes and their values reflecting the chunk’s **name**, **version**, **chunkID**, and **variant** are displayed for each chunk. Note that these are the repository attributes of a chunk. The **show**, **xref**, **content**, and **static** variables (discussed in Section 6.3.2 on page 138) are also displayed, reflecting

the check-button options in the theme’s tree widget. For example, if the theme tree widget shows that chunk <> is referenced by chunk <<A>>, and chunk <> has show set to “yes”, xref set to “no” and content set to “no”, and the **variant** attribute is given a default value of null (no value), we would expect the following XML, or similar, to appear in the text widget:

```
<chunkref chunkName="A" chunkID="ch6" version="1.2" show="yes"
  xref="no" content="yes" variant="">
  <chunkref chunkName="B" chunkID="ch2" version="1" show="yes"
    xref="no" content="no" variant=""/>
</chunkref>
```

The theme text and theme tree widget representations of this example are presented in Figure 6.3.3 on the facing page.

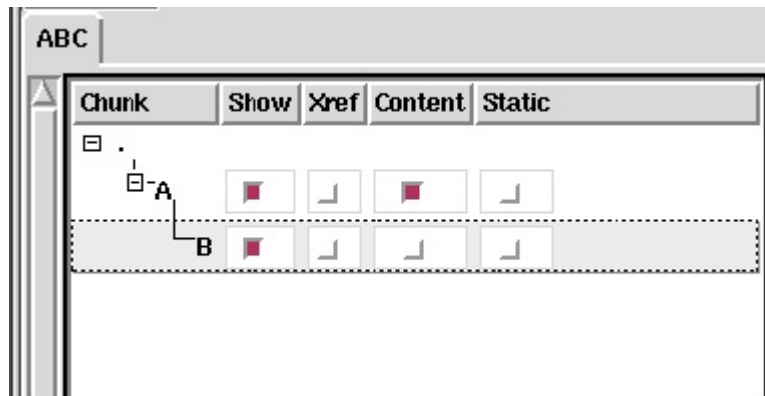
Free Text

XML markup is displayed in a light-grey font. The choice of colour is an effort to emphasise the textual content of a chunk, which is presented in standard black font. (Figure 6.17 on page 148 shows an example theme document presented in a theme-browser widget.)

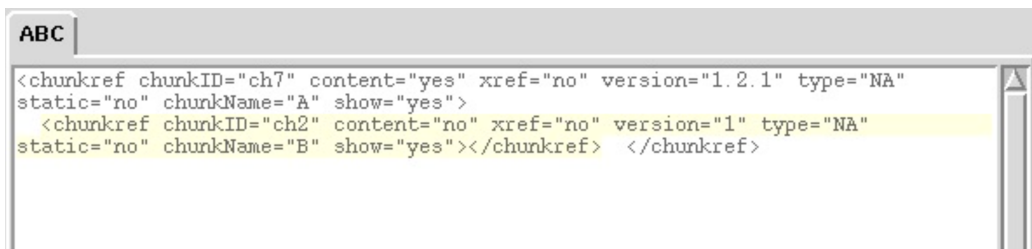
A chunk’s free text is displayed within a **<chunkref/>** element. Remember that the display of a chunk’s references is coupled with the display of its free text. Thus, if a chunk’s content is to be displayed, the textual content as well as the nested chunk references are also displayed¹⁶.

The author is then able to work through the chunk tree and determine the display details of each chunk.

¹⁶ similar to a traditional weave operation

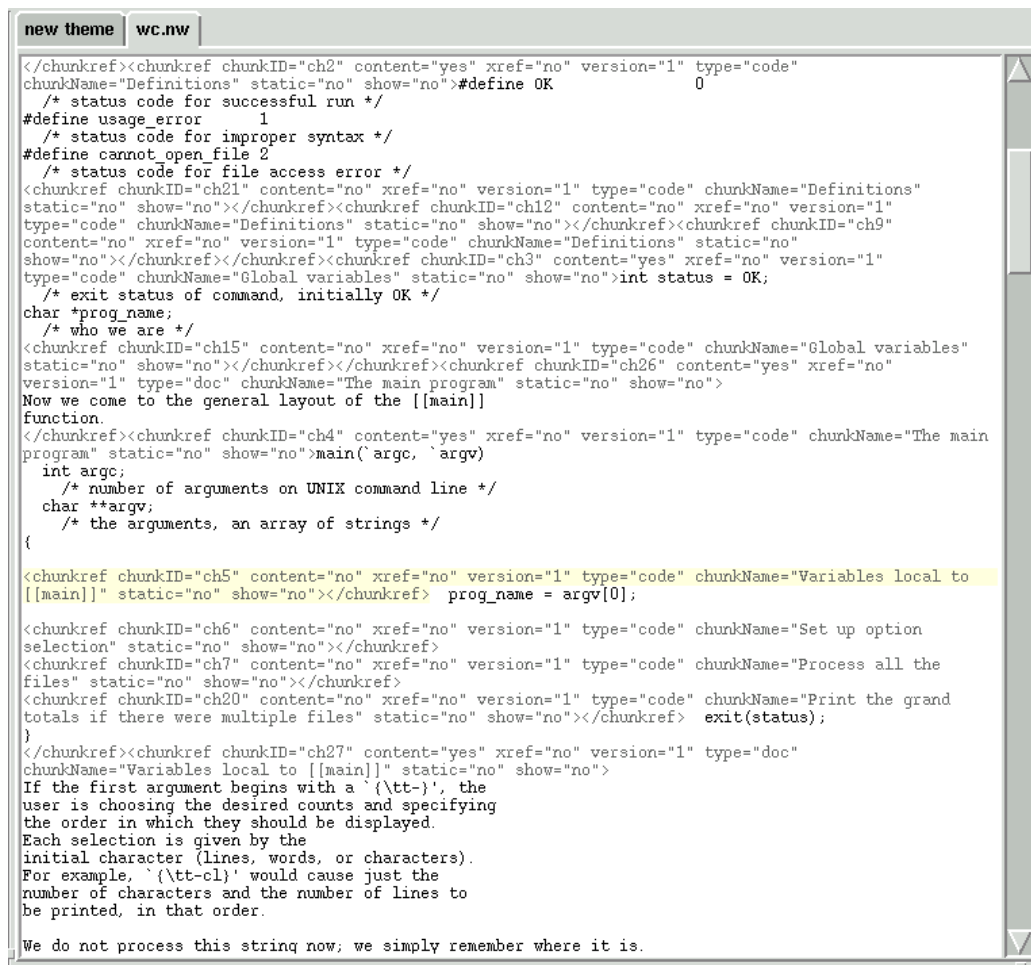


(a) A theme tree widget showing chunk A referencing chunk B. Both chunks have their theme attribute check-buttons set to either an 'on' or 'off' state.



(b) A theme text view widget reflecting the theme attribute settings in Figure 6.16(a).

Figure 6.16: The theme attributes of each chunk in the theme tree are reflected in the XML markup of that chunk in the text view widget.



```
</chunkref><chunkref chunkID="ch2" content="yes" xref="no" version="1" type="code"
chunkName="Definitions" static="no" show="no">#define OK          0
/* status code for successful run */
#define usage_error    1
/* status code for improper syntax */
#define cannot_open_file 2
/* status code for file access error */
<chunkref chunkID="ch21" content="no" xref="no" version="1" type="code" chunkName="Definitions"
static="no" show="no"></chunkref><chunkref chunkID="ch12" content="no" xref="no" version="1"
type="code" chunkName="Definitions" static="no" show="no"></chunkref><chunkref chunkID="ch9"
content="no" xref="no" version="1" type="code" chunkName="Definitions" static="no"
show="no"></chunkref></chunkref><chunkref chunkID="ch3" content="yes" xref="no" version="1"
type="code" chunkName="Global variables" static="no" show="no">int status = OK;
/* exit status of command, initially OK */
char *prog_name;
/* who we are */
<chunkref chunkID="ch15" content="no" xref="no" version="1" type="code" chunkName="Global variables"
static="no" show="no"></chunkref></chunkref><chunkref chunkID="ch26" content="yes" xref="no"
version="1" type="doc" chunkName="The main program" static="no" show="no">
Now we come to the general layout of the [[main]]
function.
</chunkref><chunkref chunkID="ch4" content="yes" xref="no" version="1" type="code" chunkName="The main
program" static="no" show="no">main( argc, argv)
int argc;
/* number of arguments on UNIX command line */
char **argv;
/* the arguments, an array of strings */
{
<chunkref chunkID="ch5" content="no" xref="no" version="1" type="code" chunkName="Variables local to
[[main]]" static="no" show="no"></chunkref> prog_name = argv[0];

<chunkref chunkID="ch6" content="no" xref="no" version="1" type="code" chunkName="Set up option
selection" static="no" show="no"></chunkref>
<chunkref chunkID="ch7" content="no" xref="no" version="1" type="code" chunkName="Process all the
files" static="no" show="no"></chunkref>
<chunkref chunkID="ch20" content="no" xref="no" version="1" type="code" chunkName="Print the grand
totals if there were multiple files" static="no" show="no"></chunkref> exit(status);
}
</chunkref><chunkref chunkID="ch27" content="yes" xref="no" version="1" type="doc"
chunkName="Variables local to [[main]]" static="no" show="no">
If the first argument begins with a '\tt-', the
user is choosing the desired counts and specifying
the order in which they should be displayed.
Each selection is given by the
initial character (lines, words, or characters).
For example, '\tt-cl' would cause just the
number of characters and the number of lines to
be printed, in that order.

We do not process this string now; we simply remember where it is.
```

Figure 6.17: A theme text view of Norman Ramsey's Noweb version of wc loaded. XML is displayed in light-grey.

Create a Chunk			Reference a Chunk			After Chunk Details		
Name:	B	ChunkID:	ch2	Version:	1			
Type:	NA	Include Chunk in Current Theme?: <input type="checkbox"/>						
Create New Chunk								

Figure 6.18: The Development Panel populated with chunk <<A>>’s details after clicking chunk <<A>> in either the repository widget, the theme tree, or the theme text widgets.

The display of a chunk reference’s attributes can clutter the theme text panel, thereby obscuring more important information and decreasing the readability of the theme document. XML documents are not suitable or easily-read representations of prose or program source code as. It is probable that future enhancements of the CBDE will present each theme as a more readable marked-up document that is adaptable by the user, both in granularity of detail — such that only user-specified attributes of a chunk are displayed, and in the formatting of content — such that chunk content is presented in a more readable font, colour scheme, and so on.

Chunk Selection

Clicking on a chunk in the text widget causes two actions to occur:

1. The tree widget locates, highlights, and (if necessary) automatically scrolls to the respective chunk.
2. The “Chunk Development” panel is populated with the chunk’s details (this is the same functionality as clicking a chunk in the theme tree-view — Section 77 explains).

The Hide Markup Button

The “Hide Markup” button (positioned on the button panel) allows the programmer to toggle the display of the XML chunk markup. Hiding the markup

aids the readability of a theme because it allows the author to concentrate solely on the textual content of each chunk.

When a chunk’s markup, and therefore its attribute details, are hidden, it is still possible to determine the chunk the content belongs to by clicking on the content of interest; the corresponding chunk in the tree widget is then highlighted. Specific chunk details are also displayed in the “Chunk Development Panel”.

Figure 6.9 on page 131 shows a theme text view with the chunk XML markup hidden. It also illustrates how a chunk (`<<The main program>>`, in this case) can be highlighted with a different coloured background. Note also the chunk details in the development panel. Figure 6.9 can be contrasted with Figure 6.17 on page 148. It is clear that Figure 6.17’s readability suffers due to the XML markup display.

Whether XML markup is displayed or not does not affect the output document in any way.

Theme Document Traversal

All output documents are themes. Source code documents are therefore ‘just another theme’. Consider the following scenario: after outputting theme “Ramsey’s wc source” to a file (`wc.c`) and compiling it, the compiler reports an error on line 121. Right-click “Ramsey’s wc source” text theme widget in the CBDE. Select the “View” and then the “Goto Line” options of the cascading raised menu that is raised. Enter “121” in the dialog box that appears. The text widget scrolls to line 121 of the “Ramsey’s wc source” theme text view widget.

Editing the related chunk(s) — `<<Scan file>>` (see Section 6.3.4 on the facing page), and re-compiling `wc.c` fixes the bug. Literate program debugging problem solved (Section 4.1.1 on page 61 discusses the debugging problem).

Figure 6.19 demonstrates that source code documents are themes in TBLP. It also demonstrates, importantly (to our non-literate friends), that programmers can edit these documents, and thus still use a conventional programming approach using TBLP. Of course, in a multi-user environment,

each user is free to use whatever approach suits best.

6.3.4 Editing

The editing text pane in the lower right hand side of the CBDE in Figure 6.9 on page 131 allows the editing of individual chunks. Unlike the theme text view widget, this is a writable, or editable **Text** widget.

The purpose of having a separate chunk editing widget distinct from the theme environment — the text and tree theme widgets — is due to the desire to ensure editing is largely a theme-unspecific, theme-context-free operation. By doing this, chunk editing is universal and occurs across all themes that include a chunk. Chunk editing is therefore all-contextually relevant. We refer to the editing approach employed by CBDE as repository-based chunk editing.

*repository-based
chunk editing*

Repository-based chunk editing is so termed because it transparently allows the author to adapt the repository. This method of editing encourages the separation of content and presentation — the author concentrates, when editing, on developing the content of the chunk. He is not concerned about how the chunk will appear within its referencing themes (although it must be logically and syntactically correct). Repository-based chunk editing also maps well to the model-view-controller pattern adopted by the CBDE, which is discussed in Section 6.4.

De-contextualised Chunk Editing

A different approach would be to allow in-theme editing, such that the author, though editing the repository, was given the impression that a specific theme contains chunk content and is being edited¹⁷. Thus, it would be possible to maintain repository-based chunk editing using an in-theme approach. In-theme chunk editing, however, *promotes* de-contextualised edits. Although we have no empirical evidence to support this claim, we believe that the author will not be as aware of the effect of the edits on other themes. Editing a chunk in one theme might render another theme invalid.

in-theme editing

¹⁷ (it is debatable whether this incorrect impression should be given.)

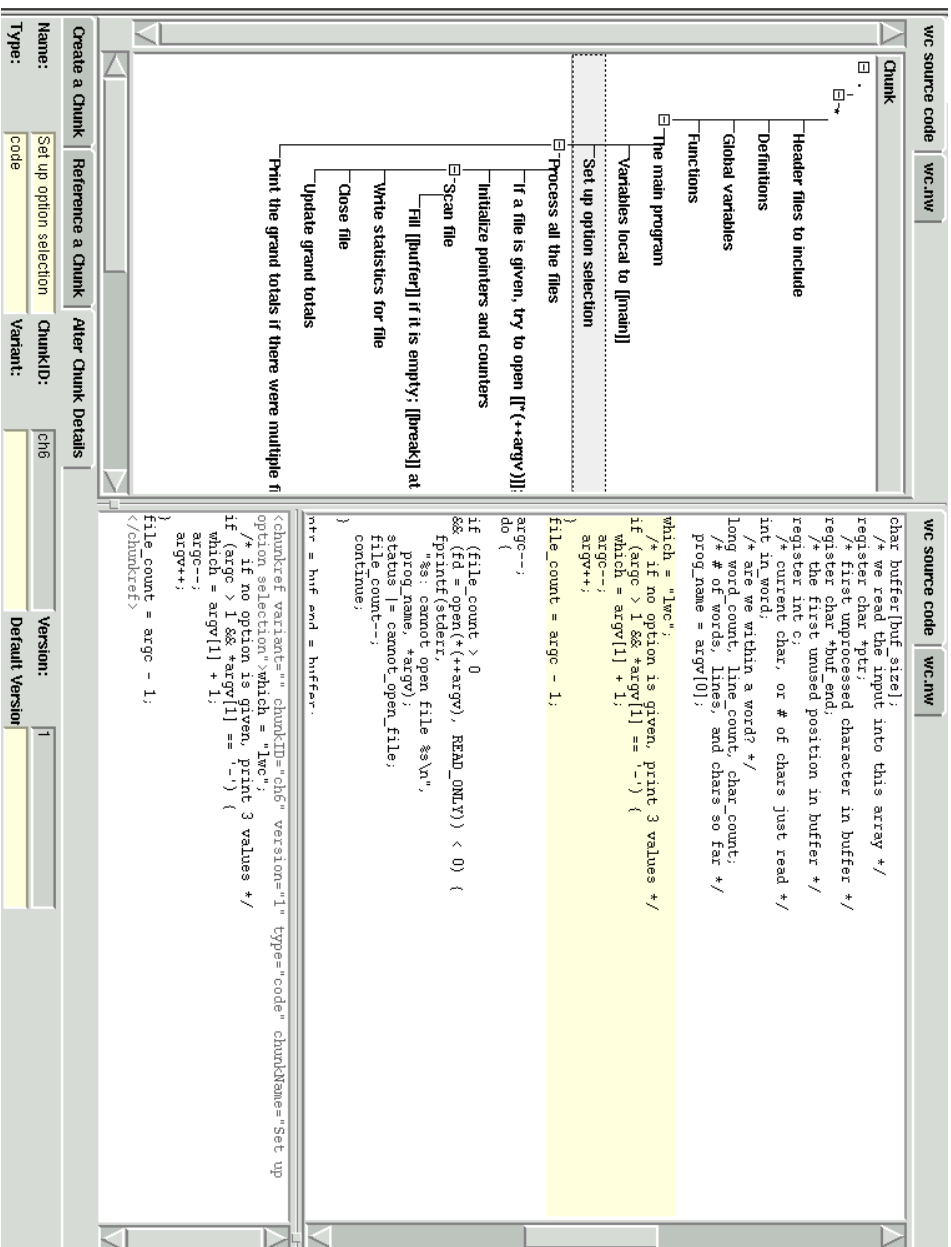


Figure 6.19: Theme wc source code shows that it is possible to view a literate program in the traditional compiler-oriented order. The chunk edit pane is ready to edit <<Scan file>> (ch16).

Although the chunk edit might be appropriate for the theme being edited, other themes referencing the edited chunk become semantically, and even syntactically, compromised.

As an example, consider the following: theme **a** contains the following badly scoped chunk:

```
<chunk chunkName="chunk1" type="c-code" version="1">
  for ( i=0; i<max; i++ ) {
    add( i, max );
  }
</chunk>
```

Note the lack of a closing parenthesis in the **for** loop. This chunk is included in theme **a**, which compiles and runs well.

Theme **b** also includes this chunk; however, its tangled source does not compile due to the missing closing parenthesis. Spotting the badly scoped chunk, the programmer, allowed to edit the chunk while working in theme **b**, inserts the closing parenthesis at the end of the chunk.

```
<chunk chunkName="ch1" type="c-code" version="1">
  for ( i=0; i<max; i++ ) {
    add( i, max );
  }
</chunk>
```

This in turn alters theme **a**'s chunk. Theme **b** now compiles, however theme **a** doesn't. There are many other similar scenarios where such de-contextualised edits could prove problematic. \TeX , \LaTeX , XML, and HTML are all examples of document type-setting languages where extra, or omitted delimiters or elements can cause unexpected errors.

Although the ultimate solution is to develop well-formed chunks from the onset, a currently feasible solution, given the programmer's dilemma, is one of the following:

1. Create a new chunk in theme **b** that adds the missing parenthesis.

```

<chunk chunkName="ch0" type="c-code" version="1">
  <chunkref chunkName="ch1" type="c-code" version="1">
    }
</chunk>

```

2. Alter theme a’s chunks so that they are properly scoped, thereby generating compilable code for theme a and theme b.

We have demonstrated the grammatical discrepancies that can arise with decontextualised chunk editing. Using a separate chunk editing pane also encourages the separation of content from presentation — something firmly employed by the XML paradigm. TBLP chunk editing focuses on two things: (1) the textual content of a chunk, and (2) the chunks that it may reference. This is distinct from the display of a chunk within a given theme; the alteration of display attributes of a chunk is theme-specific and therefore is not performed in the text edit widget.

Editing a Chunk

By selecting the chunk to be edited via the theme tree or theme text-view widgets, and then clicking the “Edit Chunk” button, the chunk — its containing XML element and attributes — and its content are displayed in the text edit widget. Both the chunk’s textual content and its referenced chunks are displayed.

The chunk displayed in the text edit pane is a *representation of the chunk as it exists in the repository*. Note that in Figure 6.19 on page 152, the chunk reference made by <<Scan source>> does not display theme-specific attributes (the xref, show, content, and static attributes and their values are not displayed). This is because theme-specific attributes do not exist in the repository.

Whilst a chunk is in ‘edit’ mode, content (chunk references and free-text) can be added, deleted, or moved.

Deleting a chunk reference:

1. Selecting the chunk to be deleted on the tree widget.
2. Clicking the “Remove Chunk Reference” button.

Adding a chunk reference:

1. Positioning the text-insertion cursor at the point where the chunk reference should be inserted.
2. Choosing the appropriate chunk via the “Chunk Development” panel or the repository widget and clicking the appropriate chunk inclusion button (see Section 6.3.1 on page 130 for details on including a chunk from the repository, and Section 6.3.5 on page 157 on including a chunk from the “Chunk Development” panel).

Note that the manual entry of XML markup can be a tedious, error prone, and time-consuming operation. The CBDE automatically generates `chunk` and `chunkref` XML elements¹⁸. In fact, the programmer does not enter any TBLP-based XML throughout the development of a theme document. A chunk’s free-text content (which could be DocBook XML markup, for example), must be manually input.

Altering, inserting, and deleting the textual content of a chunk: As supported by the Tk Text widget, right-clicking on the widget causes a menu with various editing and search options to appear. Cutting, pasting, and copying is supported through this menu, and via the common Microsoft Windows key bindings of Ctrl-C, Ctrl-X, and Ctrl-V for copy, cut, and paste operations, respectively. The pop-up menu also supports a line positioning functionality such that a particular line may be selected to appear to in the text widget (see Section 81 on page 150 for use of this function).

¹⁸ This helps to alleviate the 3-syntax problem discussed Section 4.1.2 on page 62.

Committing a Chunk

The edited chunk can be committed to the repository by pressing the “Commit Chunk” button. This action causes the repository to be updated and for the update to be reflected in the repository widget. Namely, committing a chunk causes the following:

- The newly committed chunk becomes the default version. The new version is a new chunk that receives an incremented version number as well as its own unique `chunkID` (Section 7.5.2 on page 198 discusses the version numbering system, Section 6.3.1 on page 130 discusses versioning within the context of the repository widget).
- All occurrences of the old version of the chunk, in all themes, are replaced with a reference to the new version, unless the old chunk reference’s static attribute is set to “Yes”.
- Chunk references with static attributes set to “Yes” are not updated (Section 79 on page 144 explains the **static** attribute.).

The edits to a chunk will be dynamically reflected in the theme view text widget after they are committed.

Note that all chunk content entered in the text edit widget is stored in a `DOM::Text` node. The escaping of common programming language symbols such as ‘<’, and ‘>’, represented by entity references such as ‘<’ and ‘>’ respectively, is performed automatically by the CBDE. This avoids distorted interpretation and laborious input issues associated with the testing of literate program input. It also served useful when importing existing **Noweb** programs, such that the chunk content did not need to be explicitly marked up with CDATA ‘<![CDATA[’ and ‘]]>’ delimiters.

The downside to this is that text nodes must be post-processed in order to extract XML elements. A future implementation of the CBDE would allow the author to automatically mark up CDATA sections and use a SAX parser to filter the XML chunk content.

6.3.5 The “Chunk Development” Panel

The chunk development panel is a tabbed **Notebook** of pages that facilitates chunk creation, attribute alteration, and chunk referencing. These three features are presented in the following sections.

Creating a Chunk

The development panel’s “Create a Chunk” tab facilitates the addition of new chunks to the repository. To do this, the **Name**, **Type**, and **Variant** entry¹⁹ boxes are completed by filling in the necessary values. The **chunkName**, **type**, and **variant** attributes, as illustrated by the DTD in Figure 6.3 on page 122, are not necessarily completed.

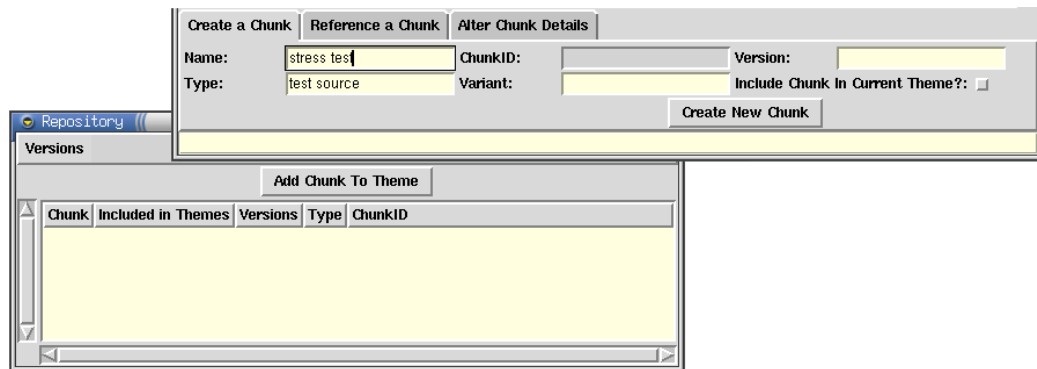
In order to create a code chunk named “stress test”, for example, enter “stress test” into the **Name** entry box and “test source” in the **Type** entry box²⁰; then click the “Create New Chunk” button. The chunk is given an automatically generated **chunkID** and chunk **version** number. If the **Version** text box is left empty, the chunk receives an initial version number of 1; however, any valid version number can be attributed to this new chunk. The **Variant** entry box, which allows the distinction between alternative implementations of a chunk, can also be completed. Figure 85 on the following page illustrates the results of this example. The result of adding a new chunk is reflected in the automatically updated repository.

A chunk’s name is displayed in the theme tree and also appears in the **chunkName** attribute as displayed in the theme text view, chunk edit, and repository widgets. All other attributes are also displayed in the theme text view, chunk edit, and repository widgets.

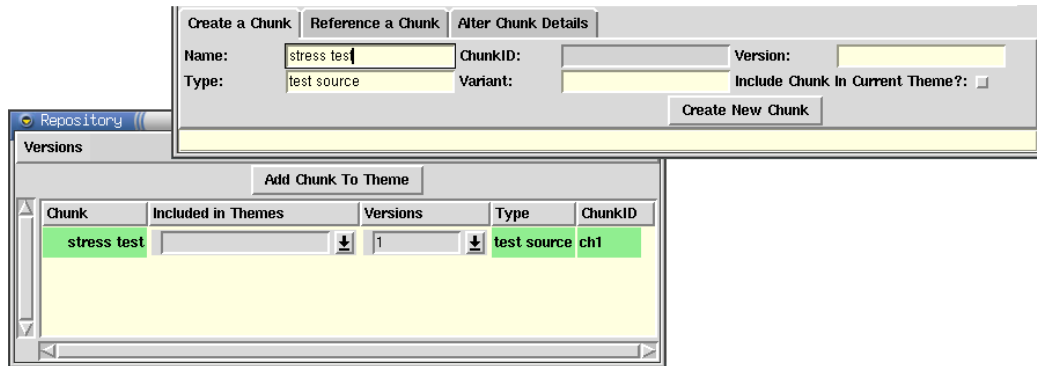
Selecting the “Include Chunk In Current Theme?” check-button allows newly created chunks to be included as chunk references in the currently raised theme document. Chunk referencing is explained in Section 85.

¹⁹ Entry is a Tk widget.

²⁰ Planning is required to create a list of descriptive and relevant chunk types. This task is imperative to the intuitive development of readable literate programs. The **type** attribute pertains to the type of chunk; conventional literate programming made the distinction between “code” and “documentation” chunks. TBLP enables the programmer to enter any type deemed pertinent to describe the nature of the chunk’s content.



(a) Complete the chunk's Name and Type Entry boxes appropriately.



(b) Click the “Create New Chunk” button, and the repository is automatically updated to reflect the addition of a new chunk.

Figure 6.20: The process of creating a new chunk: The appropriate chunk details are completed in the “Create a Chunk” pane of the Chunk Development Panel (Figure 6.20(a)). Figure 6.20(b) shows the result of creating a new chunk.

Create a Chunk Reference a Chunk Alter Chunk Details					
Name:	The main program	ChunkID:	ch26	Version:	1
Type:	documentation	Variant:			
Include Chunk Reference					

Figure 6.21: The “Reference a Chunk” tab in the “Chunk Development” panel. <<The main program>>, from Ramsey’s `wc.nw` is selected to be included by reference. The attribute values can be automatically completed as a result of clicking on the chosen chunk in the repository, theme tree, or theme text view widgets.

The theme tree, theme text view, and repository widgets are responsive to new chunks and are updated to reflect these changes.

Reference a Chunk

Referencing a chunk is performed via either the repository widget or the “Reference a Chunk” tab in the chunk development panel, as displayed in Figure 6.21. Referencing a chunk is mode dependent:

1. When in “Edit Chunk” mode, the referenced chunk is nested within the edited chunk.
2. When not in “Edit Chunk” mode — we’ll call this “Theme browse mode” — the referenced chunk is added to the currently raised theme as the last top-level chunk.

To reference a *set* of chunks, regular expression²¹ matches are made against chunk attribute values. It is possible to reference any number of chunks using this technique.

Some examples follow:

- If a chunk with `chunkID`, `ch9`, is to be included in a theme, the user simply enters “ch9” in the respective entry widget.

²¹ Perl-style regular expressions.

- If the author wants to reference chunks with **chunkIDs** ranging from **ch3** through to **ch5**, the regular expression “ch[5-9]” will return these chunks.
- A more practical example is to reference all versions of chunk <<a>> of type **code**. These chunks would perhaps be included in a theme that aims to show the evolutionary progression of a chunk’s development. “a” and “code” are entered in the chunk name and chunk type text entries respectively. All chunks that have a chunk name “a” and type “code” will then be included in the theme.

This is a powerful feature. All variations of regular expressions can be used to include any set of chunks. All matches are included as chunk references in either the theme or chunk being edited (depending on the mode, as already described in this section).

An improvement to this functionality is to return a list of chunks that match the input details, allowing the programmer to select certain chunks that are returned to be referenced. This differs to the existing functionality whereby all chunk matches returned are referenced.

Extending this feature is essentially creating a search mechanism for chunks. This feature could also be extended to search the content of a chunk. A combination of XML XPath language and such regular expression searching would provide a powerful search mechanism that would allow queries such as:

- Which chunks make use of the ‘isMatch’ variable?
- Which chunks reference chunk ‘x’?
- Which chunks does chunk ‘x’ reference?
- How many chunks does chunk ‘x’ reference?
- How many times is chunk ‘x’ referenced?

Alter Chunk Details

The “Alter Chunk Details” tab facilitates the change of a chunk’s **name**, **type**, **variant**, and **version** number. These changes are dynamically reflected in the browsing widgets (the repository, and theme tree and text view widgets).

For example, to rename chunk <<A>> to chunk <<X>>, the “Alter Chunk Details” tab is raised. Chunk <<A>> is selected (in one of the browsing widgets). This action populates the entry widgets of the “Alter Chunk Details” page with the details of chunk <<A>>. “A” can now be replaced with the value “X” in the **Name** entry box. Figures 6.22 on the next page and 6.23 on page 163 illustrate this operation in three distinct stages, with the results reflected in the automatically updated display widgets.

Note that the alteration of chunk’s name is localised to that chunk; its

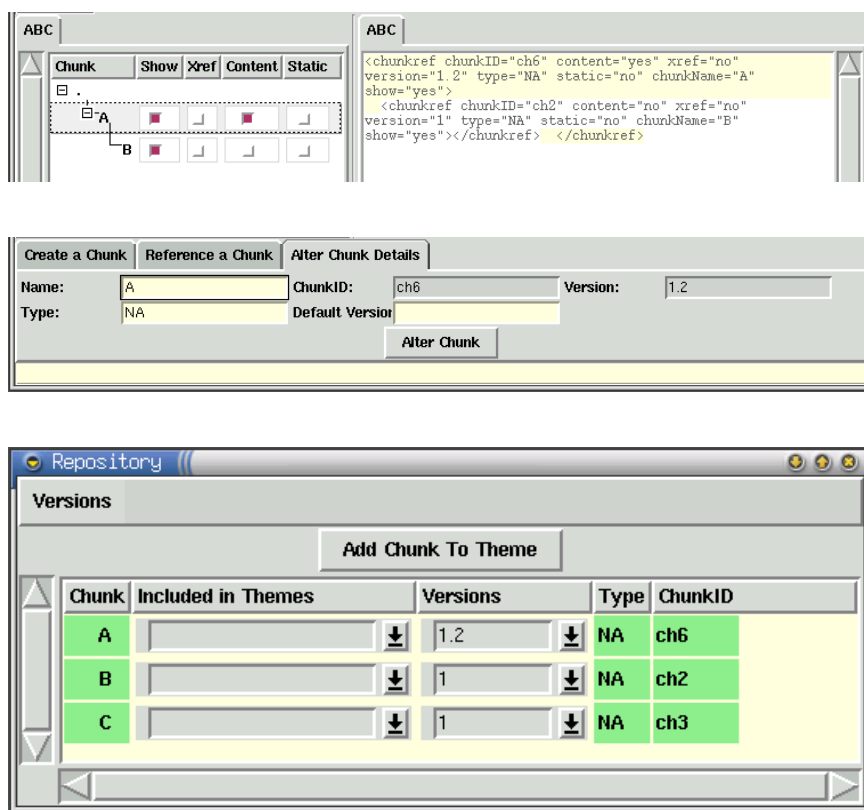


Figure 6.22: Chunk <<A>> is presented in the repository, and tree and text theme widgets.

related versions are unaffected, and thus, retain their same name.

The Default Version Entry The “Default Version” entry box on the “Alter Chunk Details” tab is used to nominate any chunk, by means of its version number, as the default version of the chunk version-tree to which it belongs. This action causes the nominated default chunk to be highlighted in the repository widget (Section 75 on page 133 discusses default chunk versions).

Enabling the selection of the default chunk is conceptually similar to RCS or CVS version control systems that allow version branching to occur: a current working document may be rolled back to a previous revision, and from there, work continues on the document. This operation creates a branch

Alter Chunk Details

Name: X ChunkID: ch6 Version: 1.2

Type: NA Default Version:

Alter Chunk

(a) Enter 'X' in the Name Entry box and the "Alter Chunk" button pressed.

ABC

Chunk Show Xref Content Static

ABC

```
<chunkref chunkID="ch6" content="yes" xref="no"
version="1.2" type="NA" static="no" chunkName="X"
show="yes">
<chunkref chunkID="ch2" content="no" xref="no"
version="1" type="NA" static="no" chunkName="B"
show="yes"></chunkref> </chunkref>
```

(b) After pressing the "Alter Details" button, 'X' appears in the repository...

Repository

Versions

Add Chunk To Theme

Chunk	Included in Themes	Versions	Type	ChunkID
X		1.2	NA	ch6
B		1	NA	ch2
C		1	NA	ch3

(c) ...tree and text theme widgets.

Figure 6.23: The chunk alterations are made and automatically reflected in the display widgets.

in the document version tree. The CBDE applies a similar version model, except that the granularity of version control is far finer — at the chunk level — and hence, far more powerful. Chunk-based version control is explained further in Section 7.5.

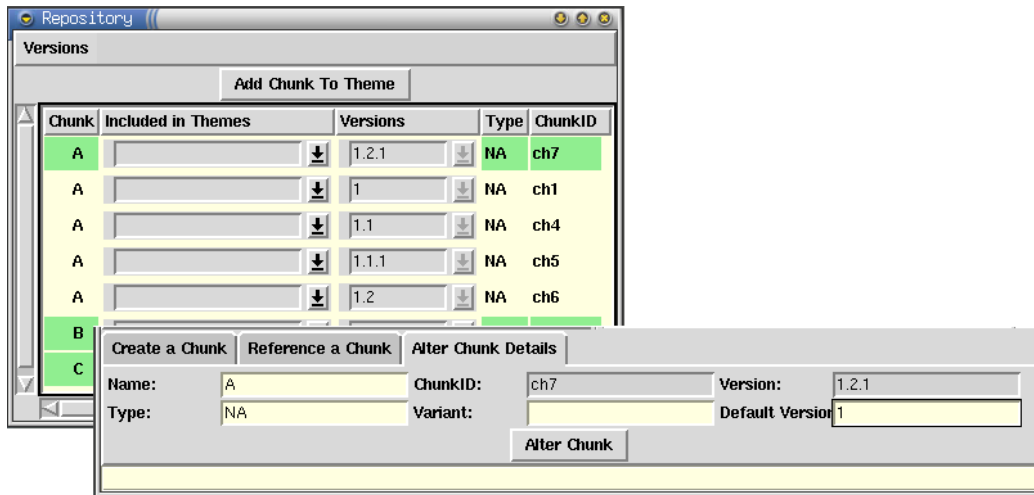
As an example, chunk `<<A>>` has been revised three times throughout development, thereby creating a version hierarchy of 1, 1.1, and 1.1.1. The current default version of chunk `<<A>>` is 1.1.1. It is determined, however, that version 1.1 is a better option for reasons of program efficiency. Version 1.1 is made the default chunk by:

1. Clicking on the existing chunk in the theme text (or theme tree). The chunk’s details are displayed in the raised Chunk Development panel.
2. Clicking the “Alter Chunk Details” tab to move this tab to the front.
3. “1.1” is entered in the Default Version Entry.
4. The Alter Chunk button is pressed.

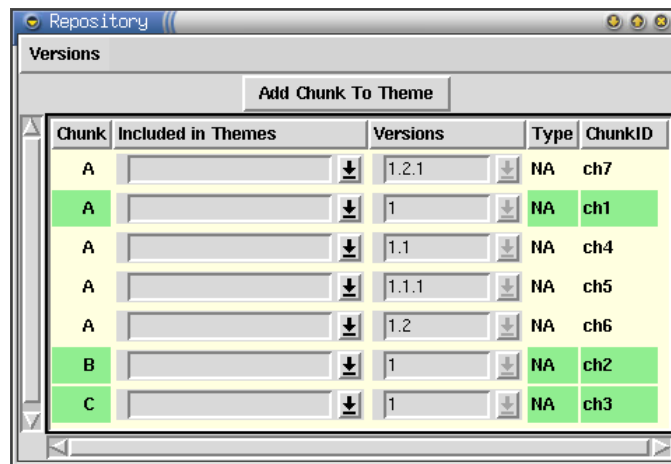
6.3.6 *The Variant Attribute*

TBLP provides a method to program, or demonstrate, by difference. Minor differences in the way we wish to treat elements drive the need to substitute one chunk for another. Examples are language translations of a chunk or source code chunks that implement an operation that differs for two operating systems.

Another example use for demonstration by difference is in the development of XSLT stylesheets. Using the CBDE, we are able to develop a variety of `xslt-source` chunks that are utilised to perform the same function, however, each producing different results. For example, cross-references between chunks in an HTML document may be displayed in a number of ways. The current method, as discussed in Section 7.2.2 on page 181 generates a list of hyperlink with a textual representation of the theme name the chunk is included in (illustrated in Figure 7.2 on page 184). The XSLT source is presented in Figure 6.25 on page 167 in `<<theme name xrefs>>`. Another



(a) Version 1.2 is the default of chunk <<A>>. <<A>>'s attribute details are entered in the "Alter Chunk Details" tab. The desired default version (1) is entered in the "Default Version" entry box.



(b) Version 1 is now the default version of <<A>>.

Figure 6.24: The process of altering the default chunk version.

method, contained in `<<numbered xrefs>>` is present a list of numbered cross-references and display the theme name when a mouse pointer is placed over the link.

These two chunk variants would commonly appear in the same theme document as immediate siblings. When an XML theme document is output by the CBDE (Section 7.1 on page 176 examines the XML theme document), the author stipulates (via the theme menu; see Section 6.3.7) the variant of chunks that should be output. Chunks that possess a different variant are not output (unless they contain the default null value).

6.3.7 Theme Functionality

The theme menu enables the creation of new themes. This results in a new set of theme tree and text view widgets appearing in the respective tabs. Themes may also be deleted, thereby destroying the respective theme widgets. A theme's name can be changed at any time. The "Remove Path from Theme" option removes a highlighted reference from a theme. The update is automatically reflected in the theme widgets. The "Stipulate Variant" option raises a dialog box that allows the author to stipulate the chunk variants that should be output in the XML source document.

Theme Document Processing

The "Run XSLT on Theme" option in the theme menu allows the author to choose a stylesheet to transform the currently raised theme with. The resulting file is output to a file that receives the theme's name and an extension of `.out`.

The Saxon ²² XSLT engine is utilised to process the theme XML document.

6.3.8 Loading and Saving — Repositories, Themes, and Projects

A project consists of a repository and any number of themes that are composed from the repository. Themes, like the repository, can be saved individ-

²² <http://users.iclway.co.uk/mhkay/saxon/>

```

<chunk chunkID="1" chunkName="generate xrefs"
  type="xslt-source" variant="" version="1">

  <xsl:template match="xref" mode="cross-reference">
    <chunkref chunkID="2"/>
    <chunkref chunkID="3"/>
  </xsl:template>

</chunk>

<chunk chunkID="2" chunkName="theme name xrefs"
  type="xslt-source" variant="expanded" version="1">

[<a href="{@theme}.html#{@target}" title="{@theme}"><xsl:value-of
  select="position()"/></a>]

</chunk>

<chunk chunkID="3" chunkName="numbered xrefs"
  type="xslt-source" variant="compact" version="1">

[<a href="{@theme}.html#{@target}" title="{@theme}"><xsl:value-of
  select="@theme"/></a>]

</chunk>

```

Figure 6.25:

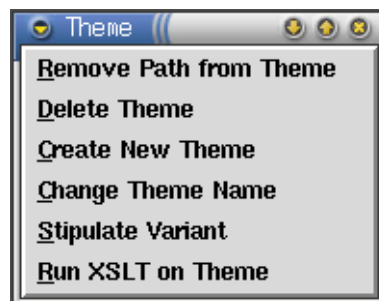


Figure 6.26: The CBDE Theme menu.

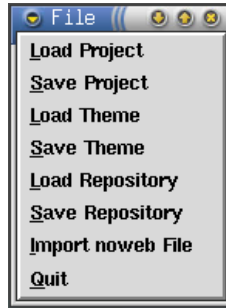


Figure 6.27: The CBDE File menu.

ually or together as a project. Saving a project consists of storing the chunk repository and all themes currently loaded (in-memory) in the CBDE.

Loading individual themes is possible, provided that the relevant chunk repository has been loaded previously. Loading a project loads the corresponding chunk repository, and the accompanying themes, into the CBDE system.

Loading and saving functionality is available in the File Menu, as shown in Figure 6.27. The details of persistent storage and retrieval are discussed further in Section 7.3 on page 189.

Importing Noweb Files

To maintain compatibility with existing tools, we developed a **Noweb** conversion class that converts **Noweb** source to TBLP CBDE readable source. The **Noweb** syntax is relatively simple and therefore easily filtered. Because TBLP does not facilitate additive code chunks (see Chapter 1, Section 1.4.1), all additive chunks are nested within a code chunk that is created specifically to contain and order these chunks.

6.4 Internal Architecture

The data storage, display of theme content, and editing of content are developed with respect to the Model View Controller (MVC) design pattern [29].

The CBDE is implemented utilising the MVC design pattern as follows (a

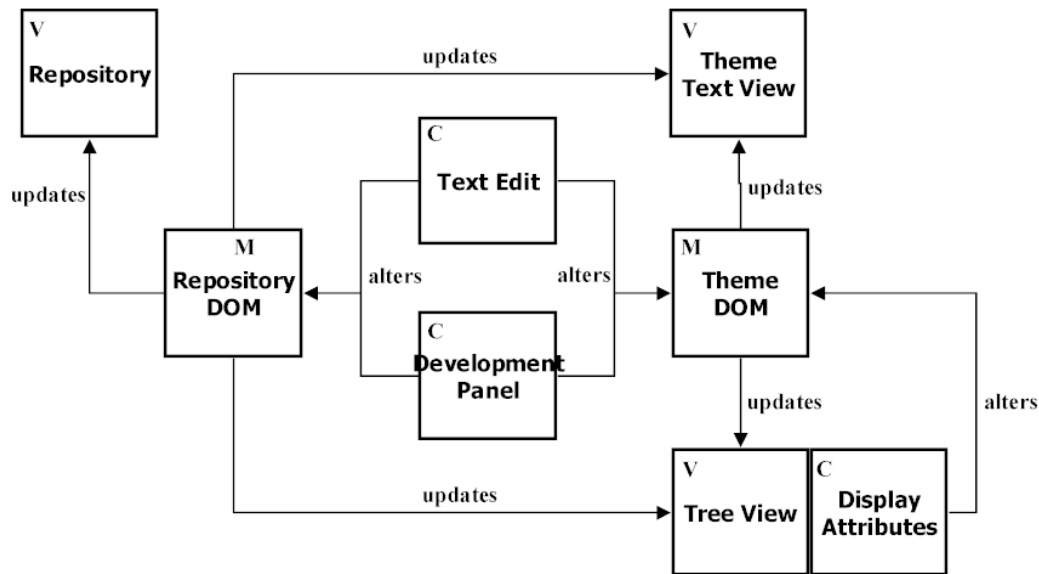


Figure 6.28: High-level view of the CBDE's system architecture illustrating the adoption of the Model (M) View (V) Controller (C) pattern.

high-level view is illustrated in Figure 6.28):

Model: A separate DOM is utilised to store each theme and repository (Section 7.3.1 on page 190 discusses the selection of DOM).

View: Tk classes that form the display widgets (repository, theme tree, and theme text view) reflect a view of the model.

Controller: The chunk text edit widget, the theme tree-view widget's chunk attribute buttons, and the chunk development panel facilitate the model's manipulation.

The MVC architecture facilitates the replacement of an architectural component — replacing the DOMs that implement the model with customised data structures, for example — without the need to alter the view and controller implementations.

Perl (5.6) is used as the implementation language and the Tk module utilised for interface development. The XML::LibXML (version 2.4.13) module is DOM2 compliant and is based upon the Gnome libxml2²³ library. It is utilised throughout to provide DOM support (the repository and each theme is stored in DOM structures) and XML validation and verification throughout. Node searching on the DOM structures is performed using the (extended) XML::LibXML::findnodes() method, which provides access to the XPath API in libxml2.

Several classes are utilised to manipulate the theme DOMs (control), and dynamically update the display widgets to reflect alterations (view). The classes that facilitate mapping and version relationships are:

- Theme
- Resolver
- PathUnits
- ClonedChunks
- AbsChunk

Specifically, these classes facilitate;

- the mapping of chunk references made by each tree-node and theme-view text widget to the relative position in the theme and repository DOMs. This is illustrated in Figure 6.29 on page 172. The theme tree view chunk attribute check-boxes are utilised to manipulate the display attributes of the respective chunk in the theme DOM. Edits of the repository are reflected in the structure of the theme tree.
- tree-based chunk version relationships. The tool facilitates chunk version management (the chunk model does not currently facilitate the definition of multiple hierarchical chunk relationships — see Section 9.1

²³<http://xmlsoft.org/>

on page 222 for further explanation) and maintains the association between chunk versions and their hierarchical state. This is illustrated in Figure 6.30 on page 173.

Each theme in a project is represented by a **Theme** object. This **Theme** object contains one **Resolver**. The **Resolver** object is used to resolve the mappings between the nodes in the GUI theme tree and text view widgets and their related chunks. These mappings are expressed in the form of a set of paths (**PathUnits**) — each unit of a path contains a reference to a chunk (**AbsChunk**).

For example, a tree widget contains a set of nested chunks; <<A>>, <>, and <<C>>. Figure 6.31 on page 173 illustrates the nesting arrangement in a theme tree view widget. As illustrated in Figure 6.32 on page 174, the **PathUnits** object maps paths to these three chunks: path .1 leads to chunk <<A>>, .1.2 to <>, and .1.3 to chunk <<C>>. A path therefore contains a series of units, each which reference a chunk: effectively unit .1 references <<A>>, unit .2 references <>, and unit .3 references <<C>>. Thus, we are able to determine stateful information about the nesting of chunks; not only that the path .1.3 leads to <<C>>, but also that <<A>> is a parent of <<C>>.

One **PathUnits** object is instantiated for each **Resolver**, and therefore, each **Theme**. The **PathUnits** object contains references to chunks, which exist as an instance of the **AbsChunk** class.

The creation of a chunk instantiates an **AbsChunk** object, which contains the following details:

- the chunk's name (**chunkName**).
- the chunk's **type**.
- the chunk's **version**.
- the chunk's **variant**.
- the chunk's unique identifier (**chunkID**).

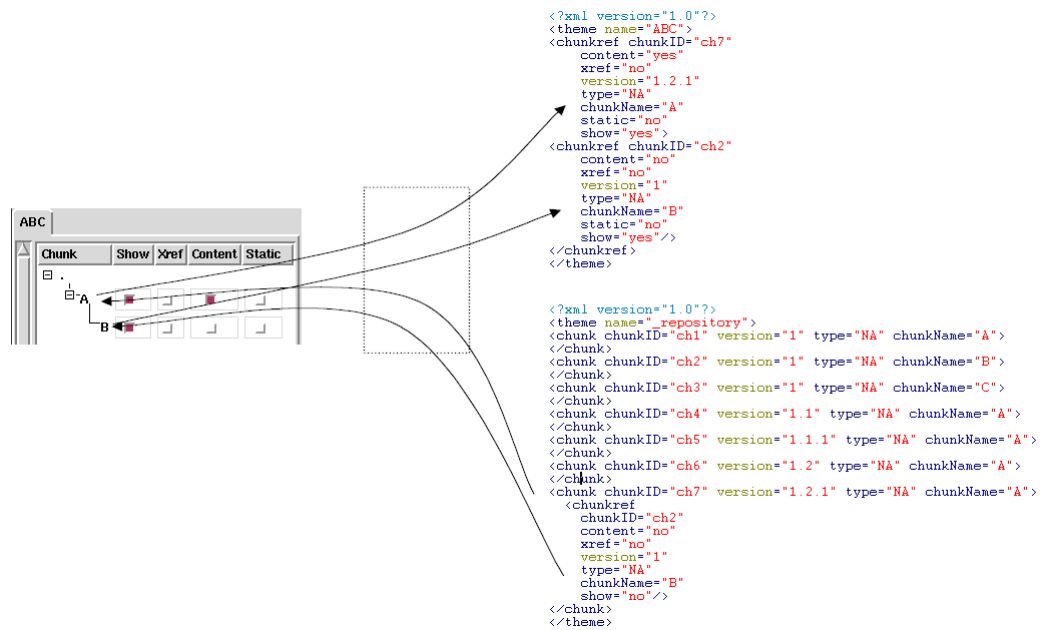


Figure 6.29: The CBDE (middle) maps theme tree (left) chunk references to the theme DOM (top right). The CBDE also reflects alterations to the repository DOM (bottom right) in the theme tree.

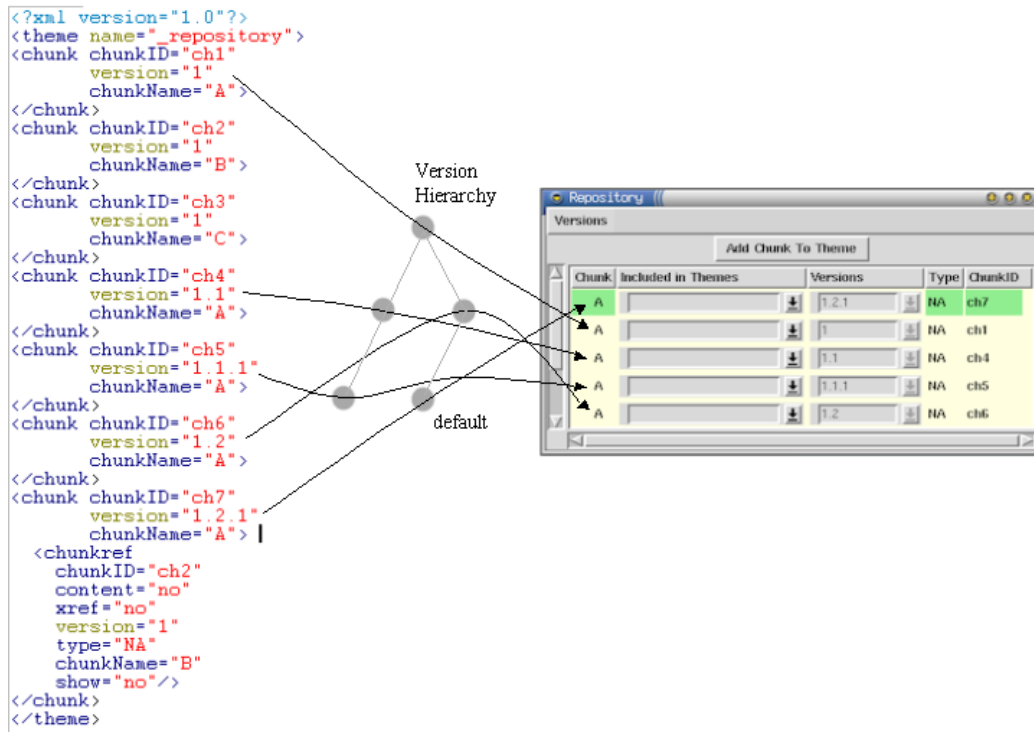


Figure 6.30: The CBDE maintains a chunk's version hierarchy (middle) and expresses it via the repository widget (right). The chunk model does not maintain this information, hence it is not expressed in the (edited) repository DOM (left).

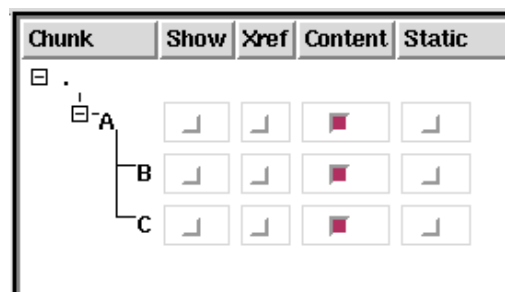


Figure 6.31: A series of nested chunks illustrating the hierarchical nature of paths.

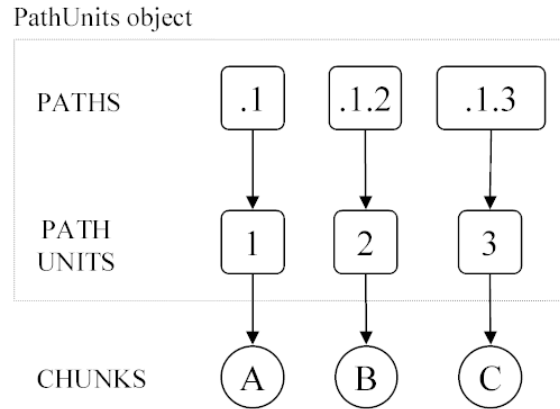


Figure 6.32: A populated PathUnits object contains two arrays. The PATH UNITS array uses the each path unit as an index value. A reference to a chunk is stored in each element of this array. The PATHS array contains the list of paths.

- the clone-set (version hierarchy) the chunk belongs to.

Its purpose is to abstractly define (hence **AbsChunk**) a chunk separate from the underlying repository data structure — although much of the information of a chunk is retrievable by the DOM (model) structure, we alleviate some of the workload of DOM queries by facilitating this light-weight data structure, thereby incurring a smaller time penalty than would be incurred querying the DOM — the price paid is the maintenance of each chunk’s attribute value updates.

The **AbsChunk** class is little else than getter and setter methods for the accessing and initialising/updating of a chunk’s attributes. The constructor (the **new** method) is notably important because it is responsible for adding a chunk to a collection of cloned chunks.

An explanation of the instantiation of an **AbsChunk** follows. If the chunk to be instantiated is an updated version of a previous chunk, it is *associated* with an *existing* **ChunkClones** object — the **ChunkClones** object maintains version tree relationships between chunks.

To clarify, each chunk is a distinct object. A set of chunks may be related through a version hierarchy. This relationship is stored in the form of a list

of `chunkID`'s. These `chunkID`'s are stored in a `ChunkClones` object.

Each `ClonedChunks` object holds an array of `chunkIDs` that each chunk references. This means that each `AbsChunk` object can query the `ClonedChunks` object that it references in order to determine fellow chunk versions.

Upon instantiation, if a chunk is a version of an existing chunk, it receives an incremented version number based upon the chunk it is based upon. This version numbering system is discussed in Section 95 on page 198.

6.5 Summary

Theme based literate programming is a document development framework that employs a pipeline based architecture. We initially discussed repository and theme composition as XML-based entities — the foundation of which is based on the findings of Chapter 5. This involved the development of respective DTDs, whose rules are reflected in the context based development environment (CBDE).

The prototypical CBDE implements and facilitates the authoring stage of the document development process. The CBDE provides a good platform for support tool development. Such tools are crucial to the success of TBLP because they enable the interaction between repository and theme content. The CBDE illustrates that it is possible to effect a contextualised TBLP environment, whereby chunk development is universal and affects all referencing themes.

The CBDE functionality reflects its architecture, which is based on the model view controller pattern.

In this chapter, we have introduced, as part of that architecture, the CBDE, an archetypical implementation of a TBLP authoring tool — a proof of concept.

Chapter VII

Document Output, Version Management, and Storage Concerns

In Chapter 6, we presented the theme-based literate programming document development framework. Authoring forms the initial stage of this framework and is implemented in the form of the Context-Based Development Environment (CBDE). The CBDE manages the development of multiple theme documents. The CBDE was demonstrated, and we also presented an architectural overview of its implementation. The CBDE generates literate theme documents ready for the transformation and formatting phases of the TBLP document development process. This chapter focuses on these later stages of the TBLP document development process.

XML theme documents, the product of the CBDE, are introduced in Section 7.1. XSLT Stylesheets transform a theme document into multiple theme-specific literate documents (Section 7.2.2).

In Section 7.3 we describe external and internal storage considerations of the CBDE's implementation. Finally, chunk versioning and chunk identification are discussed in Sections 7.4 and 7.5, respectively.

7.1 XML Theme Document

The XML theme document bears similarities to a traditional literate source document. It is the transformation of the repository and theme source documents — the chunk references from the theme source are resolved against the repository to produce the theme document. This transformation is performed by the CBDE authoring tool, and the resulting document is, according to our processing framework (see Figure 6.1 on page 118), processed and

transformed by any number of XSLT stylesheets.

Processing the theme document bears similarities to the tangle process of traditional LP, formatting instructions excluded. Its purpose is to represent, to the processing stylesheet, the *display* instructions and the *content* of each chunk. The content and display instructions are determined by the **show**, **xref** and **content** attributes that each chunk reference possesses (see Section 63).

An XML theme document representation of the source code theme example progressed in Chapter 6 is illustrated in Figure 7.1.

Points to note about the XML theme document are:

- The content of each chunk (free text and chunk references) is included (contrasting with the XML source document, which contains *only* chunk references).
- The nested hierarchy of chunks, as represented in the XML theme source document, is maintained.
- We have decided to use the chunk's type as the element name. This decision is made to ease post-processing by an XSLT stylesheet.
- The **static** chunk reference attribute, which is present in each chunk reference in the XML theme source document, bears no relevance to the presentation and formatting of the theme document. It is thus not included in the XML theme document.

In addition each chunk element can contain several **xref** elements (discussed further in Section 7.2.2 on page 181), which contain cross-referencing information about each occurrence of the chunk in every theme document. The method of automatically generating these details by the CBDE largely eases process of otherwise reading multiple XML theme documents to determine chunk use.

7.1.1 *An Condensed Processing Model*

One may reason that less development overhead is required by a stylesheet that processes the chunk repository and theme source document to produce

```

<c-code chunkName="define printIterator" version="1" chunkID="ch1"
  show="no" xref="no" content="yes" variant="">

void printIterator (int max, char * str) {

  <c-code chunkName="define and initialise variables" version="1"
    chunkID="ch2" show="no" xref="no" content="yes" variant=""
    anchor=".1">

    int i = 0;

  </c-code>
  <c-code chunkName="loop and print" version="1"
    chunkID="ch3" show="no" xref="no" content="yes" variant=""
    anchor=".1.2">

    for (i = 0; i < max; i++ ) {

      <c-code chunkName="output string" version="1"
        chunkID="ch4" show="no" xref="no" content="yes" variant=""
        anchor=".1.3">

        printf("%s\n", str);

      </c-code>

    }

  </c-code>

}

</c-code>

```

Figure 7.1: An XML theme document. Elements receive the name of the chunk type they represent; their attributes reflect the state of the matching chunkref element's attributes in the XML theme source document (Figure 6.5 on page 125).

the final theme document, thereby bypassing this stage in the development process. Adopting this approach, however, combines the processes of (1) content transformation and (2) theme document formatting. These are two distinct processes that are rightly separated. Our approach of separating these two processes not only treats content generation and content formatting distinctly, but also benefits the author by enabling the (development and) inclusion of intermediary processes throughout the TBLP pipeline processing model.

7.1.2 Theme Document Validity

The theme document's validity must be dynamically determined based upon a DTD derived from the repository. It is otherwise impossible to determine the set of chunk types that will compose the XML theme document's element names before the repository exists.

7.2 Theme Document Output: Formatting

Formatting is, ostensibly, the end of the TBLP processing model (as illustrated in Figure 6.1 on page 118). XML theme documents are processed by XSLT stylesheets to produce the final literate document. Before discussing stylesheet development, we consider why XSLT was chosen as a stylesheet language and what other possible technologies may be used.

7.2.1 Is XSLT the Only Option? Other Technologies

XSLT is XML's transformation stylesheet language¹. Although we utilise XSLT predominantly as a formatting translator, it may also be used throughout the TBLP processing model to transform one XML document, or chunk, into another. For example, chunks conforming to one DTD can be transformed to chunks that conform to another DTD.

Although we use XSLT to implement to the formatting stage of TBLP's processing model, it is not the only option available. Other viable options

¹It essentially forms the translation part of the XSL standard — its intended utilisation is for translation from one XML document to another.

are:

Cascading Style Sheets (CSS): Allow the formatting of documents (commonly used for HTML pages). They do not allow computing, such as element reordering (e.g., sorting), content-based computations (e.g., aggregation), addition of content to the document, or multiple document processing. CSS do, however, allow hierarchical and positionally-based contextual formatting.

Extensible Style Sheet Formatting Objects (XSLFO): A more sophisticated form of CSS (that uses XML syntax). XSLT and XSLFO form the two parts of the XSL whole. Whereas XSLT is predominantly used to transform, compute, sort, order, and add elements and content, XSLFO is designed to provide formatting instructions (often two-dimensional) to the resulting document.

Simple API for XML (SAX) event handlers: An event-based API. Programmers are able to add customised handlers to events that occur on a single pass through the XML document. Different handlers can be utilised when the opening of an element and the closing of an element are encountered, for example.

The TBLP processing model can incorporate any combination of these technologies. It is common practice to use CSS with XSLT, for example. Indeed, this practice provides a cleaner distinction between content management and formatting. For example, XSLT can be used to process the XML theme document and generate the required content (cross-reference details, chunk names, and chunk content), while CSS is used to format this content.

In light of these choices, we chose to use solely XSLT. Because of its rich language structure, it is possible to *easily* and *quickly* produce literate documents that accurately reflect intended representation. Moreover, XSLT documents can be used in a template/formatting² combination in the pipeline process.

² <http://www.xml.com/pub/a/2002/03/27/templatexslt.html> presents a good example of XSLT templates

SAX can also be incorporated as a processing option; however, we do not believe it is well suited to our needs of document transformation and formatting. SAX can be utilised appropriately, however, as an interface to CBDE's storage mechanism for the chunk repository, such that event handlers store and retrieve data from a database management system (DBMS).

7.2.2 *Stylesheet Development*

The XML theme document is processed by an appropriate XSLT stylesheet. The chunk-display options selected in the CBDE environment are ultimately reflected in the chunk's representation in the final theme document. The chunk-display options are `show`, `xref`, and `content` (described in Section 6.3.2 on page 138).

In order to develop the following literate document:

```
void printIterator (int max, char * str) {
    int i = 0;

    for (i = 0; i < max; i++ ) {
        printf("%s\n", str);
    }
}
```

the XSLT stylesheet must process `c-code` elements in Figure 7.1 on page 178, and output their content.

```
<xsl:template match="c-code">
    <xsl:apply-templates/>
</xsl:template>
```

In order to include other chunk types, such as documentation-style chunks (`cons-doc` chunks in this example) in the output source code³, a template can

³In traditional LP, this is equivalent to tangling documentation chunks — impossible with existing LP tools.

be developed to mark the chunk content up with language-specific comment syntax — in this case C’s ‘/*’ and ‘*/’ comment delimiters.

```
<xsl:template match="cons-doc">
/*
    <xsl:apply-templates/>
*/
</xsl:template>
```

Norman Ramsey’s `wc.nw` provides the basis for the illustration of how chunk display instructions can be managed by XSLT formatting instructions. We take `wc.nw` and import it into the CBDE. Two themes are developed:

1. `wc.nw`: the traditional literate document — traditional `documentation` chunks describe `code` chunks.
2. `wc source code`: the source (tangled) document, which consists only of `code` chunks.

We process both themes with a stylesheet that generates HTML output. The following operations are, specifically, important:

display a chunk’s name: When a chunk’s `show` attribute set to ‘yes’, the chunk’s name is displayed by invoking the “show-title” template with the chunk’s `name` as an argument.

```
<xsl:if test="@show='yes'">
    <xsl:call-template name="show-title">
        <xsl:with-param name="title" select="@chunkName"/>
    </xsl:call-template>
</xsl:if>
```

The `show-title` template italicises the chunk name, and sets the font style as appropriate with the “`chunkName-font-format`” attribute set. Figure 7.2 on page 184 illustrates a formatted excerpt of the `wc.nw` XML

theme document. Note the formatting of the <<The main program>> chunk's name in the HTML equivalent.

```
<xsl:template name="show-title">
  <xsl:param name="title"/>
  <font xsl:use-attribute-sets="chunkName-font-format">
    <xsl:element name="i">
      &lt;<xsl:value-of select="$title"/>&gt;
    </xsl:element>
  </font>
</xsl:template>
```

If a chunk's `show` attribute is not set to “yes”, this operation is bypassed.

display chunk cross-referencing: Theme documents can be inter-linked by cross-referencing chunk occurrences in each theme. Figure 7.2 illustrates the resulting cross-references of a the <<The main program>> chunk. <<The main program>> is cross-linked to its other occurrences in the theme document set: it appears twice in the `wc.nw` theme document and once in the `wc source code` theme document. Clicking on these hyperlinks will transport the reader to the chunk's occurrence in the respective theme document. The following XSLT template illustrates the production of HTML cross-references from an XML theme document.

```
<xsl:template match="xref" mode="cross-reference">
  [<a href="{@theme}.html#{@target}" title="{@theme}"><xsl:value-of
    select="position()"/></a>]
</xsl:template>
```

It is important to note that the `target` value is the chunk reference's path in the XML theme document, as generated by the CBDE. This provides a unique chunk value that no other chunk reference possesses

```

...
<doc variant="" chunkID="ch26" anchor=".17"
content="yes" xref="yes" version="1"
chunkName="The main program" static="no"
show="yes">
<xref target=".17" theme="wc.nw" name="The main program"
chunkID="ch26" href="/chunk[@chunkID='ch26']"/>

```

Now we come to the general layout of the `[[main]]` function.

```

</doc>
<code variant="" chunkID="ch4" anchor=".18"
content="yes" xref="yes" version="1"
chunkName="The main program" static="no" show="yes">
<xref target=".1.6" theme="wc source code"
name="The main program" chunkID="ch4"
href="/chunk[@chunkID='ch0']/chunk[@chunkID='ch4']"/>
<xref target=".18" theme="wc.nw"
name="The main program" chunkID="ch4"
href="/chunk[@chunkID='ch4']"/>
<xref target=".2.7" theme="wc.nw"
name="The main program" chunkID="ch4"
href="/chunk[@chunkID='ch0']/chunk[@chunkID='ch4']"/>

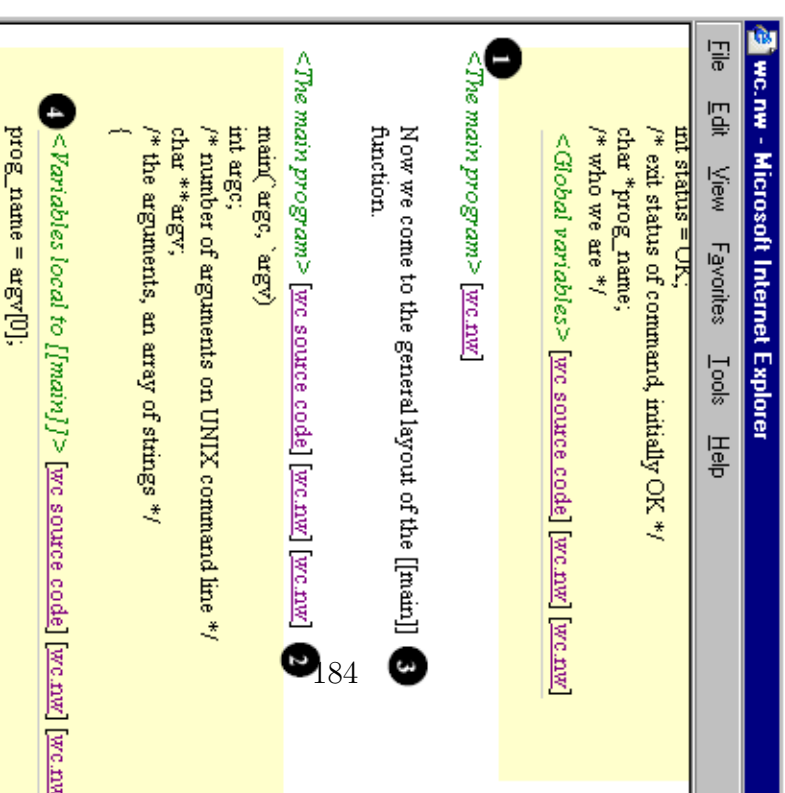
```

```

main('argc', 'argv')
int argc;
/* number of arguments on UNIX command line */
char **argv;
/* the arguments, an array of strings */
{
...

```

Figure 7.2: The XML theme document (left) is processed by an XSLT stylesheet to produce HTML (right). Note the markup of each chunk's (1) name, (2) cross-references, (3) content, and (4) nested chunks.



in a theme document. It is extracted from each `xref` element's `target` attribute in the XML theme document. The `target` value is output, and cross-references a chunk's `anchor` value. Each chunk receives an anchor element:

```
<a name="{@anchor}"></a>
```

Section 6.3.6 on page 164 illustrates how it is possible to alter the presentation of a cross-reference. It is quite possible that icons or symbols may be used instead. Future research will investigate more dynamic functionality by determining the worthiness of combining the advance of XML compliant browsers with this cross-referencing ability — folding theme documents utilising a holoprasting technique, for example. This research would also evaluate the use of XML technologies to dynamically reflect the status of the repository, thus avoiding static documents.

determine the chunk's type and nested position: A chunk can receive formatting depending on its parent's type. In the traditional literate model, this enables the formatting of `construction documentation` chunks up with programming language specific comment syntax should they occur as children of `code`-type chunks, for example.

The following stylesheet excerpt recursively processes chunks as they occur in the XML theme document. It produces a hierarchically formatted HTML source code document. It treats top-level `code` chunks differently to nested `code` chunks. The result of this is illustrated in Figure 7.3 on page 187. Note that nested `code` chunks appear as indented code fragments and are marked up with a different background colour to top-level chunks. This gives the reader visually suggestive information about a theme's composition and chunk hierarchy.

Figure 7.3 also illustrates audience-specific theme document formatting; the `wc source code` theme is formatted as an HTML document. We can make good use of XML's extensibility by processing this same XML

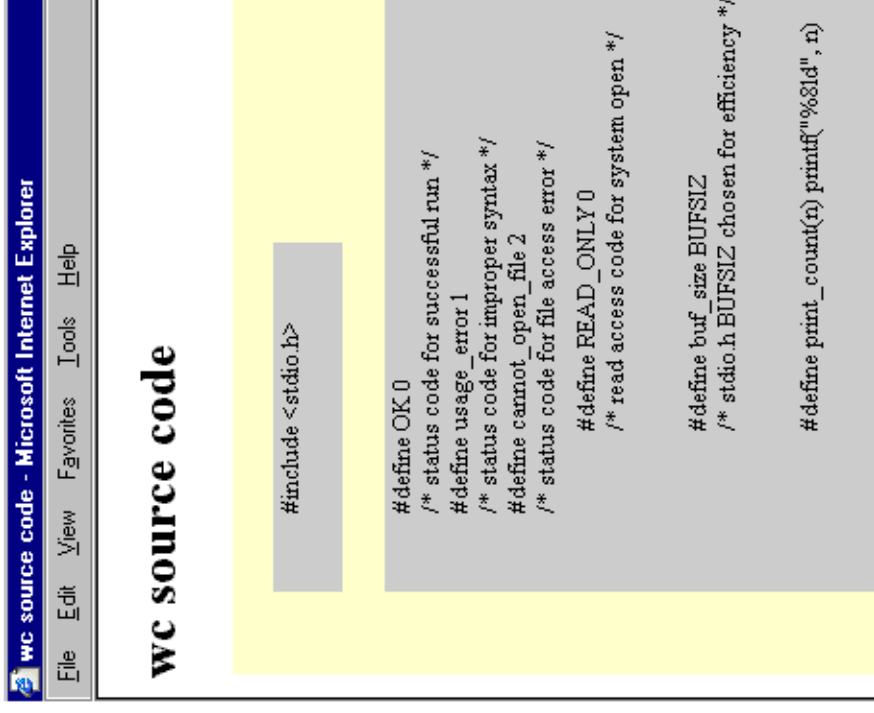
*audience-specific
formatting for
theme documents*

theme document with a different (and simpler) stylesheet to output the program source code ready for compilation. This powerful technique facilitates the reuse and multiple presentations that are obtainable from TBLP.

XML theme documents are unstructured — it is not possible to pre-determine in what order, and the nesting structure, elements will appear — and therefore suits a data driven transformation process.

```
<xsl:when test="name()='code'">
  <xsl:choose>
    <!--test for code/code-->
    <xsl:when test="parent::code">
      <td xsl:use-attribute-sets="nested-code-cell">
        <blockquote>
          <pre>
            <xsl:apply-templates/>

          </pre>
        </blockquote>
      </td>
    </xsl:when>
    <xsl:otherwise>
      <td xsl:use-attribute-sets="code-cell">
        <blockquote>
          <pre>
            <xsl:apply-templates/>
          </pre>
        </blockquote>
      </td>
    </xsl:otherwise>
  </xsl:choose>
</xsl:when>
```



```
<code variant="" chunkID="ch0" anchor=".1" content="yes"
xref="no" version="1" chunkName="*" static="no" show="no">

<code variant="" chunkID="ch1" anchor=".1.2" content="yes"
xref="no" version="1" chunkName="Header files to include"
static="no" show="no">
#include <stdio.h>
</code>

<code variant="" chunkID="ch2" anchor=".1.3" content="yes"
xref="no" version="1" chunkName="Definitions" static="no"
show="no">
#define OK 0
/* status code for successful run */
#define usage_error 1
/* status code for improper syntax */
#define cannot_open_file 2
/* status code for file access error */
...
```

Figure 7.3: An HTML version of the wc source code theme is produced. Chunk nesting is visualised by using differing coloured chunk backgrounds.

content: An iterative, or template-driven driven approach, is useful for chunks that contain common structures. For example, to generate an HTML theme document, the text nodes of each chunk need to be processed in an iterative manner in order to replace line breaks that occur in the CBDE environment with the HTML `
` (line break) element. This facilitates the display each chunk's content as it appears in the CBDE. If this method is not employed, chunk content appears as a singular paragraph with no line breaks. This is visually unappealing and an incorrect representation of the two-dimensional layout of a chunk's content. The following template illustrates a search and replace mechanism (borrowed from <http://mailman.real-time.com/pipermail/cocoon-users/2001-April/013695.html>)

```

<!-- template that does a search & replace -->
<xsl:template name="replace-text">
  <xsl:param name="text"/>
  <xsl:param name="replace" />
  <xsl:param name="by" />

  <xsl:choose>
    <xsl:when test="contains($text, $replace)">
      <xsl:value-of select="substring-before($text, $replace)"/>
      <xsl:value-of select="$by" disable-output-escaping="yes"/>
      <xsl:call-template name="replace-text">
        <xsl:with-param name="text"
          select="substring-after($text, $replace)"/>
        <xsl:with-param name="replace" select="$replace" />
        <xsl:with-param name="by" select="$by" />
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="$text"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

```

7.3 *Chunk and Theme Storage*

The CBDE relies upon an underlying data structure to support the chunk repository and supported themes. These data structures form the model part of the model view controller pattern adopted by the CBDE (see Section 6.4 on page 168). They must be stored both persistently — external to the application — and internally as stateful, immediately accessible structures. We investigate the storage options available for implementation.

7.3.1 *Storage Options*

With respect to the prototypical status of the CBDE, the Document Object Model (DOM) has been selected as a memory-resident storage medium. XML documents are selected as persistent storage for themes and repositories. The DOM, via its API, creates a memory-resident tree representation of the XML data — in the CBDE’s case, a repository and its linked themes. The API provides navigation and manipulation, programming language-neutral functions. DOM storage, however, is cumbersome and inefficient. Access and insertion times of the DOM data structure are relatively slow and memory requirements are greater than other storage methods.⁴

Internal DOM storage

Existing XML technologies, such as XPath[40], are executable on a DOM’s tree structure. DOM navigation and querying is greatly simplified, therefore.

The immediate functionality of the API makes DOM’s selection one of convenience; it enables the progression of thought and immediacy of implementation. This suits the CBDE’s current experimental status.

Summarising, DOM is the data structure of choice because:

- Functionality is already provided for tree building and manipulation. A tree representation maps well to theme data structures.
- There is a lower overhead in code development for model support, which allows a more explorative design approach.
- Applicability to other XML technologies, e.g., XPath.

The use of DOM, however, can be considered excessive for the needs of TBLP because:

⁴Innovations in DOM technology, however, have produced the Persistent DOM (PDOM) (see <http://www.darmstadt.gmd.de/oasys/projects/pdom/> and <http://www.infonyte.com/prod.pdom.html>). The PDOM stores XML in persistent binary, or textual format that can be read transparently by the application layer (the DOM API). These advances favor the choice for DOM as a storage option for our system. Although the DOM low-level behaviour might change, the code developed for this application can largely be re-used.

1. A chunk repository will never possess a nesting level greater than two — a chunk references another chunk. Thus, the repository DOM can be searched without intricate tree traversals.
2. Themes are essentially a list of lists (as described in Section 63 on page 109) and may be stored as such — rendering the bulky overhead accompanying DOM storage rather inefficient.

External XML Document Storage

Themes and their supporting repository are stored persistently as XML documents. XML documents have been chosen because they enhance the speed of development:

- The DOM API provides methods to easily read and write XML files, to and from the DOM tree, respectively.
- XML documents are human-legible. This facilitates debugging and random inspection.

Themes and repositories are not static objects, however, and are therefore not well suited to being persistently stored as XML documents, especially with respect to the future development of distributed, multi-repository systems. It is also unlikely that memory-resident structures such as DOMs will be useful unless state can be maintained between persistent and temporary storage. Other options of data storage that were considered are:

A flat file system: Themes and repositories are mapped to separate directories; chunks are stored in individual files. Although fast access times benefit this option, issues such as file-locking (to avoid dirty-reads, for example), file security (corruption/loss/accidental deletion/illegal access), and the upward scalability to distributed systems, all must be implemented and catered to, and thus, create a large overhead on design and development.

Database management systems (DBMS): Object oriented databases are likely candidates for future TBLP storage requirements (both persistent and dynamic) because they support possible future enhancements to the chunk model (such as explicit chunk relationship types).

In a multi-user TBLP environment, issues of data integrity will arise due to simultaneous read/write transactions — database management systems (both OO and relational) provide robust and tested solutions.

The size of repositories and themes is likely to be very large; containing these entire structures in working memory (as the DOM does) is not a realistic solution, therefore. Accessing these TBLP artefacts directly from a DBMS could be a more viable method. Indeed, integrated development environments, such as Jade⁵, provide an excellent opportunity for CBDE implementation.

XML enabled databases exist that are inherently able to read and store XML. This eases the mapping that would be required from XML to relational tables (and vice versa) at run time, for example.

Custom built data structures: Memory-resident structures can be either partially or fully serialised to alleviate working memory storage needs. They can also be used in combination with the external database or flat-file storage options. This technique is potentially the fastest of the options, however, can suffer from poor, and therefore, inefficient implementation.

Compatibility between current and future TBLP authoring implementations can also become an issue if data structures are tightly coupled to a fixed chunk model.

Efficiently implemented, custom data structures can support the development of TBLP tools that are custom-built for specific user types such as “thin client” use. Thin clients may be required to run on under-resourced systems or hosts with slow network connectivity, for example.

⁵ <http://www.jade.co.nz/>

SAX: In combination with the database and customised data structure solutions, SAX can be used to parse singular XML-based, database-resident chunks, to create a customised data structure.

SAX is commonly contrasted against the tree-based, memory-resident DOM alternative. SAX is an event based parser that invokes event-handlers upon XML-generated events; `start_element`, `end_element`, `start_cdata` are examples.

7.4 *The ID attribute*

Existing literate tools use a chunk's name as its unique identifier. Our chunk model differs by associating a unique `chunkID` (chunk identification) attribute to each chunk. The `chunkID` is essentially a chunk's primary key (in data modelling terminology). This overcomes several of the shortcomings of existing LP tools covered in Chapter 4, including:

Additive Chunks: Additive chunks are an ordered concatenation of chunks that possess the same name (see Section 1.4.1 on page 15). In existing LP tools, this ordering is derived by the chunk's lexical ordering in the source file (the web).

In TBLP, however, it is likely that chunks will possess the same name, but be different types. A `requirements analysis` chunk can be attributed the same name as the `requirements specification` chunk that it influences (Section 8.3.11 on page 214 makes recommendations on chunk naming strategies). These two chunks are not additive, however. A chunk's name is no longer an adequate unique identifier. Additionally, using a chunk's name as its key contravenes the DTD ID attribute type (the `chunkName` would be attributed an ID attribute) and therefore invalidates the XML document (see Figure 6.3).

Moreover, as discussed in Section 5.2.1, the chunk repository does not order chunks in any specific manner, thus rendering additive chunk composition by repository-based lexical ordering inappropriate.

Object Orientation: OO overloading and overriding, in TBLP, is captured using two or more **construction code** chunks possessing the same name. Existing LP tools do not facilitate such representation (see Section 4.2). The TBLP **chunkID** scheme disassociates chunk identification from chunk naming thereby allowing same-named chunks to represent different programming abstractions.

Scalability: The larger literate programs become, and the more reuse that is made of them (such that the author is able to include chunks from multiple repositories) and the more likelihood there is for a conflict between same-valued chunk attributes. It is quite possible that two chunks exist, each in a different repository, each with the same **chunkName** and **type** (take chunk `<<main>>` of type **code**, for example), however possess completely different implementations.

It may also be likely that two chunks from two different repositories contain the same **chunkID** value. Thus, the **chunkID** is not universally unique. Namespaces (an XML standard: (<http://www.w3.org/TR/1999/REC-xml-names-19990114/>)) can be used to realise such multiple repository implementations. This remains the work of future research.

The rules governing the repositories use of a **chunkID** are:

- Each and every chunk receives a **chunkID**.
- No two chunks may share the same **chunkID**.
- The generation of a **chunkID** is likely to be an automated process (as it is in the CBDE).
- A **chunkID** bears no semantic information. For example, a **chunkID** value less than another **chunkID** value is not an outright indicator that one chunk was developed before another. This encourages safer theme development practices.

7.4.1 *Multi-Valued Chunk Identifiers*

The utilisation of a distinct chunk identifier facilitates easier mapping to XML schemas⁶ and DTDs. The alternative option of using a chunk's combined attribute values as a multiple key does not map so cleanly. XML commonly utilises singleton attributes as element keys. Technologies such as XSLT, for example, allow the storage of single value keys in hash tables (via the `key()` function and `key` element) for efficient lookup and reference. Multiple keys would introduce unnecessary complexity to this approach. A multiple key representation such as:

```
<chunkref type="chunktype" name="chunkname"/>
```

is feasible, but inefficient. The matching of these attributes at run-time could place high processing demands on TBLP tools (such as the CBDE). It is likely, therefore, that a tool's implementation would generate unique keys associated with chunk objects regardless of the existence of multi-valued keys.

One possibility for further research is to investigate user-specified attributes — perhaps insisting on the `chunkID` and `version` attributes as a chunk's minimal attribute set. This approach would render a multi-valued chunk key impossible.

7.5 *Version Control — Evolution and Utilisation*

It is possible to develop a chunk model without chunk-level version control. If this is the case, a chunk is implemented, and all themes referencing a chunk will necessarily reflect the status of the chunk as it exists in the repository.

This scenario is inadequate, however, because of three key points:

Inadvertant theme corruption: Discussed in Section 6.3.4, de-contextualised edits may inadvertently render a theme invalid, and therefore incorrect.

⁶ we have concentrated on representing document structure using DTDs, however, it is likely that future work will introduce schema support (<http://www.w3.org/XML/Schema>).

Lack of scalability: Future TBLP environments will support theme and chunk composition from multiple distributed repositories. It is unlikely that chunk edits from distributed applications are relevant to all other themes referencing the same chunk. This issue emphasises de-contextualised edits; on a larger scale, however.

Disregard for the nature of chunk development: All previous states of a chunk, as it is edited throughout its life-span, are disregarded. This important information can be vital to a software system’s development — entire themes can be devoted to represent such information (as explained in Section 7.5.1. A CVS, RCS, or similar file-based version control system, not only does not support such historical theme development, but also do not solve the de-contextualised edit problem.

7.5.1 Themes of Versions

A set of chunk versions can be referenced and ordered to compose a theme. By doing so, a theme illustrates a historical perspective of a chunk’s edits, and provides the ability to document the reasons behind the alterations. Figure 7.4 on the next page, for example, expresses a version history of a set of chunks. This is a theme developed from Ramsey’s `setspace.stywc.nw`. The revision documentation chunks in this literate program are of type **revision-history**. Note that the **version** number of the chunk is output alongside the code chunk’s name; this is achieved by adding a template to the stylesheet. Note also that cross-reference information is displayed alongside **version 1.1** and not **version 1**. This is because all themes have been automatically updated to reference the latest version (1.1).

File-based version control systems cannot achieve such granular version control. Moreover, our TBLP model allows higher-order chunks — effectively documentation of the documentation — such that it is possible to include these chunks in a requirements analysis theme, for example, which discusses the need to add this extra functionality as presented in the theme document in Figure 7.4 on the facing page.

Interestingly, one could imagine maintaining a chunk versioning system

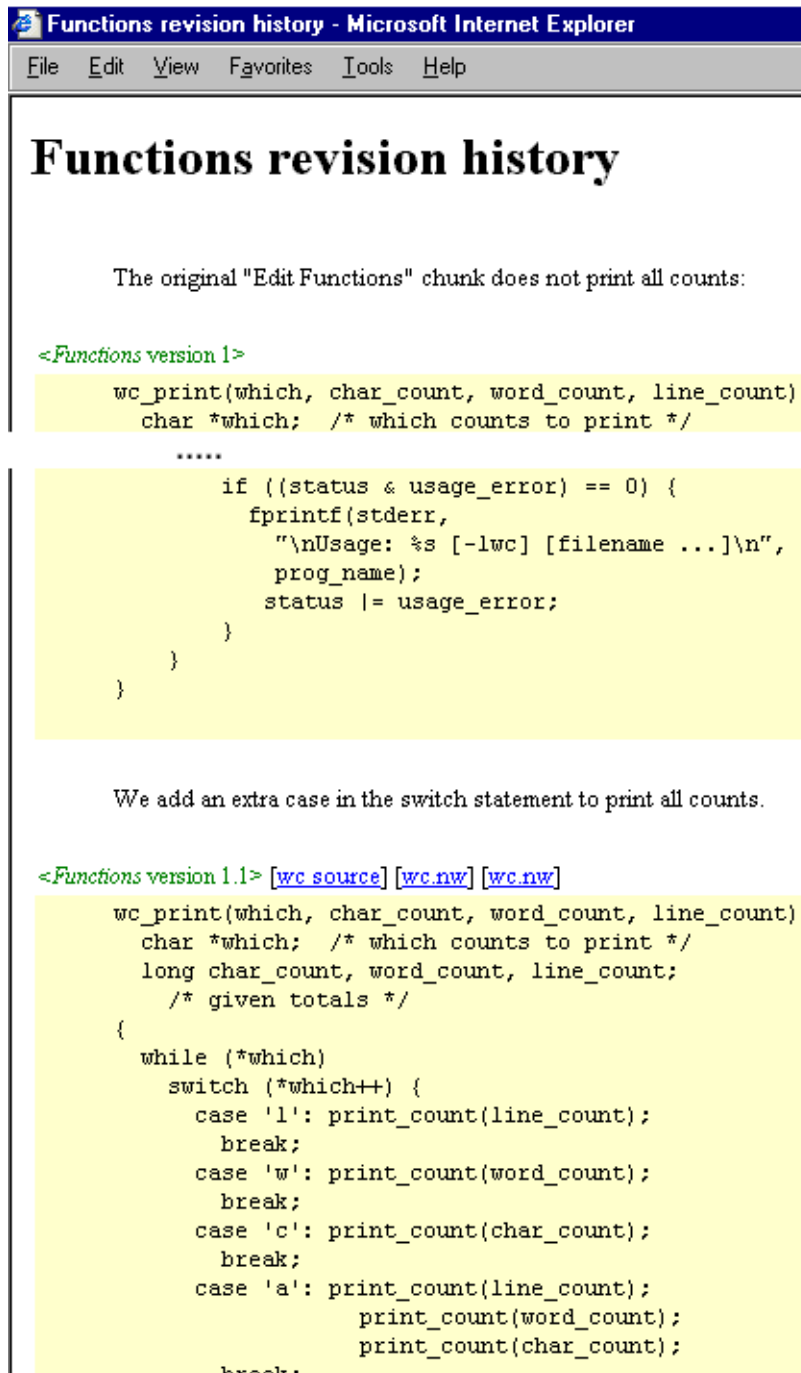


Figure 7.4: The version history of a chunk can be documented and expressed as a theme.

that stores all theme source in one chunk, effectively reimplementing a file-based version control system — this is not recommended practice.

7.5.2 *Branched Hierarchies of Chunks — Chunk Version Control*

Effectively, chunk version control is implemented by copying the new chunk's parent's attributes, i.e., **chunkName**, **version**, **type**, and **variant** and deriving a new **version** number and **chunkID**.

Given the new chunk's parent and the sibling order it finds itself, it is attributed an appropriate version number⁷.

Version Numbering

The chunk **version** numbering system utilised by the CBDE is illustrated in Figure 7.5 on the next page. It shows a hierarchy of chunk versions extracted from a chunk repository:

- The initial chunk is numbered version 1. This chunk is then cloned.
- The cloned chunk receives its parent's version number plus the number of the sibling order it represents i.e., 1.1.
- The programmer wishes to revert back to version 1 of the chunk. An edit is committed and the newly created chunk is numbered version 1.2 — its parent's version number is 1, and the new chunk's sibling order is 2.
- Editing and then committing version 1.2 generates version 1.2.1.
- The author then makes three edits on version 1.2.1 that are committed to the repository. Each one of these sibling chunks receives an incremented sibling number; 1.2.1.1, 1.2.1.2, and 1.2.1.3.

⁷ An appropriate version number is one which is able to be generated in a systematic and automatic manner.

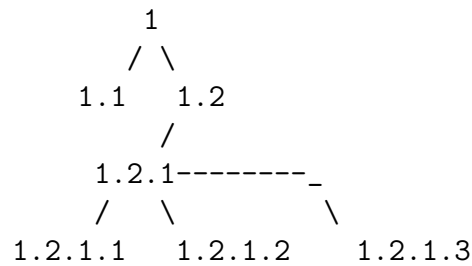


Figure 7.5: An example of a version hierarchy for a given chunk.

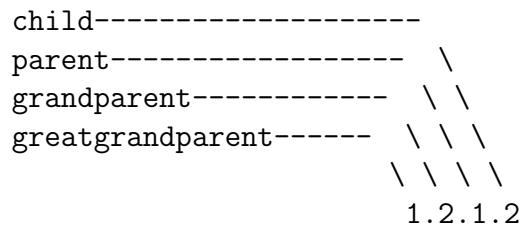


Figure 7.6: The hierarchical makeup of chunk version ‘1.2.1.2’.

Essentially, the version of a chunk is a concatenation of the chunk’s parent’s version number, and its sibling order.

Dissecting a given version number of a chunk reveals something about the chunk’s past. For example, version 1.2.1.2 tells us that this chunk has three levels of ancestor — a parent (1.2.1), a grandparent (1.2), and a greatgrandparent (1). It also tells us that it is the second child of its parent. Figure 7.6 shows the breakdown of the hierarchical numbers.

7.5.3 Theme-Based Version Control

Theme-based version control is not enabled in the CBDE. Thus, it is currently impossible to commit document rollbacks. Appendix E on page 353 discusses how this can be implemented as a simple extension to the CBDE and the theme model.

7.6 *Summary*

In this chapter, we discussed and illustrated example implementations of the transformation and formatting stages of the TBLP document processing model. Specifically, we showed how an XML theme document, as produced by the CBDE authoring tool, can be transformed by multiple XSLT stylesheets to produce, likewise, multiple formatted literate documents.

The second part of this chapter focused on TBLP data storage. We concluded that DOM facilitates exploration and speed of development. This suits the prototypical status of the CBDE. Future developments of TBLP tools are likely to utilise other technologies, however, such as a combination of DBMS, SAX processing, and customised data structures. These provide advantages of processing speed and efficient use of persistent and immediate memory.

We then discussed the benefits of using the `chunkID` attribute as a chunk's unique identifier. Important benefits are that it provides a solution to object orientated language representation (a limitation of existing LP tools), and provides a scalable solution to future TBLP developments, by allowing multiple chunks to possess the same attribute values, yet possess a mutually exclusive relationship. Finally, chunk versioning and version control was discussed. This is a powerful technique unique to TBLP, which enables the construction of historically-based theme documents that are comprised of chunk version hierarchies.

The next chapter examines the future considerations of TBLP.

Chapter VIII

An Approach to the Practice of Theme-Based Literate Programming

But I *know* what English words mean. I *speak* English. You must be a bit of a thicko. — *BlackAdder, Ink and Incapability*.

In this chapter, we present some basic guidelines that we believe will lead to the design of good literate programs. The ability to use TBLP effectively corresponds to good quality software. It is *not* our intent, however, to definitively develop a set of rules that must be strictly adhered to. The guidelines we propose are not set in stone and are presented as a tentative set of recommendations for the standardisation of a theme-based literate programming approach to software development. It is imperative that further research is conducted on the TBLP development — with this in mind, we set the ‘ball rolling’.

It is our intent to release these guidelines on a discussion forum to elicit and learn from the responses of the intending TBLP user-public. Note that many of the principles in this chapter can also be utilised for the development of traditional literate programs. Also, many of the examples used throughout this chapter are focused on source code representation; however, the concepts should be extrapolated over all chunk types.

A priority in TBLP, in addition to our recommended guidelines, is to maintain, Kernighan and Pikes [42] principles:

Consistency: follow a similar pattern throughout the literate program. Use similar documentation techniques for common items and common themes. Use similar chunking techniques for common abstractions.

Simplicity: Break complex ideas into simpler ones. Pity the audience of your efforts — after all, it may be yourself some time in the future.

Clarity: Simplicity promotes clarity. TBLP offers the opportunity to document and represent ideas to different audiences; use this opportunity to promote comprehension.

As implementation languages, C will be used for imperative languages, and Java for OO languages. Other languages are used for examples befitting their use. **Noweb** is utilised as the literate programming tool unless otherwise stated. **Noweb** has been chosen due to its clean and easy-to-read literate program presentation.

8.1 *Underlying Aims*

TBLP development encompasses three major processes:

1. The definition of chunks. Determining the scope of a chunk, and when to convert a piece of code into a chunk.
2. The procedural layout of these chunks in a logical order which aids program comprehensibility.
3. Documenting these chunks to provide meaning to intending readers.

8.2 *TBLP in Software Engineering*

TBLP is not a replacement for the SE principles of analysis and design. It is, however, an effective tool to be used to augment these principles. Moreover, we claim that TBLP is methodology-unspecific and can thus be utilised to meld methodologies together.

For example, a company situation may see two teams working on a large project. One team uses XP principles and the other uses a more formalised process, adhering more strictly to the phases of the SDLC. TBLP can be used to represent both results of development, and furthermore, can be used to *combine* aspects of both methodologies.

TBLP can be used to add structure when there seemingly doesn't exist any. Kernighan and Pike [42] mention that although formal engineering methods are suggested, and maintained, in many instances they are only partially used, or not used at all. While UML, as an example, provides an excellent foundation for formalised system modelling, there exist systems that evolve over time; whose future content or exact functionality is unknown at the beginning of the development process. Or perhaps, whose problem set is known, but whose exact implementation cannot yet be derived due to issues such as performance. Or perhaps, the project team simply does not use, or deem it necessary to use such formal methods. The points raised are all arguable points — to religious heights: TBLP provides an escape route by facilitating iterative, or evolutionary development, yet also provides the ability to represent these systems using multiple perspectives (themes) and thus, add structure.

8.3 Guidelines for the Good

Following, are a list of guidelines for TBLP development.

8.3.1 Target the Intended Audience

Code chunks can be construed as easiest to write because the audience is one¹ — the computer; other chunk types, however, have multiple audiences. Other chunks, however, may have multiple audiences.

We recommend, therefore, that the audience of the chunk is targetted by the content of the chunk. General purpose chunks may not suffice, however, can be augmented by adding audience specific chunk implementations, and are a good strategy to promote the reuse of chunks. Thus, building a chunk to represent a central aspect, and further developing (nested) audience-specific chunks for audience-specific illustration can be an effective method.

The author must ask the question “Who is my reader? What are his technical abilities?”, and develop chunks to suit.

¹ of course, source code must be comprehensible by other programmers also

8.3.2 Atomic Chunk Mapping Must be Strong

When two chunk types represent the same unit of abstraction, both chunks should maintain a strong mapping such that the cohesive nature of one chunk is reflected by the other. The scope of all chunks in an atomic set, therefore should be the same — both chunks should effectively restate each others implementation. The only difference should be in the audience, and therefore, nature, of expression.

8.3.3 Consider Physical Chunk Scope as a Cohesive Measure

A chunk's physical scope should not cross the boundaries of the abstraction that it represents². It is outrightly bad style to use a chunk to transgress the scope of a the abstraction it is meant to represent. A simple (program source code) example illustrates how this may occur:

```
@
<<define function count>>=
int count(int counter) {
    <<increment counter>>
@
<<increment counter>>=
    return ++counter;
}
```

Note the incorrectly scoped `<<increment counter>>` chunk. It contains the closing parenthesis which is logically part of the `<<define function count>>` chunk's scope.

Whilst syntactically correct, a transgression of scope has occurred. Such use of literate programming can *add great complexity* to program comprehensibility. Literate programming, in this case, undermines the strength of the programming language. Parallels can be drawn between all other possible chunk types — the interruption of prose that is illogically contained in

² A chunk can draw from the concept of module cohesion [60] — the functional relatedness of the chunk's content must be strong.

separate chunks, for example.

A correction of the previous example would resemble:

```
@
<<define function count>>=
int count(int counter) {
    <<increment counter>>
}
@
<<increment counter>>=
    return ++counter;
```

8.3.4 Use the Consequence of Cohesion To Determine a Chunk's Size

The size of a chunk should be a consequence of a chunk's cohesion. Length should not be an issue in the scoping of chunks. Questions such as “Is this chunk too short?”, or “Is this chunk too long?” should not drive the content of a chunk (unless the underlying methodology requires it).

8.3.5 Distinguish Comments from the Source Code

Source code comments should be contained in separate chunks to the source code they represent, and aptly typed; **code comment**, or more specifically, **Java code comment** for example.

In traditional LP, it is possible, and sometimes warranted, to find source code comments alongside the source code. Literate programming did not (intend to) rid source code completely of comments. TBLP facilitates the distinction of source code to comments, and thus, allows the conditional inclusion of comments given theme processing. ‘Clean’ source can therefore be easily generated (an XSLT stylesheet instruction can be included to ignore comment chunks, for example), as the commented version.

8.3.6 *One chunk — one idea.*

Using a chunk to contain only one idea will not only aid the understanding of the implemented code chunk, but also the implementation, and thereafter maintenance, of it. The following excerpt from [54] (adapted to **Noweb** source) shows the combination of two separate abstractions using the one chunk. It illustrates also the utilisation of a chunk name as (a missing) documentation chunk. This is also not recommended.

```
<<Seed the random number generator using
    command line argument 1>>=
if (argc == 2)
    srand48(atoi(argv[1]));
else {
    srand48(7);
    cerr << "*****Rerun with one command line argument, an
integer,"
    << "to get a different scrambling of the output." << endl;
}
```

Two possible source description chunks have been concatenated into one chunk. The `<<Seed the random number generator using command line argument 1>>` chunk would be better separated into: `<<extract seed>>` and `<<seed the random number generator>>`, and thus present the following literate equivalent:

```

<<seed the random number generator>>=
    <<declare seed variable>>
    <<extract seed>>

    srand48(seed);

<<extract seed>>=
    if (argc == 2)
        seed = (atoi(argv[1]));
    else {
        seed = 7;
        cerr << "*****Rerun with one command line argument, an
integer,"
        << "to get a different scrambling of the output." << endl;
    }

<<declare seed variable>>=
    int seed = 0;

```

Our solution creates more cohesive chunks. Note the extraction of the seed for the random number generator is now delegated to a separate chunk. This chunk can further be sub-divided, if necessary, depending upon the complexity of the operation. This allows future alterations to the generation of seeds more easily manageable. It dissociates the seed generation from the generation of the random number itself.

Our solution is not perfect. We have not added supporting documentation. We also have not further chunked the `<<extract seed>>` chunk into its possible cohesive units. The error message should be handled separately to the extract seed chunk. Reporting and detecting this error is functionally separate from determining the seed value. The point that we make, however, is that chunk declarations should contain one idea only.

8.3.7 *Chunk first, code later*

One of the difficulties of literate programming is that it is easy for the programmer to slip into a mode of reverse literate programming.

reverse literate programming Reverse literate programming is a method employed to translate existing source code into literate code (and is discussed in depth in Appendix F on page 356). It is the opposite, in terms of process, of developing chunks, documenting them and their intent, and then populating their body with the appropriate content source code. Knuth’s idea of psychological order is hence, applied in a post-hoc manner.

Developing chunks in a post-hoc manner, forgoing preemptive chunk decomposition in favour of adding unrelated content to an existing chunk, is bad practice and reduces the cohesion of a chunk.

We quote Parnas [62]:

Documentation that has been created after the design is done, and the product is shipped, is usually not very accurate. Further, such documentation was not available when (and if) the design was reviewed before coding. As a result, even if the documentation is as good as it would have been, it has cost more and been worth less.

8.3.8 *Dissociate intent from implementation*

Chunk implementation and chunk intent should be expressed separately. Separating these two factors can aid in the development of higher quality programs. A chunk should aim to describe either (i) the implementation, or (ii) the intent; not both.

Implementation explanations typically utilise source description and grouping chunks. Source description chunks are used to describe low level programming implementations. They employ a low level of abstraction. Similarly, grouping chunks — chunks that are utilised to collate other chunks (as atomic units), rather than being used to represent the immediate chunk content — can employ a low level of abstraction to represent data. Chunk names such as:

- included header files

- definition for findmember
- exported functions
- variable definitions
- insert node in linked list
- linear search for element in array

are indicative of implementation oriented chunks. They are not necessarily concerned with describing a concept, but rather, with collating like program instructions. They tend to answer the question “How is this implemented?”.

“What is the program attempting to do?” is asked by *intent* oriented literate chunks. They concentrate on explaining the intention of what function the program is to perform. The difference between intent and implementation literate code may seem subtle, but both are conceptually estranged.

It has to do with the level of abstraction that the programmer is attempting to embody in the literate program. The higher the abstraction, the closer to an intent oriented chunk. The lower the level of abstraction; that is, the more pragmatic the chunk, the more the programmer will tend to re-represent the source code.

The following example illustrates a situation in which the programmer has mixed intent with implementation into one ill-fashioned chunk.

@ A search for a member entails a traversal of the sorted binary tree, which, in this case is implemented as a two-dimensional array of values.

```
<<binary search for member>>=  
    int memberFind (char *name) {  
        int i;  
        for () {  
            i *= 2;  
            i += 1;  
        }  
    }
```

The following solution is conceptually clearer, however. It shows that <<search for member>> is an abstract concept represented by an intent oriented chunk. It is implemented using source description chunks, namely <<variables for member search>> and <<traverse binary tree>>.

@ Given a member's name, we return a member ID after searching the [[member_t]] data structure.

```
<<search for member>>=  
    int memberFind (char *name) {  
        <<declare variables for member search>>  
        <<traverse binary tree>>  
    }  
  
<<traverse binary tree>>=  
    for () {  
        i *= 2;  
        i += 1;  
        <<determine if member is found>>  
    }
```

It clearly separates intent from implementation³, thereby allowing room for greater emphasis of both of these factors. This means that although the implementation may change, the intent is able to remain the same. It allows the flexibility to alter the implementation details if so needed. It also allows the distinct elaboration of which particular searching method is used, and reasons thereof, in the documentation accompanying the intent oriented chunk.

The first example combined the intent with the implementation. Although this is often possible with functions, literate programming operates with a finer granularity, thereby allowing us to separate the two to greater effect.

Notice how the improved example's <<search for member>> documentation does not specifically associate itself with actual implementation issues. This is left to its lower level content. If the data structure changes, the conceptual structure of <<search for member>> may remain intact, or is easily

³ a comparison can be drawn between the OO concept of encapsulation

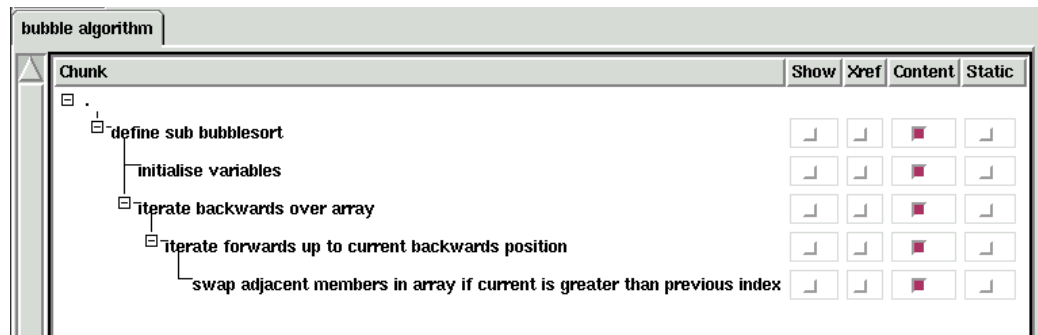


Figure 8.1: Hierarchical view of the bubblesort algorithm utilising the CBDE's tree view.

altered.

8.3.9 Create Self-Documenting Hierarchies

A self-documenting hierarchy is created by a set of chunks that are (1) aptly named and (2) aptly nested to represent the structure of a software system. The theme-tree view of the CBDE facilitates the exploitation of such a view⁴.

An good chunk hierarchy is able to show, without requiring the querying chunk content, the structure of a theme. In particular, it can help the reader:

- determine parts of immediate interest,
- determine the overall structure and method of design of the entity,
- orient his, or her, method of thinking to that of the writers', and
- allow an abstract understanding of the document.

For example, the set of code chunks illustrated in Figure 8.1; if nested and labelled properly, a pseudo-code, or APL (abstract programming language), is generated:

⁴ tools like Leo and Jaba also promote such views

8.3.10 Use Smooth Transition Between Levels of Abstraction

Quantum leaps from abstract chunks to low-level chunks make for quirky reading and are not reflective of the programmer's true thought process of program development. We highlight the need for the middle ground in literate programming.

Consider this excerpt [38]:

```
<<service routines>>+=
insert_parent( ART *current, ART *newart )
{
    ART *temp;

#ifdef DEBUG
    printf("@@@@@@ insert %s as parent of %s/%s\n",
        newart->message_file, newart->message_id,
        current->message_file, current->message_id);
#endif
    temp = malloc(sizeof(ART));
    <<copy current to temp>>;
    <<copy newart into current>>;
    <<link temp as current's child>>;
    <<clean up after parent insert>>;
}
```

`<<service routines>>` is an additive code chunk. It is a grouping chunk with an extremely wide scope; its name indicates that its content is a number of miscellaneous functions that allow common, low-level functionality of the program to occur. The generic nature of the `<<service routines>>` chunk contrasts greatly with the low-level nature of its nested chunks, however. The next four chunks, from `<<copy current to temp>>` to `<<clean up after parent insert>>` represent low-level operations that insert a node into a linked list.

Note that `<<service routines>>` utilises the `insert_parent()` func-

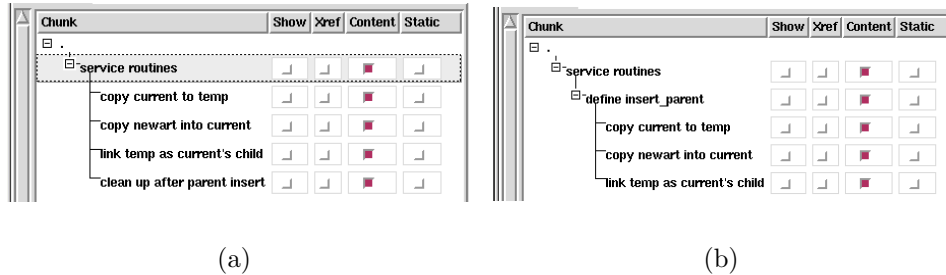


Figure 8.2: The level of abstraction in a chunk hierarchy should reflect the software’s structure.

tion in place of a chunk. We consider this bad practice (see Section 8.3.14 on page 219). Extracting a sub-hierarchy of these nodes, we arrive at the following:

A chunk hierarchy of this LP excerpt is displayed in Figure 8.2(a). Note the distinct gap of abstraction between `<<service routines>>` and its children — one may incorrectly assume that `<<copy current to temp>>`, for example, is a function.

The hierarchy in Figure 8.2(b) is more reflective of the LP’s actual hierarchical structure. We have simply added the `<<define insert_parent>>` chunk. This hierarchy reads more intuitively than the original. Effects of inserting this one chunk would also break up the accompanying documentation to associate more closely with each chunk. In other words, documentation would not be associated, in a monolithic manner, with one chunk only.

Note that the above example is one theme view of the literate program. Any number of others could be developed.

8.3.11 *Be lazy — write self-documenting chunk names*

Choosing a good chunk name is imperative to maintaining comprehensible literate programs. An appropriately named chunk can divulge a lot of information about the chunk’s content, without the need for further elaboration by supporting documentation chunks. One should not fall into the trap of replacing a chunk’s name with documentation, however (as illustrated in the example in Section 8.3.4 on page 205).

The following equation is a useful method of viewing the balance between descriptive chunk documentation and the chunk name. (We assume two chunks exist in an atomic relationship. For example a **requirements specification** chunk and its supporting documentation chunk — **requirements specification-doc**.) It is important that the documentation and title of an atomic chunk match the chunk’s implementation.

$$\textit{documentation} + \textit{title} = \textit{implementation}$$

An atomic chunk that conforms to this equation is considered well-balanced.

*well-balanced
chunk*

Choosing a good chunk name means choosing what the nature of the chunk being developed is and determining a representative name. This is not a straightforward issue. Representing the chunk’s content and describing the intent of the chunk will commonly result in two different names. For example, the chunk `<<iterate over array>>` could also be represented more abstractly as `<<sift through member list>>`. The more abstract name has the advantage of decoupling the name of the chunk from its implementation, however, if the chunk names in context are sufficiently descriptive, it may be pertinent to use a low-level descriptor. Therefore, a chunk’s name should be appropriately descriptive, and also appropriately usable in the context in which it is declared. We recommend that chunk names are developed in a contextually unaware manner; however, we recognise that this is not always feasible.

As a rule-of-thumb, a code chunk with a ‘low-level’ label is likely to require more abstract documentation. A code chunk with an abstract label is likely to require more low-level documentation. Moreover shorter chunk names tend to be more abstract, whereas longer names, tend to be less so.

A **documentation** chunk in traditional LP does not receive a name label. We recommend, as a standard for the naming of source code documentation chunks in TBLP, that they adopt the name of the source code chunk they represent. Consideration of an appropriate name must be given where the mapping is not one-to-one; a high-level source code documentation chunk is used to represent two or more source code chunks; or one or more source

code documentation chunks are used to describe one source code chunk.

If a documentation chunk's name eludes the author, this may be an indicator that the documentation chunk content is not cohesive, or even entirely unnecessary.

An example of **source code documentation** that adopts the naming of the **source code** chunk is illustrated in Figure 6.1 on page 118. Note that <<The main program>> chunk exists twice in the theme hierarchical tree-view. One is a **source code documentation** chunk and one is a **source code** chunk.

8.3.12 *Avoid Temporal Commentary — Reduce Chunk Cross-Coupling*

This should apply to procedural and temporal — laterally themed chunks.

Temporal chunks are psychological ordered such that only when presented in a sequential manner, within context of surrounding chunks, is their full meaning divulged. Given a worst case scenario, a literate program developed in such a fashion would only be comprehensible only if it was read from beginning to end. This would render literate documents unusable as works of reference.

This approach should be avoided when possible. There must be a compromise between a psychological *flow* and the comprehension of a chunk as a standalone unit. Neither should dominate outright, however.

A chunk can be perfect (a standalone all-comprehensible entity) without including all data and documentation in that chunk. We address this extreme solely to prove our point.

The programmer should keep in mind that chunks are predominantly used by an audience:

as a reference source: Enabling the reader to skip to a particular section and, by and large, understand the nature of the section without needing to start reading from the beginning of the document.

as a theme composition source: Each chunk should form part of the greater whole (the software system) such that together, all chunks are able to convey the intended message to the audience.

As a reference source, a chunk is considered as a self-contextual entity. Surrounding chunks, and links to other themes, which that chunk is included will help divulge more information about the chunk.

As a theme composition source, enough detail should be present to generate a psychological flow amongst a set of other chunks.

Careful consideration of these factors can help create better literate programs.

8.3.13 Chunkify Programming Language Abstractions

Programming language abstractions should explicitly address the inherent cohesiveness of a programming language's abstractions by using a chunk to represent it. This method combines well with the hierarchical approach suggested in Section 8.3.9 on page 212 and is discussed in detail in Appendix F on page 356.

There exist three compelling reasons that suggest this practice should occur:

1. a chunk is a cohesive entity and at some point this cohesive scope should match the cohesion of a routine.
2. psychological scope recognises that chunks and programming language abstractions may differ in scope, however, programming language abstractions should be explicitly recognised.
3. semi-automated generation of API-based theme literate documents is thereafter a simple process and can be generated at the chunk level (as opposed to the programming language level).

A good example is presented:

```

<<TreeMapApp>>=
    <<TreeMapAppImports>>
    public class TreeMapApp extends JFrame {
        <<TreeMapAppProperties>>
        <<Constructors>>
        <<buildGUI>>
        <<treeBuilding>>
        <<JTreeMaintenance>>
        <<TreeMappAppMain>>
    }
    @

```

The XML descriptions of treemaps need to be read from both files and strings. Fortunately, Java's streams are very flexible.

```

<<treeBuilding>>=
    <<buildTreeFromFile>>
    <<buildTreeFromString>>
    @

```

Given an XML file name, and access to an appropriate DTD, a SAX parser (a [[TMSaxHandler]] from the treemap package) can generate the treemap from the XML description.

```

<<buildTreeFromFile>>=
    public void buildTreeFromXMLFile(String fName) {
        ...
    }

```

Note that each major programming abstraction is chunked.

8.3.14 Don't Use Implementation-Level Commands as Chunk Substitutes

Neither program source code, such as function definitions, nor text formatting language commands (such as the `LATEX` `\section` command) should be utilised as chunk substitutes. Just as a chunk is not a substitute for a routine, nor is a routine a substitute for a chunk. Section 8.3.10 on page 213 presents an example of bad use of implementation language abstractions as chunk substitutes.

Chapter IX

Future Work

Our concept of theme-based literate programming facilitates the development and view of software systems using multiple perspectives. In Chapter 5, we introduced the TBLP model, which is comprised of the generic chunk model, the theme model, and the processing model, and an XML-based development framework. We illustrated in Chapters 6 an authoring tool (the CBDE) to facilitate the authoring stage of the theme document development process.

We wish to progress the paradigm of TBLP. The CBDE is an experimental authoring tool demonstrating the manner in which we intend TBLP to progress. Using the CBDE, we have demonstrated that multiple themes and multiple formats of these themes can be developed. Although it has facilitated the experimental approach to TBLP that we have required for this thesis, further research is needed to progressively develop such supportive tools that withstand rigorous use¹.

Specifically, we see future work falling into the following categories.

Extending the TBLP model: Enhancing the relationships between chunks, and therefore enhancing their development.

Tool support: Developing tools to support the TBLP stages of development.

Human Computer Interaction: support and development: Facilitating effective and efficient theme authoring and viewing.

¹ as opposed to academic exercise

9.1 *Extending the Model*

Theme-versions: As discussed in Appendix E.1 on page 353, the theme model will be extended to include a version attribute, thus facilitating the maintenance of theme version hierarchies.

Sub-themes: Sub-themes are themes that comprise another theme. The theme-model will be extended to allow sub-theme composition. Similarities between chunk nesting and theme nesting can be drawn. A key consideration is whether themes are to be composed of a combination of theme references and chunk references, or solely of one type of chunk (either chunk references or theme references).

Issues of correctness of themes arise, and requires further work to determine when themes should be used instead of chunks. The results of this investigation will go toward forming a styleguide for TBLP.

Dynamic chunk attributes: Our generic chunk model, as proposed in Section 5.2.2 on page 98, possesses a **name**, **type**, **chunkID**, **version**, and **variant** attribute. This fixed set of attributes may prove limiting to authors of literate themes.

We believe that dynamically assigned chunk attributes will allow enhanced chunk definition. For example, a “language” attribute would allow language-specific formatting, or a chunk’s conditional processing in a theme document. For example, the **c-code** chunks, in the example in Chapter 6 may be more generally typed as “code”, but receive a **language** attribute of “C”, denoting that the chunk contains C source code. Effectively, this would enable a formal method of sub-categorisation.

If dynamic attributes are introduced to TBLP, future work must also determine the minimum set of chunk attributes a chunk must possess. We suspect that the **version** and **chunkID** attributes will form this minimum set.

Enhanced tool support is necessary to accommodate this functionality.

Enhanced model support: Future work will determine the worthiness of introducing an extended chunk model that allows authors to define multiple chunk relationships. This enhanced model would support the definition of parent-child and sibling-sibling chunk relationships.

Nesting relationships are necessary to represent step-wise refinement. Chunk version hierarchies can also be represented as a series of nested relationships. Nesting can also be useful to facilitate a sibling relationship rather than a hierarchical one. For example, a `class description` documentation chunk that is associated with a relevant `java-code` chunk represents a sibling relationship between the two chunks; the parent chunk merely maintains sibling order. Essentially, these two chunks form an atomic relationship. The enhanced model would allow the author to explicitly stipulate atomic relationships.

Allowing the definition of such multiple relationships enables the TBLP author to stipulate how the chunks involved in a relationship should appear in the theme document. And given appropriate tool functionality, how they should be treated by the tool.

Abstract chunks that are used for the purpose of collating and ordering chunks also warrant further investigation. These chunks resemble themes, however, would not contain content themselves. To reference the set of chunks contained by an abstract chunk, the reference to the abstract chunk is made.

Facilitating the definition of such relationships will allow full extensibility of TBLP development. The visualisation and representation of these relationships would aid in determining and understanding software structure.

Note that these types of chunk representations are possible using the current model, however, the current model does not elegantly distinguish between chunk relationship types. Should the current model be kept in favour of these suggested enhancements, it is left to supporting TBLP tools to accommodate this functionality². The shortcoming

² Whether the enhanced model is supported by the tool or not, tool-based support is

of not offering such enhanced model representation is that distributed users of the repositories' chunks are unable to exploit the author's intended chunk relationships.

Another avenue of further work is to investigate model support for multiple chunk names. This will enable specialised names to be attributed to chunk given their surrounding context, whilst representing the same chunk content.

9.2 Tool Support

Multi-media chunks: Although our focus of TBLP is predominantly software-oriented, TBLP can be applied equally well to other domains of document preparation and, with further research, even adapted to suit working with multiple media such as video, audio, two and three-dimensional diagrams and notations. Tool support for such development must be investigated.

Repository management: Repositories store each and every version of each chunk. It is likely that not every chunk will be referenced, or indeed, desired to be referenced. Facilitating a repository purge will positively affect the repository's size, and thus, the efficiency of supporting tools. Supportive functionality that indicates the existence of unreferenced chunks, and that provide an indication of a chunk's life-span versus the number of times and when it was referenced will aid the author to commit 'safe' purges.

Research into chunk storage and compression methods can alleviate the replication of data that is likely to occur in a repository (and theme source documents). Chunk versions, especially at the end of their versioning life, are likely to receive incremental alterations. Much of their content is therefore replicated. Good compression techniques with fast access times must be investigated. Our thoughts are oriented towards an approach similar to the MPEG compression system whereby chunks

required regardless

are differentially compressed and every n 'th chunk (in a chunk version hierarchy) serves as a reference of differentiation, thus facilitating speed of access and good compression (depending on the compression algorithm used).

As discussed in Section 7.3.1 on page 190 investigation into the best means of chunk storage must be conducted. Whichever method is used, the repository storage and authoring tool functionality must be kept separate.

IDE support: Crucial to TBLP's success is its availability of use. We do not intend TBLP to replace existing editors and tools — ideally the TBLP model would be supported by existing tools. Whether a purpose built IDE(s) support is developed, or use can be made by adapting existing IDEs (such as Microsoft's .NET, Rational's DevelopmentStudio, and TogetherSoft's TogetherJ, for example), through their programmable API's, needs to be investigated.

Research into TBLP software development may reveal the need for specific niche-supportive tools such as web-authoring, test-programming, maintenance development tools. Niche-level support could be offered via extending (or even restricting) general purpose tool support through the use of programmable IDEs.

Content-based support: Content aware tool support can indicate artifact relationships between chunks in order to facilitate automatic and manual chunk manipulation. Facilities such as programming language support that recognise artifacts such as class, method, and variable identifiers can indicate the use of these identifiers in related chunks.

Content aware tools can be used to highlight chunks that may be affected by edits made to a given chunk. For instance, two chunks share the same identifier. The identifier is renamed in one of the chunks. The tool highlights the second chunk notifying the author that his changes affect another chunk.

Moreover, tools that recognise chunk relationships (atomic relationships, for example) can remind the author that potentially unattended updates exist from prior edits to a related chunk.

Functionality such as content search for chunk location should be provided by supporting TBLP tools.

Inter-process development: The TBLP framework supports the utilisation of sub-processes throughout its three stages of document development (see Section 6.2 on page 117). Future work will involve the development of these sub-processes to provide TBLP support. For instance, source code parsers that return the XML equivalent can be developed to facilitate automatic markup of code type chunks. These code chunks could then be further transformed into UML notation, for example, and important relationships drawn between the corresponding artifacts through the use of themes. Another example of sub-process development is to develop utilities to transform legacy software repositories to TBLP usable source.

Template development of XSLT and XSLFO stylesheets to support theme document formatting is another example.

Chunk-based document rollback: An elegant method of committing entire document rollbacks based on the version of a given chunk must be explored. There are several possible methods (ranging in degrees of elegance) to achieve this, however, the overhead in their execution may prove a shortcoming.

One method is to commit a full hierarchical version update for each chunk that references an edited chunk. The theme version will also be updated to a new version (assuming theme versions are implemented). Effectively this means that a chunk's edit would cause all chunks that reference it to also be updated with a new version number. This would need to occur for each chunk edit to provide granular rollbacks; however, the recursive and continual increment of chunk versions combined

with the incremental theme version data places great overhead on the processing and storage media required.

Auto-Theme Development: Themes can be automatically generated by tools that retain information such as common chunk navigation tendencies of users. Thus, a scheme to automatically generate the most well-travelled path, for example, can be automatically generated. Variations of this idea abound.

Legacy repositories: Research into the (automatic) conversion of existing code and documentation bases to TBLP chunks must be performed to explore the most suitable method to make reuse of legacy repositories of software.

For example, the documentation of software bugs and their fixes are often stored separately to the affected source code. Utilising these legacy storage methods to maintain their structure, or outrightly converting these legacy repositories to a TBLP repository are two options. We stress that regardless of the repository-based approach, tool support must afford an interface to a repository(s), and are therefore independent repository solutions. Thus, the TBLP repository should not be coupled to the repository. The TBLP development framework treats the repository as a distinct object, as should supporting tools.

Developing utilities to interface and parse with legacy systems will be investigated and is likely to be the work of commercial entities who (perhaps) wish not to reveal their proprietary storage formats, whilst providing good customer service, of course.

A (tentative) recommended conversion, or reverse engineering, practice from OO and imperative languages is presented in Appendix F on page 356.

Multi-repository, distributed TBLP Future TBLP tools must be developed that facilitate theme development using multiple repositories.

Issues that must be investigated are:

- The use of namespace support for multi-repository theme development. XML provides a namespace standard.
- Will repositories be considered ‘local’ or ‘external’ (geographically distributed)? Should the concept of local repository exist, or should each repository be considered external.
- Chunk ownership. If a chunk exists in repository A, however is referenced in repository B, and then updated, which repository stores the new chunk version? We tentatively consider the use of ‘creative’ ownership, whereby the repository in which a chunk is first created stores also the version hierarchy regardless of where the edit emanated from. This suits the external repository model, whereby all repositories are considered external. However, slow network connections can delay response times of chunk updates.
- Should chunk updates in external repositories be effected in all referencing themes? Perhaps there is a need for more elaborate tool functionality, which allows differentiated control of chunks that are referenced from alternative repositories (with the external repository scheme, however, there is no concept of a ‘local’ edit). Should it be to the user’s discretion as to which repository is updated?

9.3 *Human Computer Interaction*

The presentation of the TBLP development process and its supporting tools, such as the CBDE, is paramount to TBLP’s success or failure. Representation of the enhancements suggested in this chapter must be in a manner that facilitates transparent and intuitive use.

- TBLP causes a proliferation of chunks, and these must be represented in an intuitive and functional manner by interfaces that facilitate chunk selection and retrieval.
- Hierarchical, tree-based and textual views, as displayed by the theme tree view and theme text view widgets of the CBDE, respectively, are an

effective representation of programming structure. Chunks, as discrete units of abstraction are amenable to other methods of visualisation also. Investigation of three-dimensional visualisation techniques, for example, as a method of both visualisation and development of themes and repositories, and also theme analysis, will greatly enhance theme-based literate programming. Techniques such as holophrasting, DOI, fan-in fan-out, and fisheye views are also useful considerations in combination with the existing and future visualisations of themes.

- The CBDE employs repository-based editing because we believe it more transparently reflects the chunk and theme models. Importantly, this technique also facilitates contextualised editing. Empirical evaluations must be conducted, however, to determine whether in-theme editing or repository-based editing is indeed more effective³, and in which cases either is better, or indeed, best. It may eventuate that the author is able to choose which editing environment he uses.
- XML provides an excellent medium for theme and chunk representation, however, is not well suited for use in programming environments. It is our expectation that future TBLP authoring environments will not represent themes as XML documents, but in a more ‘readable’ format.
- Most existing tools, including the CBDE, produce static documents, thus documents are easily rendered obsolete. The dynamic display of themes, using XML technologies such as XSLT and XSLFO transformations, is an important area of future research. Browsers utilised for both editing and browsing must be investigated to avoid the limitations of static documentation.

Further work into the development of a set of standards or guidelines that can be employed by TBLP authors and incorporated in tool-based support to facilitate the development of themes must be performed. TBLP provides an extremely flexible environment that can easily and unintentionally, be

³ and not just theoretically superior

misused. The development of guidelines to this effect will greatly benefit authors. Chunk naming standards are an important step in this direction.

And finally, not to be understated, we are yet to derive a term that befits the composition of and processing of chunks. Knuth introduced *weave* and *tangle*; we look forward to much further deliberation as to the choice, or creation, of a similarly descriptive verb⁴.

⁴Dr. Neville Churcher has expressed a preference to “wangling”.

Chapter X

Conclusion

In this thesis, we have introduced the paradigm of theme-based literate programming (TBLP). TBLP is driven by the need to modernise literate programming (LP), as invented by Donald Knuth in the early 1980s, to cope with present-day software engineering requirements.

The inability of existing LP tools (both traditional and XML-based) to elegantly meet the requirements of current and future software engineering is a result of the fixed-model approach they adopt. Specifically, the traditional LP model suffers from the following limitations:

Fixed chunk model: Only two chunk types exist: documentation and code.

This distinction becomes blurred when producing artifacts such as design documentation, design specifications, UML notation, and working with the likes of database query languages, XML, HTML, client-side and server-side programming languages, for example. One is no longer able to accurately categorise each of these as either documentation or code.

Asymmetric processing model: Documentation and code chunks are treated differently; only code chunks can be nested. LP is therefore constrained largely to the construction phase of the software development life cycle (SDLC). This is because it is impossible to introduce documentation (requirements analysis, for example) that documents the construction documentation. Documentation chunks cannot be included in the tangled source as a consequence of the differentiated treatment.

The processing model and chunk model are combined: The ordering of chunks in the literate document is derived from their lexical ordering

in the source file (web). There is consequentially a limitation of one psychological ordering; it is impossible to reorder chunks such that they appeal to more than one audience.

Another limitation is LP's inability to adequately cope with the object-oriented concepts of overloading and overriding. Not only does OO tax the indexing features of LP tools, but also the chunk identification system commonly disallows an overloaded chunk naming scheme.

Our contribution, in this thesis, stems from the introduction of a generic chunk model. This model differs from the traditional chunk model because:

Content is separated from ordering: Chunk content is stored separately from chunk ordering, thus allowing multiple chunk orderings; and therefore the development of multiple psychological orders (or themes).

Multiple chunk types may be defined: A chunk may receive any author-attributed type, and

All chunk types may be nested: The nesting functionality of the traditional code chunk is supported by all chunk types.

The separation of content and ordering allows software systems to be developed and presented using multiple perspectives — we call each of these perspectives a theme — making TBLP amenable to current technologies such as AOP and Hyperslices. Indeed, TBLP is language and methodology un-specific. Furthermore, TBLP facilitates “equality of concerns” and, thus, does not impose a ‘literate’ style of programming upon the author; source code is ‘just another theme’.

Multiple chunk types allow chunks to receive independent processing and formatting. Furthermore, because all chunk types can be nested, chunks can be processed and formatted depending on their relative position (hierarchical and sibling order) and the type of their neighbouring chunks in a theme. Chunk nesting also allows the development of higher-order documents whereby chunks of various types may be nested. This enables the elegant elaboration (and distinction) of all phases in the SDLC, for example.

We also introduced a pipeline theme document processing framework. This framework consists of three main stages; (1) authoring, (2) transformation, and (3) formatting, where input for each stage is the output of the previous stage.

We chose XML as a medium of representation, which eventuated as a wise decision because it fits in well with the pipelined processing development. Using XML as the interfacing medium of each process, the pipeline framework supports process inclusion and/or substitution. We have shown that it also provides a flexible presentation system, whereby theme documents are transformed into multiple document formats, such as HTML or source code.

The Context-Based Development Environment (CBDE) facilitates the authoring stage of the document development framework. We developed this tool as a proof of concept and to influence the further development of TBLP tools.

A chunk versioning system was discussed and employed using the CBDE. Chunk versioning allows the development of chunk version hierarchies. Programmers can rollback to previous chunk versions and design historically-oriented themes, as we showed, to elaborate on a chunk's evolution.

We believe that our research will have a significant impact on software development and software comprehension. This approach of TBLP can be applied equally well to other domains of document preparation and even adapted to suit working with multiple media such as video, audio, two and three-dimensional diagrams and notations. Before that happens, however, its success lies in the hands of future research and efforts to provide suitably supportive environments and effective standards.

Appendix A

Thoughts on Literate Programming

Literate programming has well-inspired ideals. It orders programs psychologically. It is thus a seemingly natural solution to the problem of program comprehension and development.

Although LP is conceptually sound, it is often met with resistance by programming practitioners. We attempt, in this appendix to present some of the arguments behind their resistance, and how LP solves them.

A.1 Documentation — How Should It Function?

Even when a program is designed and developed with comprehensibility as a key consideration, years of alterations will slowly see it lose this comprehensibility. Parnas [62] terms this phenomenon as software ageing.

LP couples documentation tightly with program source code to produce an atomic unit. Although other documentation tools attempt to enforce the same coupling, the physically separate nature of the documentation to the source code inhibits this. Figure A.1 illustrates the connectivity between documentation and code sections in a LP source file and in a traditional documentation tool. The distant nature of non-embedded documentation makes it difficult, and often a laborious exercise, to update simultaneously with the program source code. While Parnas' software ageing may be inevitable, LP helps extend the time of this inevitability¹.

¹ rather analogous to an anti-ageing moisturising-cream to keep those wrinkles at bay

Figure A.1: atomic chunks in a file. two files, doc/code interconnection

LP documentation provides a method to include program design as an integrated part of the code development process. Source code becomes a natural extension to the design documentation rather than a separate document.

LP documentation also helps to reduce design entropy:

Ryman [75] defines *design entropy* as a conceptual measure of the discrepancy between the design specification of a system and its implementation. In a traditional setting, where an implementation continually diverges from its design, the amount of design entropy increases to the point where the design specification is of no value for reference or maintenance purposes. Ryman proposes that the amount of design entropy can be reduced only if the design is considered as an artifact (rather than a process) and is maintained in step with the software implementation, through the entire life-cycle of the system. — [31]

In the following sections, we consider LP’s ability to provide abstraction beneficial to human understanding, documentation’s ability to aid the the comprehension of the “big picture” of a program, and finally consider documentation’s benefits as compared with code commentary.

It is interesting to deliberate whether literate programming is at all useful. Is documentation merely complimentary to source code? Or does the act of documentation positively affect the process of source code development? Does literate programming improve on traditional documentation methods? Does literate programming simply add another layer onto the existing programming infrastructure, thereby adding complexity and aiding confusion?

A.1.1 LP Documentation versus Source Code — Audience Specific

“Due to the inadequacy of documentation, reading the source code is the most trusted and widespread means to acquire knowledge about a program. Unfortunately, code is not written to be read by humans: it contains a large amount of details that make it difficult to extract the needed information.” [15]

The ability to present the problem in an efficiently comprehensible manner to the intended audience is the goal of the tangle and weave processes.

While source code can be considered as the definitive documentation source, it is audience-specific. While it may be definitive, it is not readily comprehended by humans — even trained humans in the art of computer programming.

Consider the following statement: “*source code is its best documentation*”. This statement implies that if a program is developed well; if its code is presented in an uncomplicated manner; if variables are descriptive of the data they contain, functions and methods are descriptive of the operation they perform; if consistent layout and coding practices are employed, the source code will be easily understood. This is the approach taken by the Extreme Programming (XP) methodology. This is true. The source code will be understood; but what about the program? What about the “bigger picture”?

It is possible to understand the source code, but not to understand the intention of the program. “I understand what this is doing, but not what it’s trying to do.”

Even in the case of well-written source code:

*it is difficult understand the program for the source*².

Different types of documentation exist. LP does not suggest documentation for documentation’s sake. The right amount of documentation is the right amount of documentation. If the source code is clear and understandable and doesn’t need documenting, then it shouldn’t be documented. There is more to software development than coding, however.

Outside your extreme programming project, you will probably need documentation: by all means, write it. Inside your project, there is so much verbal communication that you may need very little else. Trust yourselves to know the difference. — Ron Jeffries³

Program source code cannot tell its reader explicitly which design pattern was used in its development. Program source code does not describe

² to quote the oft used metaphor (and manipulate it slightly)

³ (<http://www.xprogramming.com/xpmag/expDocumentationInXp.htm>)

user requirements in the context of program development. It does not tell the reader why a piece of functionality was developed in the method that it was; there may have been a range of options from which to choose. Source code does not promote important sections of code fundamental to the programs design and operation, while relegating other sections of code to the background. The list goes on.

And what of the case of poorly developed source code: should the opportunity not exist for the programmer to at least describe the program's intentions? The programmer may not be technically sound, however, LP gives him the opportunity to, at least, inform others of his attempts.

Program source code tells the reader what has been done. Documentation can tell the reader what the source code is supposed to do — if there is a discrepancy, it should be documented. Documentation provides emphasis by highlighting important areas. This is not possible with source code alone.

Many programs do not follow an easy-to-read program structure. It is not readily apparent from perusing the contents of the `main()` function of a large C program, what the procedural nature of the program is; where the important areas of the program source code lie, how the data structures look and work. Even a well documented program will not reveal this information readily if documented in a purely linear (program language order) fashion. Literate programming allows the placement of important ideas foremost in the sequence of the program's documentation. It allows emphasis; it promotes the concept of psychological order.

We have covered some important reasons to document software. It is quite possible that more than one documentation type is written for one (or more) code chunks. Melding these documentation types together in one document, in one psychological order can misplace the emphasis desired by the author. Themes enable the development of software among multiple paths; allowing therefore an author to develop a set of user story themes (for XP), a theme discussing design decisions, a theme discussing “dirty” code, for example.

A.1.2 Perspective Simplifies Complexity

We relate Batory's [4] analogy of geocentric versus heliocentric views to this argument. He argues that we, as software developers, commonly take a geocentric view, thus providing complicated solutions: early astronomy considered the earth and not the sun as the centre of this solar system. Thus, it was with some overly complex mathematical transformations that a correct model of planetary motion was developed — until it was determined that the sun was the centre of the universe. The mathematics of heliocentric theory then proving elegant and simple. The change in perspective is able to simplify core problems. Batory argues that this change in perspective can greatly affect software development. We believe that LP offers the possibility to generate these different views: these views then drive the development of program source code.

Facilitating different perspectives for problem solving enhances the ability to draw from one's domain knowledge and apply it to the task at hand. Currently, no tools exist to combine this domain knowledge with the general semantic knowledge in the confines of one system's solution and definition.

Facilitating perspective expression allows the author to elaborate on the 'derivation of process'. Code refinements, for example, may be driven by customer requirements. Altering the code only, and not the documentation of the requirements is therefore incomplete. By coupling requirements documentation with code, LP provides a solution. The view of development is shifted from writing code to program design. This is why literate programs can generate more elegant solutions than their non-literate counterparts. Although traditional LP is not suited to multiple views required by this perspective-based approach TBLP is an elegant solution.

*derivation of
process*

Even object oriented languages such as Java, which don't follow a flow-based order of execution, benefit from an author-imposed order, such that direction and emphasis is placed on critical areas of the program source code. The distributed nature of classes creates a web-like structure of method invocation. This web is not necessarily similar to the one that would be created using a literate programming style.

Object orientation's concepts are structured around encapsulation, which

involves data hiding. This is different to the literate programming concepts of emphasis and presentational order.⁴ Object orientation develops programs that are essentially composed of classes, which themselves contain methods and data which these methods make use of. A class is commonly a self contained entity. There exists no start point and no exit point. Adding direction to these classes can help greatly in the understanding of a program.

Even though object orientation provides abstraction through encapsulation and inheritance, an ordering of the program's design is usually not evident. Whereas `main()` is not necessarily the best starting point to understand the design of a C program, the initial class in an object oriented program may not necessarily be the best starting point to understanding the overall program's design, either. Literate programming helps define the starting point and a comprehensible flow that ensues.

OO programming is centred heavily on the development of API's. API's generally have multiple interfaces to the world. They do not necessarily have a beginning procedural point. In this case, it is difficult to present to the user (programmer) how to interact effectively with the API in order to develop a program. LP can help in this respect by expressing the interface classes and their methods initially in the document, perhaps giving examples of their usage, etc.

Javadoc is an API documentation tool for the Java programming language (refer to Section 2.9.1 on page 35 for specific details). It helps the programmer determine how to best interface with the modules that compose a program. It does not document how the interaction of two objects will affect the program. It is unable to capture the concept that the 'sum of two parts being greater than the whole'. Javadoc is unable to present "the bigger picture" to the reader. It is also unable to present the flow, or the order of execution undertaken by a program.

⁴ one concept is not better than the other. Both should be used in their rightful ways.

A.1.3 LP Versus Plain Old Documentation/Comments

We have mentioned that documentation should be amalgamated with the program source code⁵. Building upon this, we understand that documentation's purpose is to enhance program comprehensibility — whether this comprehension pertains to maintenance, design, development, or the use of the program itself.

A common question is asked by (unwilling) programmers to whom LP is introduced: “The code I write is commented very well, and laid out in an attractive fashion. Why then, should I program literately?”

Documentation and comments differ. Comments tend to be written in a post-hoc manner. Comments tend to be terse and discuss the program source code — the domain knowledge of the problem is rarely mentioned. Thus, what is comment is simply an elaboration of the general semantic knowledge of the language.

Literate documentation often occurs in a pre-emptive manner; before the source code is written. It explains the intent of the ensuing code chunk. It therefore aids in the development of the source by way of explaining to the intended audience what the source is about to do. Documentation chunks offer to the author the ability to elaborate, even about the smallest piece of code, if deemed relevant. Comments on the other hand tend to be proportionally smaller than the code they represent. Comments are also sparser than documentation tends to be. Documentation is likely to read easier because authors are encouraged to use grammatically correct sentences.

Documentation should not encourage verbosity. Verbosity of expression should be discouraged, in fact. Likewise, concise, terse documentation can be equally difficult to read. Burdett makes the point:

This does not mean that all computer documentation must or can read like a Nobel Prize novel, but neither does it have to read like a military cryptogram. [14]

⁵ note that the view from literate programming's perspective is that program source code is contained within documentation or in the case of web pages, for example, just another kind of documentation

A.2 *An Abstract Process*

You see, wire telegraph is a kind of a very, very long cat. You pull his tail in New York and his head is meowing in Los Angeles. Do you understand this? And radio operates exactly the same way: you send signals here, they receive them there. The only difference is that there is no cat. — Albert Einstein (1879 - 1955), when asked to describe radio

An introductory chapter in operating systems [83] will mention a layer based approach that computers maintain to represent a working architectural paradigm. Layer 0, from the computer scientists' (as opposed to engineers') perspective, deals with circuit level instructions. Every layer on top of this is a more abstract layer towards the final user's expressive language capabilities. [include diagram here showing different layers of general computing layout.] The more abstract the level — the further away from level 0 — the easier it is for humans to understand. Consider writing a quick-sorting algorithm in assembler. Not a particularly entertaining exercise⁶. Expressing this in the C programming language however, would be much more tolerable (if not mundane and mostly pointless — there exist libraries that alleviate such tasks away from us). The greater the abstraction, the easier that tasks become — at least in the limited domain of basic computer architecture. There exists difficulty for humans to instruct computers; the natural method of representation is different to both entities — if both can compromise on a sufficient level of abstract communication, the possibilities for the conveying of complex commands increase.

Does LP act as a mediator between human and computer? Does it represent a layer of abstraction on top of current programming languages, therefore bringing programming one step closer to the programming human's conceptualisations? While we would love to answer "Most certainly yes!", the answer is not so clear cut. The news is good, however!

Although LP itself is a supporting harness for an abstract layer of representation, above the restraints of the programming language, it is not com-

⁶ for those that value their sanity

plete in its abstraction. It still requires program source code to be developed.

If LP is an entirely abstract layer, the programmer would simply communicate to the computer in written (or other) word. It would not be necessary to generate source code.

Instead, it promotes abstraction of thought — LP is a facilitator of abstraction. It is abstract enough to allow an alternative thought processes to be physically elaborated; a given programming methodology to be utilised — it is impartial whether object orientation or structured programming methods are used: bottom-up, top-down, or nucleus centred. LP allows a user to attack a problem using a ‘psychological order’ — something not altogether present in programming languages. Note, however, that there still exists the restraint that the program will have to be built of segments aimed at the computer.

*impartial to
methodologies*

A.3 The Affect of Programming Languages on LP Abstraction

In this section, we introduce the new term of “psychological scope”, “transparency of literate programming”, and “loss of power”. We explain how LP, when used with certain programming methodologies, can be restricted in its ability to represent reader-oriented abstractions.

Is LP affected by programming methodologies such as object orientation? OO’s (object orientation) data encapsulation techniques do not necessarily lend themselves well to the literate programming’s expression of ‘psychological order’. Why? Although LP allows for an abstraction away from the programming language itself, in an attempt to allow the programmer to focus on the problem domain — it allows the author to express and define code chunks in the sequence he sees fit, and in the scope he sees fit, rather than that dictated by the programming language — there exists a medium where *literate programming abstractions and programming language abstractions naturally coincide*; chunk definitions and the programming language constructs map closely. There is a common restrictive ground whereby both literate chunk definitions and programming language abstractions are similar, both in scope, and naming convention.

This concept is touched upon by Nørmark [56] and Cockburn [19]. To

quote Cockburn:

Object encapsulation makes the notion of a ‘psychologically correct order’ a weak one in object-oriented programming.

We term this as *loss of power of abstraction*.

A.3.1 The Power Paradigm of Literate Programming

The usefulness of literate programming, the power it contains to free the masses from the confines of the imposed structures of the oppressant programming language(s) is diminished by the very programming language itself⁷.

When defining a new class, it is usually necessary to begin a new code chunk; just as a new function in an imperative programming language would usually require a new chunk. The closer the programming language methodology constructs are to demanding a new programming abstraction, the more redundant a literate chunk is. This is what we determine as a ‘loss of power’.

For example, the following program source code:

```
public class automobile {
    private int speed;

    public accelerate() {
        ...
    }
    public brake() {
        ...
    }
    public turn() {
        ...
    }
}
```

would likely form the following literate program source code:

⁷ A little melodramatic, but hopefully, the point is taken.

```

<automobile class>=
  <define and initialise automobile variables>
  <automobile accelerate method>
  <automobile brake method>
  <automobile turn method>

```

Another way to view this concept is to consider the *transparency of literate programming*. The more the literate programming code chunks conform to the programming source constructs, the more transparent the literate program source becomes.

*transparency of
literate
programming*

To further explain loss of power, let's attempt to envisage a situation that renders LP useless: if it were possible to create a program which consisted only of class constructs, then for each class construct, we would use one literate code chunk. Assuming the class constructs could be ordered in any manner, there would be no advantage to using a literate programming approach. We would merely use (hopefully) well named class constructs which would contain effective comments throughout.

The conformance to these programming language abstractions can be swayed somewhat by the author, however, an issue of style arises when doing so. Chapter 8 considers the use of style in literate programs by presenting a set of guidelines. Further research in the area of LP development is needed, however.

A.3.2 Psychological Scope

It is interesting to view the concept of literate 'power', introduced in Section A.3.1 on the facing page, in light of Knuth's own idea of 'psychological order'. To do so, we introduce a new term — 'psychological scope'. It concerns itself with, and gives further definition to the limitations imposed by programming languages on literate programming.

*psychological
scope*

While psychological order pertains to the natural dominant sequence of code chunks that lead to a program's explanation, psychological scope deals with the size of a chunk's definition space; the size of the human-oriented

abstraction that it encapsulates. We argue that human-based abstractions and programming language based abstractions do not always map cleanly. The distortion imposed by programming languages on the psychological *scope* can cause the programmer to restrain, or expand the size of the idea, or abstraction, represented by a given chunk. If the psychological scope of a chunk is affected by the programming language, it must then be true that psychological scope affects the psychological *order* of chunks.

Psychological scope gives definition to the development of chunks whose scope is purposely altered to conform to programming language abstractions.

An example helps to illustrate our concept of psychological scope. The literate programs in Figures A.2, A.3, and A.4 are excerpts of OO programs, written in **Noweb** source, that generate statistical data of a given set of numbers. Consider the initial flow in Figure A.2. The data set is stored in an array which is then processed to determine the average or mean of the contained data. To implement this program, and in the interest of reuse of code, a **Statistics** class is written along with the **Query** class, which queries the **Statistics** class.

The chunks: `<<access file>>`, `<<populate statistical tables>>`, and `<<compute average of data>>` are psychologically ordered. The **Statistics** and **Query** classes must be ‘chunked’ and also included in the literate program. Although important to the development of the program, the inclusion of these two classes necessitates their chunking; chunks must be developed to encapsulate these programming artefacts. Namely, we create two new chunks to contain the classes: `<<Statistics class>>`, and `<<Query class>>`. The programming methodology and language have enforced an explicit consideration to the development of new chunks in order to encapsulate the programming abstractions.

Note the difference in approach of chunking the scoped code segment in Figures A.3 and A.4. Figure A.2 uses the `<<compute average of data>>` code chunk within the scope of the **average** method. Figure A.3 includes the entire method in the code chunk. Figure A.4 uses finely granular scoping to achieve what both Figures A.2 and A.3 have: the **Statistics::average** method includes the method declaration and references the `<<compute average`

of data>> chunk. <<compute average of data>> implements the body of the `average` method. This method essentially uses psychological scope to override programming abstractions.

Chunks that would normally not have been considered if dealing only with the LP language, must now be developed. This is a consequence of the programming language and methodology chosen.

This section has shown that LP is not a programming methodology outright. It is affected by the programming language used. Particularly, it is affected by the programming abstractions of a given language.

Psychological Scope Should Not Override Programming Abstractions

The following presents a poor solution to countering program language abstractions in favour of psychologically scoped chunks: although psychological scope does exist, it may be countered by obscuring the natural abstraction of the programming language with what the programmer considers the correct abstraction. The literate programming system (which allows chunk nesting) allows the programmer to embed the (abnormally scoped) chunk, say a code chunk definition equivalent to a class definition, within a chunk that represents the programmer's psychological scope. Using this technique, the programmer is able to hide the abnormally scoped chunk by using obscurity. The real chunk is encapsulated in another chunk which helps to obscure what has really happened — this is conceptually similar to using OO languages to develop in a procedural fashion.

While this is a possible solution to the problem, it is an ugly remedy that distorts the output of a literate program itself. Not only are there unnecessarily nested chunks, but the programming language constructs themselves are de-powered. LP is not an outright methodology in itself. It forms but part of the whole. It is to be used together with existing methodologies, rather than replace them. Hiding code chunks that are forcibly insisted upon by the programming language, is akin to hiding the programming language's methodology and constructs as well — it is best to use another methodology and/or language (there are many).

```

<<access file>>=
...
<<populate statistical tables>>=
...
<<compute average of data>>=
...

<<Statistics class>>=
public class Statistics {
    public int average( int[] data ) {
        <<compute average of data>>
    }

    public int median( int[] data ) {
        ...
    }
}

<<Query class>>=
public class Query {
    public void main() {
        Statistics stats = new Statistics();
        average = stats.average(array);
    }

    public int[] readFile(String file) {
        <<access file>>
        <<populate statistical tables>>
    }
}

```

Figure A.2: Psychological scoping is altered to suit the object oriented programming language.

```

<<access file>>=
...
<<populate statistical tables>>=
...
<<compute average of data>>=
public int average( int[] data ) {
    ...
}

<<Statistics class>>=
public class Statistics {
    <<compute average of data>>
}

```

Figure A.3: Psychological scoping: a second attempt.

```

<<access file>>=
...
<<populate statistical tables>>=
...
<<compute average of data>>=
...

<<Statistics class>>=
public class Statistics {
    <<Statistics::average method>>
    <<Statistics::median method>>
}

<<Statistics::average method>>=
public int average( int[] data ) {
    <<compute average of data>>
}

```

Figure A.4: Psychological scoping: a third attempt.

Gain of Power

The explicit chunking of programming language abstractions can be extremely beneficial to the literate programmer and his intended audience. The chunks that would otherwise not have been developed (the class definition chunks in Figure A.2 on page 246, for example) may be grouped together under their common encapsulating entity. That is, class definitions may be grouped together in a section of the literate document, for example, to add emphasis to the class structure of a given program. These class chunks may then be ordered and documented in a manner to illustrate the interactions between them.

A parallel can be drawn between this ability to extract programming language abstractions via the literate program chunks and the **Javadoc** tool for example. **Javadoc**, described in Section 2.9.1 on page 35 is a popular documentation tool that generates HTML based API reference for the Java programming language. Our method expresses such detail, however, in an ordered manner which would reveal and emphasise important relations between classes, for example.

A.3.3 Medium of Focus

Because it is possible for a chunk and a programming language artefact to represent same abstraction, we argue that this one-to-one mapping causes a transparency of the chunk (in Section A.3 on page 241, we term this “loss of power”). The target audience of a chunk and programming language artefact are aimed towards a different audience, however.

A chunk is an expository medium. Its purpose is to illustrate and communicate; to add direction, emphasis, and/or elaboration. Its goal is to aid comprehension. Conversely, a programming artefact, such as a method, class, or package, is founded upon encapsulation; and therefore adopt a modular ‘black box’ approach. Parnas’ modular approach to programming [63] suggested that modules communicate via interfaces; one does not need to understand how a module works in order to use it. Thus programming artefacts

*Chunks are aimed at exposition,
routines are aimed at encapsulation*

A.3.4 A Super-Language

LP tools such as **WEB**, **FunnelWeb**, and **nutweb** (discussed in Chapter 2) facilitate parameterised macros (explained Section 1.4.1 on page 11). Effectively used, these macros can enhance the implementation programming language. The best example of this rests with Knuth’s own **WEB** LP tool, which was designed to be used with the Pascal programming language. Pascal has some deficiencies as a programming language. Kernighan [41] mentions many reasons; the following is pertinent to our discussion:

- Pascal insists on the pre-declaration of labels, variables, constants, types, and routines (due to its one-pass compiler). This means that routines become the first readable components of the program source. While this may be advantageous in some circumstances, it can become a convoluting restriction. LP can overcome this by allowing the dispersion of these subroutines throughout the literate source, as seen fit.

Another example of enhancing the implementation language is the use of parameterised macros with parameterless languages such as HTML and XML. Using tools (such as **FunnelWeb**) which have multiple argument macro support allows the programmer to create *templates* of recurring output, such as headers, tables, forms, etc. The **LPML** tool (Section 3.4) uses this to good effect in its demonstration literate code.

A.3.5 LP as Program Description Language

Brown and Cordes [13] and Shum [79] use literate programming as a program description language (PDL). Although it exhibits these characteristics, we will show that as literate programming cannot be used as a complete APL for cross-interpretation to other languages.

If the programming language chosen is irrelevant and does not affect a literate program’s structure, then LP could be treated as a true abstract programming layer. The programmer could design and implement the program’s structure in a fully literate manner, with the programming language a purely arbitrary choice. No matter which language is then chosen, the programmer merely “fills in the gaps”. This is not the case, however.

Changing the programming language will affect the resulting literate program⁸. It is not only a difference of implementation methodologies (imperative or OO, for example), but also one of abstraction. Programming languages afford different levels of abstraction. For example, C is commonly considered a low-level imperative language. Perl is considered more abstract than C. Choosing between C and Perl can have a very large effect on the code implementation, thence the literate program. An example of the difference between the two languages is demonstrated by Kernighan and Pike [42] whereby a Markov chain is implemented in four different programming languages — two of these Perl and C. Whereas in C, the program may have to iterate through a linked list data structure comparing a string for equivalence, Perl offers associative arrays which allow the indexing of values with strings as their key. This not only hid the level of detail, that is, abstraction, but also changed the method of program construction.

The less abstract a programming language may be, the less abstract the low-level chunks that will contain the implementation source code. For example, a low-level programming language such as C would require chunks that deal with lower abstraction than a programming language such as Perl, which is more abstract. It would likely be the case that extra chunks would be needed to contain this lower level of abstraction. At some point however, a layer of chunks should exist which provide a basis for the pseudo-code of a program: in order to perform the Markov chain transformation, a given transformation method must be followed: an algorithm is progressively followed. However, the data units used to store the elements in the Markov chain may be language dependant. The chunks that are used to contain these abstractions are likely to be different depending on the programming language used. Therefore, although the Markov chain algorithm can be followed, the implementation will differ depending on the language. The algorithm however should be deemed largely as program language independant and therefore implementable with any programming language. This algorithm should be represented by a set of chunks.

⁸ It may even have repercussions on the implementation method, however this is outside the interests of this argument.

We have shown that LP is a tool to be used along-side programming languages, not a replacement. LP is not an abstract layer outright that can be used to substitute code chunks from one language to another.

We suggest, however, that at a particular level of abstraction, chunks will be similar amongst most languages, amongst most implementations. If one were to highlight these chunks in some manner, this would provide an interesting opportunity to provide a skeletal structure, thus generating a theme of skeletal chunks. These chunks could be implemented in the variant programming languages.

A.4 Unordered Programming Languages vs. Psychological Ordering

It is argued that literate programming is just a type of unordered code⁹, and that programming languages such as C++, allow the declaration of variables, methods, and classes wherever in the source code the programmer desires. While this is true, the code must follow a linear order for it to be comprehensible by the computer. For example, methods may not be declared and implemented outside the scope of the containing class. Literate programming allows this to occur, however. Psychological order and grammatical (un)ordering are not the same thing.

A.4.1 Are Programming Languages Literately Enabled?

Is LP an imposed phenomenon? Are programming languages able to assume the capabilities of LP tools, therefore rendering LP largely ornamental. Section 1.2.3 on page 7 lists several factors that defines literate enabled tools. One of these defining factors is the ability to create nested hierarchies of code chunks. It could be argued that programming languages provide this feature: a function can invoke another function. Can programming languages therefore be considered literately enabled?

⁹ <http://www.cs.wisc.edu/~glew/programming-languages.html>

It is possible to develop functions which are *roughly*¹⁰ able to assume the scope of a code chunk. Although certain languages such as Pascal insist on the predeclaration and definition of subroutines, and are thus unsuitable contenders for literate languages, other languages, such as Perl, do not. It could be possible to substitute code chunks for functions and then order these functions about the source code such that they read in a similar fashion to a literate program.

There are several shortcomings to this approach:

- Development of such functions contravenes what is considered to be the practice of good cohesion [60].
- Many variables will be scoped globally detrimentally affecting development, maintenance, and reuse. The proliferation of functions will involve the passing of an increased number of variables between functions; one way to overcome this is to make variables global. A dilemma ensues: do we develop tightly coupled functions, or tightly globally coupled functions. Which one of the two evils does one choose?
- There is no support for the use of multiple languages as is provided by implementation language-unspecific LP tools.

Programming languages; in particular, functions, methods, and sub-routines, are poor substitutes for the code chunk. Although LP utilises the granularity of chunks to expose semantically (and perhaps syntactically) important areas of source code, this does not stand for programming languages.

This section has shown that programming languages, even though coarsely able to emulate the literate code chunk, should not be used to do so because of the negative impact on the tangled source. We can further imply from this and add to the definition of a literate tool in Chapter 1: a literate tool should not negatively distort the computer oriented source code¹¹. Both audiences, the computer and the human should be mutually satisfied.

¹⁰ they do not possess infinite granularity

¹¹ there is an important distinction to be made here. It is possible to develop distorted source code through the misuse of the literate tool's functionality. Our concern here, however, is that the literate tool does not detract from the comprehensibility of one

A.5 Mis-direction of Focus?

Directing our programming to another human rather than at the computer directly affects the program's makeup and content. Is this a good thing? It can be argued that since the program source code is affected by programming literately, it is directed more towards the human reader, rather than the compiler. This could mean that the program source code is less efficient than it could possibly be — thereby creating an inefficient program. Is this truly the case, however?

Does the fact that the intended human audience to which the final program source code is not going to be sent to, generate an inferior program?

We could argue, rather convincingly, no. Exposition — elaboration and explanation — of a task at hand clarifies programmer intentions. Not only to the final reader, but importantly, in this case, to the compiler (or interpreter, as the case may be). It promotes full understanding of the task at hand; and elaborating on what is to be done, and then how the task will be implemented can only mean that the programmer fully understands the task at hand — and has provided a working solution to it.

The fact that literate programs are written with the human reader in mind means that the programmer elaborates on his own thoughts and, assuming that he is a good literate programmer, explains decisions, reasoning and methods encountered throughout the program. The act of thinking one's way through the meticulous details of what needs to be done before diving straight in and doing them aids not only in the clarity of code understanding, but also code efficiency and conciseness.

document for the comprehensibility of the other. It may be argued that Knuth's WEB did negatively distort the tangled code — we treat this as an exception as this was Knuth's explicit intent.

Appendix B

TBLP Methods in Software Engineering

In this Appendix, we provide some background reasoning of the need for a theme-based approach. We also consider the nature of themes when used to represent stages of the software development life cycle.

The theme-based approach asks the question: “Which part, or aspect, of the story would you like to know about?” And the child (obviously a boy) answers: “Just tell me about the exciting parts with the flying carpet. I don’t like all that silly boy-girl kissy stuff!”. The Arts student wishes however to analyse the story: “I’m sure the flying carpet has something to do with the author’s parental upbringing.”

It is apparent that multiple themes may emanate from one focal representation. In a software system, program source code is one of the many *software system* documents that may be composed by the author. User documentation may be another document. API documentation, management progress reports, and three-dimensional data visualisation perhaps another.

The executable program is considered as *part of the entire software system*; program source, testing source, install scripts, interface story-boards, APIs, various user and program documentation, requirements analysis, requirements specifications, may compose the rest of the system. We describe *holistic development* the process of developing a software system with an all-encompassing perspective, such as this, as *holistic development*. Effectively, we draw the software system boundary to capture more than just the immediate and mappable by-products of software development.

Non-Automated Themes

Interestingly, the need for multiple themes can be managed by explicit, instructional methods, as often adopted by pedagogical books. We take an example from Abrahams, Berry, and Hargreaves [91], presented in Figure B.1 on the following page. In the “Read This First” section of their book, “T_EX for the Impatient”, two sets of instructions exist, each aimed towards a reader audience of differing knowledge and ability. One group is directed to initially read the introductory sections of the book and then use the rest of the book as a reference source. The other group is instructed to use the book as a reference source. The recommended approach to either group of using the book as a reference is slightly different. Instead of writing two or more books oriented towards readers of differing ability, instructions are offered as to how the book is best read depending on the reader’s experience.

Our needs as software developers and users are somewhat more elaborate than the mere reordering of information as presented in this previous example¹. Literate programming offers only two possible orderings of chunks; the computer order and the psychological order. The reader audience however, may be vast and varied, possess different skills and abilities, different interests and queries, and possess different layout and formatting requirements.

B.0.1 A Layered Approach to Themes

Figure B.0.1 on the following page illustrates processes, in the form of layers, that may be involved in the Software Development Life Cycle (SDLC) of a software project. Each of these layers can be considered an abstraction of the one beneath it. Each layer is a possible theme candidate. Themes A, B, and C of Figure B.0.2 on page 258 are example themes that may be composed. Themes A and B are reminiscent of the traditional LP model’s ability to generate documents:

1. The tangling process of the traditional literate processing model generates the equivalent of the source code layer of the SDLC. This is

¹ This reordering, or displacement, approach is implemented as the second model in Section D.3 on page 299.

If you're new to T_EX:

- Read Sections 1–2 first.
- Look at the examples in Section 3 for things that resemble what you want to do. Look up any related commands in “Capsule summary of commands”, Section 13. Use the page references there to find the more complete descriptions of those commands and others that are similar.
- Look up unfamiliar words in “Concepts”, Section 4, using the list on the back cover of the book to find the explanation quickly.
- Experiment and explore.

If you're already familiar with T_EX, or if you're editing or otherwise modifying a T_EX document that someone else has created:

- For a quick reminder of what a command does, look in Section 13, “Capsule summary of commands”. It's alphabetised and has page references for more complete descriptions of the commands.
- Use the functional groupings of command descriptions to find those related to a particular command that you already know, or to find a command that serves a particular purpose.
- Use Section 4, “Concepts”, to get an explanation of any concept that you don't understand, or need to understand more precisely, or have forgotten. Use the list on the back cover of the book to find a concept quickly.

Figure B.1: Depending on their ability and experience, readers are advised how the book should be read.

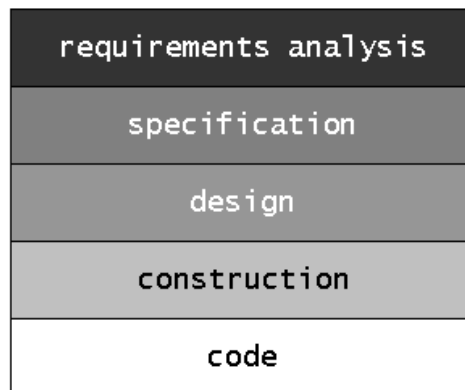


Figure B.2: The layers, or phases, of the SDLC.

expressed as theme A in Figure B.0.2.

2. The weaving process often combines the ‘code’ and ‘construction’ layers to form one theme. This is represented as theme B.

To bring literate programming into the mainstream as a plausible development choice of software engineers, an LP tool must recognise the existence of, and enable the development and representation of, all layers. In addition to themes A and B, therefore, theme C should be composable from the same web.

B.0.2 Cross-Sections of Layers

Most software development methodologies respect that user requirements may change throughout the development of a program’s life. Extreme Programming (XP) is one such methodology, and may be considered as one of the more flexible approaches to software development. The methodology accommodates new requirements being added mid-way through a system’s development, for example. This contrasts with the largely static development associated with methodologies such as the waterfall [64] process. Methodologies, such as XP, is not possible using traditional LP. The concept of user-stories driving source code development effectively requires cross-sections from multiple layers of the SDLC, and not an entire layer outright. Theme D illustrates such a cross-section. This theme is equivalent to an iteration in XP that ultimately results in the production of release code. Of course, there will be multiple iterations, and therefore cross-sections, therein.

Maintenance is another process that can affect many layers of the SDLC. Although in some circles, maintenance is considered another phase of the SDLC, it is better expressed as a pervasive process. Expressing a maintenance operation as a theme will see many chunks from many layers included.

Figure B.4 on the following page abstractly illustrates that the various layers of the SDLC can be individually chunked and relations amongst these chunks exploited to compose a theme.

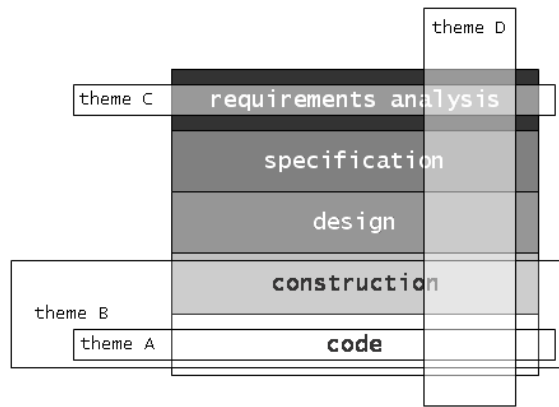


Figure B.3: Themes that can be exposed of a software system.

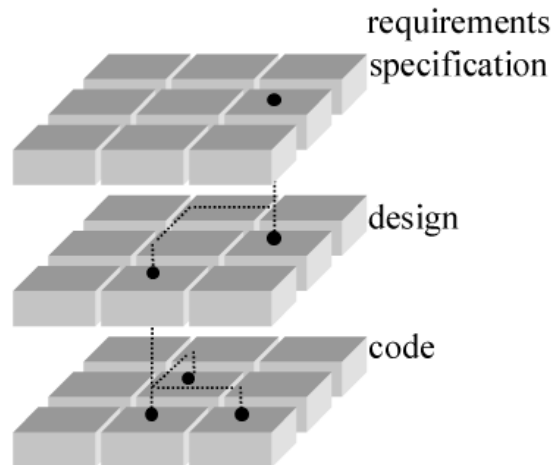


Figure B.4: Chunks from various layers are related to an abstraction that drives theme's composition..

Process Versus Result

We take care not to confuse the development process of each layer in the SDLC with the documentation requirements of that layer. The requirements documentation may not necessarily resemble the requirements elicitation process itself. This could be expressed as a separate document containing perhaps a video or audio recording of interviewed clients (should multi-media

chunks be supported).

Mapping of Layered Abstractions

Mens [51] asserts that there does not always exist a direct mapping of elements from the architectural to the code level. He highlights the need for media that allow the development of software from these immediately unmappable abstractions.

For example, it is not always possible to distinctly map a non-functional requirement to a distinct unit of code. Each is considered an abstraction, however, cannot be explicitly related in an objective manner. Requirements such as efficiency are sometimes generic enough to affect the whole system construction rather than a single module. Response requirements of a software system may be pervasive throughout all areas of code development. Effectively, in LP terms, we are unable to create a cohesive atomic chunk.

Tools such as HyperJ, an implementation language for Hyperslices, and AspectJ, a Java-based aspect oriented programming (AOP) tool (see Section 5.1.1 on page 85), assume that there is always a direct mapping between architectural components and source code.

Theme-based literate programming should facilitate both these mappable and non-mappable abstractions to be developed. We have illustrated this in Section A.3.2 on page 243, which discussed the phenomenon of psychological scope.

B.0.3 Flexibility of Approach

Key to theme-based literate programming is that it should not impose restrictions on software development methodologies. The author should be free to choose whichever programming and design methodologies he cares to use. TBLP must allow the representation of the abstractions related with any methodology utilised in the holistic development process.

The TBLP tool must support any process the author wishes to develop software and facilitate the author to reflect his thought process: one recommendation for good literate programming practice, presented in Section 8.3.8,

is to ‘program with intent’. This recommendation is based upon the reasoning that program source code is an extension of thought. People do not initially think about problem domains in terms of compiler oriented source code. The thought process that helped derive the source code can usually be more *naturally* expressed in a written language — English for our purposes². The decreasing layers of abstraction effectively iterate in expressiveness and abstraction from *thought* to the final compiler oriented source code.

B.1 A Set of Example Themes

Let us consider a simple example that incrementally captures the concept of multiple themes. Our approach is an abbreviated and informal form of software development³. Let us say that this is a stage of a refactoring exercise in XP.

There exist (arguably) five different chunk types in these examples. Each chunk is marked up in the traditional **Noweb** manner, however, unlike **Noweb**, we have marked up all chunks, including documentation chunks. The diagrams reflect chunk association throughout the example. For clarity, we do not use XML markup, and refrain from using a chunk’s ID as its reference. Instead, we resort to the traditional means of utilising a chunk’s name as its reference. A chunk’s type is displayed as part of its implementation.

Effectively, we use this example as a proof for our proposed generic chunk model in Chapter 5.

Our first theme is similar to traditional literate programs. A documentation chunk (of type **java code**) is associated with a **java code** chunk, `<<define class Expression>>` (forming an atomic chunk). This code chunk is composed of three other code chunks; `<<define Expression::display>>`, `<<define Expression::eval>>`, and `<<define Expression::check>>`. Figure 117 on page 262 depicts this scenario. The code chunks lie on the first level, whilst

² Even if it isn’t, it is usually the most widely and immediately humanly comprehensible method (which avoids the noise of human interaction — stuttering, undue pauses, inability to immediately express ideas...), due mostly to its abstract nature. This is not to say that it is the more definitive, or precise, of the two however it provides a more immediately understandable and communicative set of requirements

³ commonly referred to as a hack

the documentation chunk rests on a plane situated physically above them. Note the interconnecting arrows amongst the chunks: the fine dotted arrow shows implied association while the solid directed arrows illustrate nesting.

```
<<define class Expression  type='java source doc'>>=
The [[Expression]] class deals with evaluating, checking and
displaying a mathematical expression.
```

```
<<define class Expression  type='java code'>>=
  <<define Expression::display>>
    <<define Expression::eval>>
      <<define Expression::check>>

  <<define Expression::display  type='java code'>>=
public display() {
  ...
}
  <<define Expression::eval      type='java code'>>=
public eval() {
  ...
}
  <<define Expression::check     type='java code'>>=
public check() {
  ...
}
```

Next, modules are developed to test the developed source code.

Module testing presents another dimension of code chunks, and thus aptly receive their own chunk type; “test source”. The documentation and code chunks of the module testing theme are presented, in Figure 117 on the following page on a different dimension to that of the the code chunks of the deliverable source code. Note that the documentation chunk (chunk number 10) references the code chunk (chunk number 1) from the deliverable source

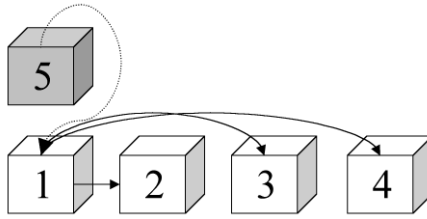


Figure B.5: Theme 1

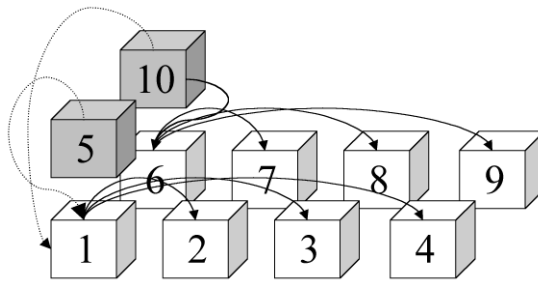


Figure B.6: Theme 2

code, and is associated with the first code chunk of the testing module (chunk number 6).

```
<<test display methods  type='test source doc'>>=
Test the [[Expression::display]] method.
```

```
<<test display methods  type='test source'>>=
  <<test no input>>
  <<test maximal characters>>
  <<test nonsense characters>>
```

Once tested and passed, this software is deposited into the baseline source. The baseline source is another theme.

The baseline (a repository of functional, tested, quality code), to which each code unit must pass a series of tests to be accepted, is deposited to the baseline. This forms another orthogonal view of the literate source. The

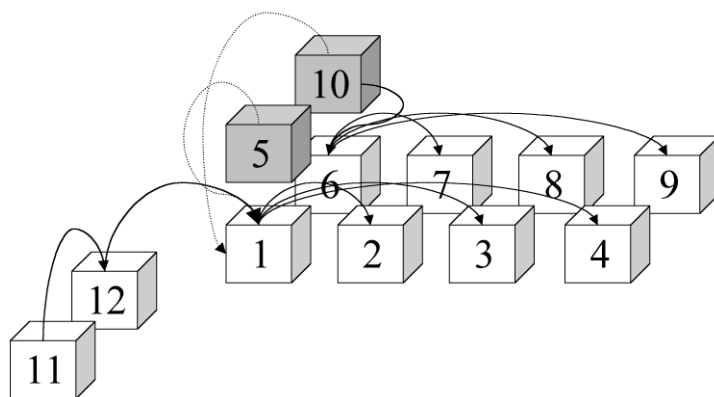


Figure B.7: Theme 3

“passed” chunks are integrated with the `<<baseline update 2002 06 09 01:09:41>>` (chunk number 12 in Figure 117) theme, which itself, exists as part of the overall `<<baseline>>` theme.

```
<<baseline type='baseline'>>=
...
<<baseline update 2002 06 09 01:09:41>>
...

<<baseline update 2002 06 09 01:09:41 type='baseline'>>=
<<define class Expression>>
```

API documentation is also be developed for the tested module; documentation equivalent with the expository power of **Javadoc** for example.

The API chunks (`<<class Expression>>`, `<<Expression::display>>`, `<<Expression::eval>>`, `<<Expression::check>>` and chunk numbers 13, 14, 15, 16 respectively) are situated on a different level, or dimension, to other chunks. Each API chunk references a code chunk which contains a method definition, upon which it expounds as to its possible use. This is abstractly illustrated in Figure 117 on the following page by directed arrow from chunk number 14 to chunk number 2, for example. This example illustrates the

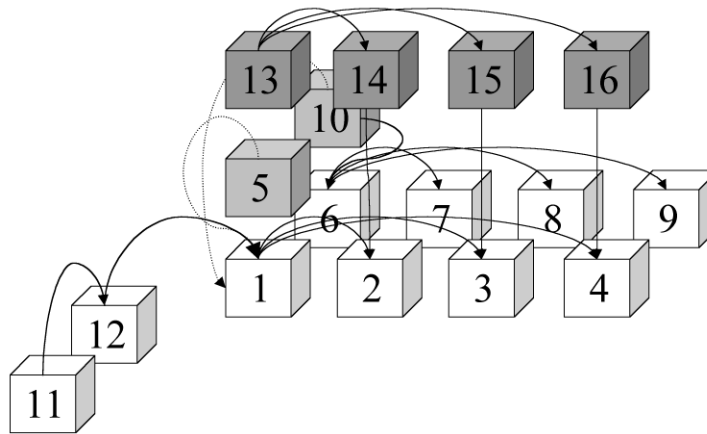


Figure B.8: Theme 4

concept of the API theme as a meta-theme.

Note that the API code chunks share the same name as their source code counterparts. They are differentiated by an implied `chunkID`, however.

```
<<class Expression  type='java API doc'>>=
  API for Expression class
  contains the methods:  that return:  that accept arguments:
  <<Expression::display>>
  <<Expression::eval>>
  <<Expression::check>>

<<Expression::display  type='java API doc'>>=
  <<define Expression::display>>

<<Expression::eval API  type='java API doc'>>=
  <<define Expression::eval>>

<<Expression::check API  type='java API doc'>>=
  <<define Expression::check>>
```

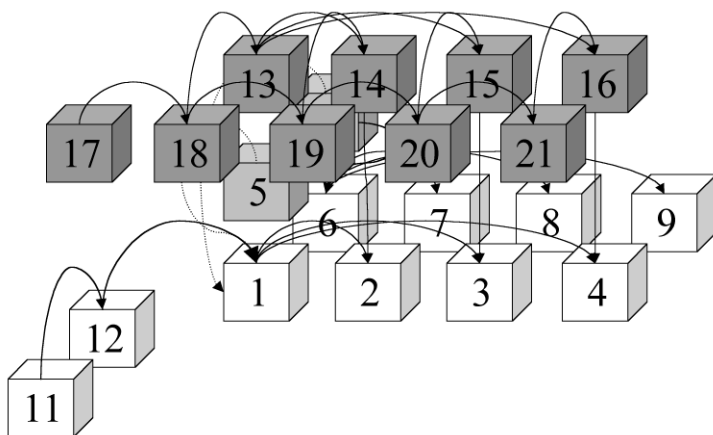


Figure B.9: Theme 5

The API documentation may not satisfy all programmers inquisitions. Some may need more didactic instructions; a set of examples which extend from the API and illustrate how it is practically used would be useful.

This is presented as another meta-theme, as depicted in Figure 117. A separate didactic chunk is developed to elaborate on how the class and methods may be utilised.

```
<<instantiate a new Expression object  type='java tut'>>=
To instantiate an Expression object...
```

```
<<class Expression API>>
...
```

```
<<output result to interface  type='java tut'>>=
To display results, use the
<<Expression::display API>>
...
```

It is wished to show from which part of the requirements specification document the `display` method of the `Expression` class emanated.

Figure 117 on the following page shows chunk number 22 as the chunk that influenced the implementation of the `display` method. `<<results must`

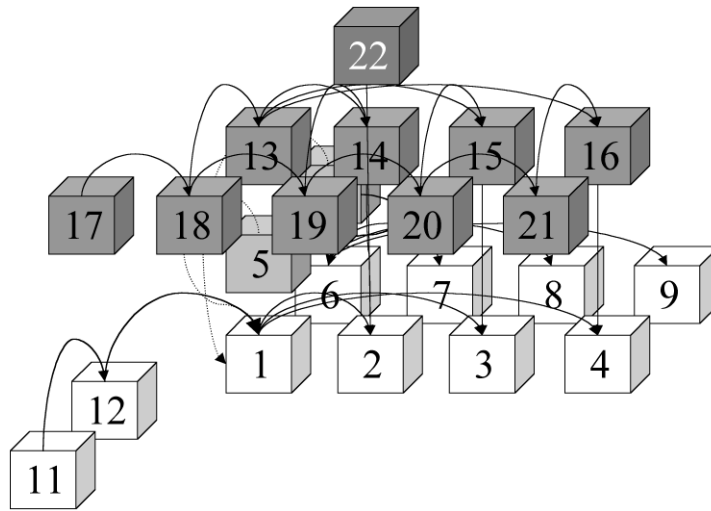


Figure B.10: Theme 6

be displayed immediately>> has been extracted from the requirements specification document (which would exist in an orthogonal theme).

```
<<results must be displayed immediately type='req spec'>>=
It is a requirement that...
<<define Expression::display>>
```

It is necessary for the clients, to whom the software is to be delivered, view how their requirements have been implemented. In an XP scenario, a user story has been developed, and the user is shown excerpts of the working and tested source code.

As part of the requirements specification document, chunk <<results must be displayed immediately>>'s development has been influenced by the <<immediate feedback given to user>> user story. This user story forms the (dark grey) user story theme — one of many. Note how the user story theme is composed of chunks from multiple themes The extraction of chunk numbers 22 and 2 specifically illustrates the development of a cross-sectional theme.

```
<<immediate feedback given to user type='user story'>>=
user story: results must be displayed immediately
```

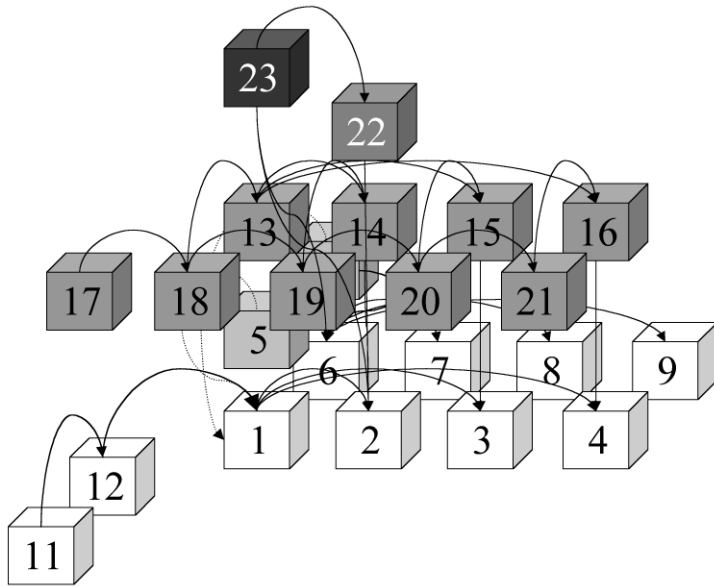


Figure B.11:

```

specific requirements
  <<results must be displayed immediately>>
implemented by
  <<define Expression::display>>
tested with
  <<test display methods>>

```

User documentation for the program is to show how to display an expression in the graphical interface.

need user doc here that links to one of the layers

We have illustrated the use and development of multi-dimensional themes. Even with such a relatively simple, contrived, and deliberately shortened example, it is possible to see that multiple themes may emanate from this type of holistic software development process. We have shown that there may exist many different orders of orthogonality and that there is no one dominant theme — this may not always be the case.

Each theme is aimed towards a specific reader audience. And each of these themes are natural views which emanate from the software system.

Appendix C

Example Literate Programs

C.0.1 Example NowebProgram

C.1 An example of noweb

The following short program illustrates the use of **noweb**, a low-tech tool for literate programming. The purpose of the program is to provide a basis for comparing **WEB** and **noweb**. The notable differences are:

- When displaying source code, **noweb** uses different typography. In particular, **WEB** makes good use of multiple fonts and the ability to typeset mathematics, and it may use mathematical symbols in place of C symbols (e.g. “ \wedge ” for “`&&`”). **noweb** uses a single fixed-width font for code.
- **noweb** can work with L^AT_EX, and I have used L^AT_EX in this example.
- **noweb** has no numbered “sections.” When numbers are needed for cross-referencing, **noweb** uses page numbers. If two or more chunks appear on a page, for example, page 24, they are distinguished by appending a letter to the page number, for example, 24a or 24b.
- **noweb** has no special support for macros. In the sample program, I have used the chunk “*Definitions 272a*” to hold macro definitions.
- **noweb** does not recognize C identifier definitions automatically, so I had to add a list of defined identifiers to each code chunk. Because **noweb** is language-independent, it must use a heuristic to find uses of identifiers. This heuristic can be fooled into finding false “uses” in comments or string literals, such as the use of **status** in chunk 272a.
- The **CWEB** version of this program has semicolons following most uses of $\langle \dots \rangle$. **WEB** needs the semicolon or its equivalent to make its prettyprinting come out right. Because it does not attempt prettyprinting, **noweb** needs no semicolons.
- Both **WEB** and **noweb** write chunk cross-reference information in footnote font below each code chunk, for example, “” Unlike **WEB**, **noweb** also

includes cross-reference information for identifiers, for example, “Defines `file_count`, never used.” This information is generated using the `@ %def` markings in the `noweb` source.

C.1.1 Counting Words

This example, based on a program by Silvio Levy and D. E. Knuth [48], presents the “word count” program from UNIX, rewritten in `noweb` to demonstrate literate programming using `noweb`. The level of detail in this document is intentionally high, for didactic purposes; many of the things spelled out here don’t need to be explained in other programs.

The purpose of `wc` is to count lines, words, and/or characters in a list of files. The number of lines in a file is the number of newline characters it contains. The number of characters is the file length in bytes. A “word” is a maximal sequence of consecutive characters other than newline, space, or tab, containing at least one visible ASCII code. (We assume that the standard ASCII code is in use.)

Most literate C programs share a common structure. It’s probably a good idea to state the overall structure explicitly at the outset, even though the various parts could all be introduced in chunks named $\langle * \rangle$ if we wanted to add them piecemeal.

Here, then, is an overview of the file `wc.c` that is defined by the `noweb` program `wc.nw`:

271a $\langle * 271a \rangle \equiv$
 $\langle \textit{Header files to include} 271b \rangle$
 $\langle \textit{Definitions} 272a \rangle$
 $\langle \textit{Global variables} 272b \rangle$
 $\langle \textit{Functions} 278b \rangle$
 $\langle \textit{The main program} 272c \rangle$

We must include the standard I/O definitions, since we want to send formatted output to `stdout` and `stderr`.

271b $\langle \textit{Header files to include} 271b \rangle \equiv$ (271a)
`#include <stdio.h>`

The `status` variable will tell the operating system if the run was successful or not, and `prog_name` is used in case there's an error message to be printed.

272a *⟨Definitions 272a⟩*≡ (271a) 274c▷

```
#define OK                0
    /* status code for successful run */
#define usage_error      1
    /* status code for improper syntax */
#define cannot_open_file 2
    /* status code for file access error */
```

272b *⟨Global variables 272b⟩*≡ (271a) 276b▷

```
int status = OK;
    /* exit status of command, initially OK */
char *prog_name;
    /* who we are */
```

Now we come to the general layout of the main function.

272c *⟨The main program 272c⟩*≡ (271a)

```
main('argc, 'argv)
    int argc;
        /* number of arguments on UNIX command line */
    char **argv;
        /* the arguments, an array of strings */
{
    ⟨Variables local to main 273a⟩
    prog_name = argv[0];
    ⟨Set up option selection 273b⟩
    ⟨Process all the files 274a⟩
    ⟨Print the grand totals if there were multiple files 277d⟩
    exit(status);
}
```

If the first argument begins with a '-', the user is choosing the desired counts and specifying the order in which they should be displayed. Each selection is given by the initial character (lines, words, or characters). For example, '-cl' would cause just the number of characters and the number of lines to be printed, in that order.

We do not process this string now; we simply remember where it is. It will be used to control the formatting at output time.

273a \langle *Variables local to main* 273a $\rangle \equiv$ (272c) 274b \triangleright

```
int file_count;
/* how many files there are */
char *which;
/* which counts to print */
```

273b \langle *Set up option selection* 273b $\rangle \equiv$ (272c)

```
which = "lwc";
/* if no option is given, print 3 values */
if (argc > 1 && *argv[1] == '-') {
    which = argv[1] + 1;
    argc--;
    argv++;
}
file_count = argc - 1;
```

Now we scan the remaining arguments and try to open a file, if possible. The file is processed and its statistics are given. We use a `do ... while` loop because we should read from the standard input if no file name is given.

```
274a  <Process all the files 274a>≡ (272c)
      argc--;
      do {
        <If a file is given, try to open *(++argv); continue if unsuccessful 275a>
        <Initialize pointers and counters 276a>
        <Scan file 276c>
        <Write statistics for file 277b>
        <Close file 275b>
        <Update grand totals 277c>
        /* even if there is only one file */
      } while (--argc > 0);
```

Here's the code to open the file. A special trick allows us to handle input from `stdin` when no name is given. Recall that the file descriptor to `stdin` is 0; that's what we use as the default initial value.

```
274b  <Variables local to main 273a>+≡ (272c) <273a 275d>
      int 'fd = 0;
      /* file descriptor, initialized to stdin */

274c  <Definitions 272a>+≡ (271a) <272a 275c>
      #define READ_ONLY 0
      /* read access code for system open */
```

275a \langle *If a file is given, try to open $*(++argv)$; continue if unsuccessful* 275a $\rangle \equiv$ (274a)

```

    if (file_count > 0
    && (fd = open(*(++argv), READ_ONLY)) < 0) {
        fprintf(stderr,
            "%s: cannot open file %s\n",
            prog_name, *argv);
        status |= cannot_open_file;
        file_count--;
        continue;
    }

```

275b \langle *Close file* 275b $\rangle \equiv$ (274a)

```

    close(fd);

```

We will do some homemade buffering in order to speed things up: Characters will be read into the `buffer` array before we process them. To do this we set up appropriate pointers and counters.

275c \langle *Definitions* 272a $\rangle + \equiv$ (271a) \triangleleft 274c 278a \triangleright

```

#define buf_size BUFSIZ
    /* stdio.h BUFSIZ chosen for efficiency */

```

275d \langle *Variables local to main* 273a $\rangle + \equiv$ (272c) \triangleleft 274b

```

char buffer[buf_size];
    /* we read the input into this array */
register char *ptr;
    /* first unprocessed character in buffer */
register char *buf_end;
    /* the first unused position in buffer */
register int c;
    /* current char, or # of chars just read */
int in_word;
    /* are we within a word? */
long word_count, line_count, char_count;
    /* # of words, lines, and chars so far */

```

276a \langle *Initialize pointers and counters 276a* $\rangle \equiv$ (274a)

```
ptr = buf_end = buffer;
line_count = word_count = char_count = 0;
in_word = 0;
```

The grand totals must be initialized to zero at the beginning of the program. If we made these variables local to `main`, we would have to do this initialization explicitly; however, C's globals are automatically zeroed. (Or rather, "statically zeroed.") (Get it?)

276b \langle *Global variables 272b* $\rangle + \equiv$ (271a) \triangleleft 272b

```
long tot_word_count, tot_line_count,
    tot_char_count;
/* total number of words, lines, chars */
```

The present chunk, which does the counting that is `wc`'s *raison d'être*, was actually one of the simplest to write. We look at each character and change state if it begins or ends a word.

276c \langle *Scan file 276c* $\rangle \equiv$ (274a)

```
while (1) {
     $\langle$ Fill buffer if it is empty; break at end of file 277a $\rangle$ 
    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 */
}
```

Buffered I/O allows us to count the number of characters almost for free.

```
277a  <Fill buffer if it is empty; break at end of file 277a>≡ (276c)
      if (ptr >= buf_end) {
          ptr = buffer;
          c = read(fd, ptr, buf_size);
          if (c <= 0) break;
          char_count += c;
          buf_end = buffer + c;
      }
```

It's convenient to output the statistics by defining a new function `wc_print`; then the same function can be used for the totals. Additionally we must decide here if we know the name of the file we have processed or if it was just `stdin`.

```
277b  <Write statistics for file 277b>≡ (274a)
      wc_print(which, char_count, word_count,
               line_count);
      if (file_count)
          printf(" %s\n", *argv); /* not stdin */
      else
          printf("\n");           /* stdin */
```

```
277c  <Update grand totals 277c>≡ (274a)
      tot_line_count += line_count;
      tot_word_count += word_count;
      tot_char_count += char_count;
```

We might as well improve a bit on UNIX's `wc` by displaying the number of files too.

```
277d  <Print the grand totals if there were multiple files 277d>≡ (272c)
      if (file_count > 1) {
          wc_print(which, tot_char_count,
                   tot_word_count, tot_line_count);
          printf(" total in %d files\n", file_count);
      }
```

Here now is the function that prints the values according to the specified options. The calling routine is supposed to supply a newline. If an invalid option character is found we inform the user about proper usage of the command. Counts are printed in 8-digit fields so that they will line up in columns.

278a *⟨Definitions 272a⟩*+≡ (271a) <275c

```
#define print_count(n) printf("%8ld", n)
```

278b *⟨Functions 278b⟩*≡ (271a)

```
wc_print(which, char_count, word_count, line_count)
    char *which; /* which counts to print */
    long char_count, word_count, line_count;
    /* given totals */
{
    while (*which)
        switch (*which++) {
            case 'l': print_count(line_count);
                       break;
            case 'w': print_count(word_count);
                       break;
            case 'c': print_count(char_count);
                       break;
            default:
                if ((status & usage_error) == 0) {
                    fprintf(stderr,
                        "\nUsage: %s [-lwc] [filename ...]\n",
                        prog_name);
                    status |= usage_error;
                }
        }
}
```

Incidentally, a test of this program against the system `wc` command on a SPARCstation showed that the “official” `wc` was slightly slower. Furthermore, although that `wc` gave an appropriate error message for the options ‘-abc’, it made no complaints about the options ‘-labc’! Dare we suggest that the system routine might have been better if its programmer had used a more literate approach?

List of code chunks

This list is generated automatically. The numeral is that of the first definition of the chunk.

*< * 271a>*
< Close file 275b>
< Definitions 272a>
< Fill buffer if it is empty; break at end of file 277a>
< Functions 278b>
< Global variables 272b>
< Header files to include 271b>
*< If a file is given, try to open *(++argv); continue if unsuccessful 275a>*
< Initialize pointers and counters 276a>
< Print the grand totals if there were multiple files 277d>
< Process all the files 274a>
< Scan file 276c>
< Set up option selection 273b>
< The main program 272c>
< Update grand totals 277c>
< Variables local to main 273a>
< Write statistics for file 277b>
*< * 285a>*
< define class Car 285b>
< define class Driver 286a>
< define method beginExcursion 286b>
< define method startEngine 285d>
< define method startEngine overload 285e>
< define variables for Car 285c>
< define variables for Driver (never defined)>
*< * 314>*
< define sub bubblesort 315>
< define sub bubbletest 319>
< gather comparisons data 320b>

<gather swap data 320a>
 <initialise variables 318>
 <iterate backwards over array 317b>
 <iterate forwards over array up to current backwards position 317c>
 <output results 316b>
 <populate partially sorted array 320f>
 <populate randomly sorted array 320d>
 <populate reverse sorted array 320e>
 <populate sorted array 320c>
 <swap adjacent members in array if necessary 317a>
 <test bubblesort 316a>
 <use packages 320g>
 <* 328>
 <define subroutines 330c>
 <define variables 333b>
 <discard cross references 329a>
 <find code section beginning 329c>
 <find doc section beginning 329b>
 <include code section 332b>
 <include doc section 332c>
 <keep track of previous line 330b>
 <output first line of input 334b>
 <process initial input 332e>
 <read themes from file 333c>
 <scan line for theme attributes 330a>
 <set up operating environment 332d>
 <sub println 331b>
 <sub process_code 331a>
 <sub process_doc 330d>
 <sub readln 331c>
 <sub read_themes_file 332a>
 <transfer STDIN to STDIN 333a>
 <use packages 334a>

Index

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

`startEngine`: 285d, 285e, 286b

`bubblesort`: 315, 316b, 319

`$i`: 317b, 317c, 318

`$j`: 317a, 317c, 318

`$ncomp`: 316b, 318, 320b

`$nswap`: 316b, 318, 320a

C.2 *Personal Greeter*

This program prompts for the user's name, then prints a personalised greeting. The process of greeting a user by name has several steps. The variable `buffer`, defined in `[[<<greet declarations>>]]`, is used by the process.

```
<<greet process>>=
<<ask what the user's name is>>
<<read name from input>>
<<print greeting>>
@
```

The `\texttt{buffer}` variable is declared as a character array. The size of the buffer is taken from the `\texttt{BUFFER_SIZE}` macro so that it can be changed easily.

```
<<greet declarations>>=
#define BUFFER_SIZE 128
char buffer[BUFFER_SIZE];
```

```
@ %def buffer BUFFER_SIZE
```

@ The `printf` function is used to print a prompt for the user's name. In order for this prompt to appear, the `\texttt{stdout}` stream must be flushed after printing. This could also have been achieved by printing a newline character (in which case the output will flush automatically). I believe it is better that the user's input appear on the same line as the prompt.

```
<<ask what the user's name is>>=
printf("Please enter your name: "); fflush(stdout);
```

@ The `\texttt{fgets}` function is used to read the user's name. `\texttt{fgets}` was chosen because `\texttt{gets}` does not check for buffer overflows.

```
<<read name from input>>=
```

```
fgets(buffer, BUFFER_SIZE, stdin);
```

@ The greeting is printed using the printf function again, this time with a `\texttt{\%s}` format to insert the name.

```
<<print greeting>>=  
printf("Hello %s", buffer);
```

@

The root chunk wraps the greeting routine in the C `\texttt{main}` function so that it will run when the program is executed.

```
<<*>>=  
#include <stdio.h>  
int main(void) {  
    <<greet declarations>>  
    <<greet process>>  
    return 0;  
}
```

C.3 Scoping (In)capabilities

285a $\langle * 285a \rangle \equiv$

285b $\langle \text{define class Car } 285b \rangle \equiv$
 `public class Car {`
 $\langle \text{define variables for Car } 285c \rangle$
 $\langle \text{define method startEngine } 285d \rangle$
 $\langle \text{define method startEngine overload } 285e \rangle$
 `}`

285c $\langle \text{define variables for Car } 285c \rangle \equiv$ (285b)
 `int i = 0;`
 `...`

285d $\langle \text{define method startEngine } 285d \rangle \equiv$ (285b)
 `int public startEngine (Jump jump) {`
 `...`

 `}`

Defines:

`startEngine`, used in chunk 286b.

285e $\langle \text{define method startEngine overload } 285e \rangle \equiv$ (285b)
 `int public startEngine (Ignite ignite) {`
 `...`

 `}`

Defines:

`startEngine`, used in chunk 286b.

286a $\langle \text{define class Driver 286a} \rangle \equiv$

```
public class Driver {  
     $\langle \text{define variables for Driver (never defined)} \rangle$   
     $\langle \text{define method beginExcursion 286b} \rangle$   
}
```

286b $\langle \text{define method beginExcursion 286b} \rangle \equiv$ (286a)

```
int public beginExcursion () {  
    car = new Car;  
    ...  
  
    if ( car.isBatteryFlat ) {  
        car.startEngine( jump );  
    } else {  
        car.startEngine( ignite );  
    }  
  
    if ( ! car.startEngine( ignite ) ) {  
        car.startEngine( jump );  
    }  
  
}
```

Uses `startEngine` 285d 285e.

C.4 Identifier Cross-Referencing (nuweb)

"test.pl" 1a ≡

```
⟨ first 1b ⟩  
⟨ scnd 1c ⟩  
⟨ third 1d ⟩  
◇
```

⟨ first 1b ⟩ ≡

```
my $var = 0;  
◇
```

Macro referenced in 1a.

⟨ scnd 1c ⟩ ≡

```
print $var++;  
◇
```

Macro referenced in 1a.

Notice that in this chunk, `var` is not found by the identifier cross-referencing heuristic. The index at the bottom of this page does not contain chunk 1d.

⟨ third 1d ⟩ ≡

```
++$var;  
◇
```

Macro referenced in 1a.

"test.pl" Defined by 1a.

⟨ first 1b ⟩ Referenced in 1a.

⟨ scnd 1c ⟩ Referenced in 1a.

⟨ third 1d ⟩ Referenced in 1a.

`$var`: 1b, 1c.

Appendix D

Theme Enabling Literate Tools

In this appendix, we illustrate that it is impossible to make existing literate tools literate-aware. This appendix represents some of our initial research that lead to the development of the generic chunk model (as described in Chapter 5).

Although the **Leo** literate programming tool supports much of this functionality, it was our intent to discover LP’s model shortcomings through experimentation. Doing so has enabled us to highlight some of the important limitations, and improvements thereof, of the traditional literate model.

The following sections illustrate the new literate model’s development process. Throughout the new LP model’s process of discovery, the solution was vividly represented as a theme-enabling technology. And although we knew what we wanted to do, we weren’t certain of how elegantly achieve it. It should be noted that others have tried (see Section 5.1.1), and although the results are good, these attempts are still apply a limiting model to their solution.

In Sections D.2.1 and D.3 ,we describe the physical implementation of the ‘switching’ and ‘displacement’ models, respectively.

D.1 Supporting Multi-Themed Requirements with a Traditional Literate Tool

D.1.1 The Journey to Enlightenment

The goals behind the development of an adequate theme-based model were as follows:

Language neutral: the literate tool should be language-unspecific, for both

the documentation and code chunks. Any markup language may be used to markup documentation type chunks (XML is recommended). Any programming language may be used in code type chunks.

Other solutions to enhancing literate programming, or presenting multiple concerns of programs largely finalise their solution with a language specific tool; the Hyper/J and AspectJ tools are an example. Although language neutral, a physical form of our model would provide a framework to allow development using any (or, indeed, many) of these tools.

Methodology neutral: we maintain the benefit of literate programming and do not impose a particular methodology of development upon the author. In fact, any number of methodologies can be used throughout the process of development.

Lossless improvement: we wish to, at the very least, provide a solution that is no worse¹ (but hopefully better) than the current one.

Unobtrusive and unimposing: the model and its implementation does not alter any of the tangled, or woven, documents' chunk content. Knuth's WEB alters the tangled source. The AOP tool AspectJ is another example where the source code is often altered through the process of converting aspect files to source. Our solution should not alter the chunk content in any way, unless directed by the author.

D.1.2 Noweb as a Development Platform

Noweb provides a good platform for model enhancement:

- Noweb has a simple syntax and grammar.
- importantly, it is able to be altered via its pipe-lined, filtering architecture.

By developing filters, we are able to implement, and alter, Noweb's weaving and tangling of literate programs.

¹ not that traditional LP is bad, of course

There is a reasonable limit to the amount of model enhancement that can be performed through **Noweb**'s filtering mechanism. We endeavour to find this limit and then explore the usefulness and the implications of implementing a more enhanced model.

D.1.3 The Bubblesort Theme-Set

Our goal is to implement a system based on thematic views. We begin with a simple enhancement to the **Noweb** tool which is derived from the *requirements* of exposition of the three **bubblesort** examples contained in Appendices D.5 to D.7. These examples implement and also test the efficiency of the bubblesort algorithm. Bubblesort is the algorithm of choice due to its simplistic, and rather short, implementation.

Specifically, the bubblesort themes are:

1. The bubblesort algorithm.
2. An evaluation of the bubblesort algorithm: this includes the implementation of the algorithm and the testing of the algorithm; the code to implement the test, and the results.
3. *Only* an evaluation of bubblesort.

The first theme is a complete implementation of the bubblesort literate program. The next two themes represent excerpts of this program. Themes two and three are effectively subsets of theme 1.

Appendix D.5 contains the complete **bubblesort** literate program (theme 2). Appendices D.6 and D.7 represent the algorithm (theme 1) and evaluation (theme 3) bubblesort themes, respectively.

It is important to note is that *the woven document is the only medium that has changed*. All documents emanate from the same web. The tangled document (source code) is not altered. The weaving of themes 2 and 3 is performed on the same web as theme 1.

Figure D.1 on page 349 illustrates an abstract view of the three themes. It shows documentation and code sections of three themes, 1, 2 and 3. All

code section names have been omitted and replaced by consecutively numbered “code” strings e.g., `code1`, `code2`, and so on. They are physically represented as ellipses. Document sections are also consecutively numbered and represented as rectangles.

All three diagrams show all chunks contained in the literate program. A shaded node denotes that the code or document section is presented for weaving in the woven document. A non-shaded node denotes that it is not present in the woven document.

Theme 2 is representative of a traditional literate program. It shows that all document and code sections are required to be output. Theme 1 represents the **algorithm** theme — theme 1. It shows that only a select number of chunks are desired to be output in the weaved document, namely document sections 2, 3, 6, 7, and 8, and code sections and 2, 5, 6, and 7. Theme 3 represents the **evaluation** theme and requires code sections and their associated documentation to be output, namely document sections 1, 2, 4, 10, 13, 14, 15, and 16, and code sections 1, 2, 3, 9, 12, 13, 14, and 15.

The reader should note that all three themes contain the same base set of chunks in the same linear order. Our concern in this experiment is to enable the literate tool to elide a chunk, or conversely include a chunk in the woven document.

Note that there are two document sections (`doc2` and `doc3`) associated with code section `code2`. `code2` represents in our literate program example, chunk `<<define sub bubblesort>>`. Theme 1 includes both document sections in its woven output, whereas theme 3 includes document section `doc2` only. Enabling this functionality in a literate programming environment is a basic requirement of theme based weaving; *multiple documentation chunks may be associated with a code chunk*.

D.2 Requirements for an Alteration to Noweb

In order to generate a number of differently themed documents, each derived from a single web, or master document², we must enforce the following

²The master document is the literate program which serves to contain all code and documentation sections; in essence a *repository* of chunks. From this master document

rules via a **Noweb** filter. Specifically, derived from the themed examples, as depicted in Figure D.1, for each theme:

- associate a given documentation chunk with the immediately following code chunk,
- do not associate a given documentation chunk with the following code chunk, and
- do not output a code chunk unless its related documentation chunk is output.

These rules assume that:

Documentation and code chunks are one atomic unit: a chunk is made up of a document section and a code section.

Multiple documentation/code chunk association: More than one documentation chunk may be associated with one code chunk allows a code chunk to be represented by any number of documentation chunks. Each of these documentation chunks may be employed by different themes. For example, consider the excerpt from the **Noweb** source of the **bubblesort** (Appendix D.5) literate program which appears in Program Listing 1 on the facing page.

This excerpt shows two document sections associated with the code chunk `<<define sub bubblesort>>`. The first line of each document section contains the names of the themes that the section should be included in. The first chunk is to be included in the **algorithm** and **evaluation** themes, while the second document section is only to be included in the **algorithm** theme.

The choice of syntax used to markup themes is explained in Section D.2.1 on page 294.

are the relevant documentation and code sections extracted to form the literate themed document

@ evaluation algorithm

Bubblesort makes multiple scans through the array to be sorted, swapping adjacent pairs of elements if they re in the wrong order, until no more swaps are necessary. The bubble is the element propagating through the array.

@ algorithm

This bubblesort routine requires, as an argument, a reference to an array of unsorted numbers. The array after being sorted, is then available outside the function.

e.g.,

```
\begin{verbatim}
  @array = (1, 5, 9, 7);
  bubblesort(\@array);
  print "@array";
\end{verbatim}
```

```
<<define sub bubblesort>>=
```

```
...
```

```
@ %def bubblesort
```

Program Listing 1:

Documentation is a pre-emptive process: documentation precedes the code chunk that it is related to. This is a rule that is commonly assumed, but not enforced, nor necessarily always adhered to. The inability, therefore, to link documentation chunks with preceding code chunks, is an implication of this rule.

Essentially, we wish to implement a mechanism with which to switch chunks' inclusion into the woven document either 'on' or 'off'. It is a matter of choosing which chunks in particular are to be presented in a literate themed document. In order to create a logically flowing literate document, clearly, not all chunks should be woven. This is because not all chunks are logically linked to the theme being developed.

*D.2.1 Model Enhancement of **Noweb** — The Initial Attempt*

From these specifications, we are able to form a representational model.

Noweb's filtering mechanism provides the ability to implement these requirement specifications. A **Noweb** filter is often written as a script that receives as piped input, the **Noweb** source emitted by the **noweave** tool before this code is finally transformed into the representative mark up language.

The filter that implements this enhanced model must determine which documentation and code chunks should be woven given a specified theme. Essentially, a filter must be developed that implements a switching mechanism; a documentation chunk may be included or not; if it is, the associated code chunk is also included.

Noweb, like most other literate tools, doesn't provide a method with which to differentiate among documentation chunks. This is because the documentation chunk, traditionally, is associated to one global output theme i.e., the single-themed weave. We must therefore develop a mechanism to allow a documentation chunk, and its associated code chunk, to be woven in a given theme, yet prevent documentation chunks not intended for the current theme from being woven. We adopt the approach in Program Listing 1 on the preceding page.

- documentation sections in **Noweb** start with the two characters, '@_'.

The documentation section is delimited by the beginning of a new code chunk e.g, `<<*>>=`.

- the themes that a documentation chunk is to be included in are placed on the same line as the `@_` code chunk terminator.

A more robust attempt at developing such a theme based tool would require some markup to assign a documentation chunk with a code chunk. For example `@_%themes`, followed by a list of pertinent themes, could be used in a similar manner as `@ %def` is used to list identifiers. For clarity of presentation and simplicity of implementation, we have opted to exclude such a feature. This would avoid the theme markup appearing on the first line of each documentation chunk.

We look to XML as a markup language for such purposes in the development of our theme-based literate tool in Chapter 6. We also make mention here that a GUI based approach to assigning chunks to themes would greatly alleviate the laborious efforts associated with manual markup.

Following is a theme-based excerpt from the **Noweb** source developed to implement the requirements³ from Section D.2 on page 291. It is derived from the requirement list presented in Figure D.2 on page 350:

³ a full version of the `themenoweb` filter can be found in Appendix D.8

To develop a filter for noweb that implements this ruleset is not a particularly formidable task. In fact, given the right language for the task, it is a rather simple process.

Using Perl as the programming language, and possessing knowledge of the noweb filtering source code file, we can implement an algorithm to process the noweb input:

We loop and process input from STDIN; line by line. We then scan each line to find documentation sections with the relevant theme attributes. If these theme attributes are present we carry on and include the documentation section, and its associated code section in the output. Otherwise, if the attributes are not present, we simply exclude the source from output to STDOUT.

```
<*>≡
  <set up operating environment>
  <process initial input>

while ($line = &readln) {
  if ( <find doc section beginning> ) {
    <keep track of previous line>
    for my $theme (@themes) {
      if ( <scan line for theme attributes> ) {
        &println($prev_line);
        <include doc section>
        $INCLUDE = 1;
        last;          # include doc chunk once only
      }
    }
  }

  if ( <find code section beginning> && $INCLUDE == 1) {
    &println($line);
    <include code section>
  }
}
```

```

        $INCLUDE = 0;
    }

}

```

⟨define subroutines⟩

One disadvantageous repercussion from performing noweb theme based literate programming. All indexing and cross-referencing capabilities are lost.

We purposely exclude all cross references to chunks and definitions because, unfortunately, it is cumbersome to determine which chunks will be included in the L^AT_EX output. Cross-references are excluded via the `process_doc` and `process_code` functions.

To do this would require processing the entire input and keeping state which chunks have been included. We would then be required to reprocess the input and include cross-references to the chunks to that are included in the output document. This process can most definitely be done, however, it is rather ugly.

⟨discard cross references⟩≡

```

    next if $line =~ /$xref/;

```

D.2.2 A Summary of the Initial Implementation

We have successfully implemented a theme-based “switching” LP model. This model allows chunks to either be included or excluded from woven document. It has been implemented using **Noweb**’s pipe-lined filtering mechanism. Utilising this filter, a programmer may perform the following operations:

- include a documentation chunk in the woven output.
- include a code chunk associated with the documentation chunk in the woven output.
- associate multiple documentation chunks with a code chunk.
- exclude a documentation chunk from the weaved output.
- exclude a code chunk from the weaved output (if all associated document sections are excluded).

We recognise the cross-referencing of chunks as an important feature of literate programming. Given the prototypical nature of the implementation of this initial model, we have not included the ability to create indices and cross-references. Submitting this filter to the public domain would necessitate such functionality.

A side-effect of requiring documentation chunks to have pre-declared themes they are to be presented in, is that a documentation chunk which does not appear in any themes may therefore be used as a comment chunk — something not elegantly possible with all literate tools. Perhaps using “comment” as a theme type may clarify the documentation chunk’s use and avoid confusion.

Effectively, we have implemented a mechanism with which to switch chunks’ inclusion in the woven document either ‘on’ or ‘off’. The controlling node in this case is the documentation section. By including this section in a theme, we tell the **Noweb** filter to include its accompanying code section as well.

Criticisms of the switching model’s implementation are:

- The theme markup is coupled with the chunk.
- A code chunk’s inclusion in the document is dependant on a documentation chunk.

D.3 Initial Model++: The Displacement Model

The **Noweb** filter that implements the initial model does not facilitate chunk displacement — chunks may not be reordered depending on theme requirements. It is confined to extract chunks only in their lexical ordering in the master document.

Both sub-figures (a and b) in Figure D.3 represent the same, singular, theme. Sub-Figure b represents the final woven and reordered theme which is contained in Sub-Figure a. It illustrates the sequence in which the chunks will be woven. Theme a illustrates the master document view of the theme document. The black edges between code chunks show their natural order as expressed in the master document. The dim-grey edges between code chunks show the order that the chunks take in the new **evaluation** theme. Code chunk **code1** is the beginning chunk in both cases. Specifically, code chunks 10 and 11 have been displaced to be presented after code chunk 15.

D.3.1 Implementation of an Enhanced Model

Following is a theme-extracted version of the theme based implementation of the model described:

Approximately, the noweb filter input assumes the following format:

```
@begin docs 3
@text Documentation chunks start with '@begin docs'. We use regular
@nl
@text expression parsing to check for the occurrence of this string.
@nl
@end docs 3
@begin code 4
@xref label NWKuRup-4U7dz4-1
@xref ref NWKuRup-4U7dz4-1
@defn scan line for doc section beginning
@xref beginuses
@xref useitem NWKuRup-1p0Y9w-1
@xref enduses
@nl
@text $line =~ /$begin\_docs/
@nl
@end code 4
```

Notice that the “docs” and “code” sections are incrementally numbered. We can make good use of this in order to help us store the documentation and code sections we see. We also make use of this in the output, and rearrangement, of the chunks.

Initially, we parse and store *all* the noweb filter input, and then rearrange this input according to a set of rules (attained from the programmer's reordering requirements).

$\langle sub\ Displacement::displace \rangle \equiv$

The sections stored in the `$Data` object are rearranged according to the order denoted in the lists read in from $\langle retrieve\ displacement\ lists \rangle$; that is, the user-defined order.

$\langle rearrange\ chunks \rangle \equiv$

One list is a line of comma delimited integers, or chunk names. Each list is separated by a new line. Leading each list is the theme name.

⟨retrieve displacement lists⟩≡

Note that the perspective we take is one that suggests a documentation section references (or is associated with) a code section. The reverse is *not* true however. This is the reason why we give as theme lists in *⟨retrieve displacement lists⟩* only the code sections. We then rely on the doc section containing an argument relating to the theme it should be included in.

⟨search backwards and find decrementing chunks⟩≡

The line given as an argument to this method contains information about the number of the section. We extract this firstly, and then access the next line which will contain any possible themes a doc section is associated with.

⟨prepare theme info from input⟩≡

We also collate information about the theme that a particular documentation section is associated with. This information is contained in the first line of a documentation section. This is the second line of the doc section in the form of: `@text <data>....`

This means we must strip out this information from the first line of the of the documentation section. We extract the space delimited themes and store the theme name along with the doc section number in a “THEME” hash of the *Nowebchunk* class.

After reading this line from the input, we can effectively discard it as we don't want to process it further and include it in the weaved document.

⟨sub Displacement::process_doc⟩≡

One of the consequences of using filters to instill a theme based approach to noweb is that we require some external processing to take place in order to write the respective documents for each theme.

This cannot be done running noweave once. Although all the necessary filtering is performed during the first execution of noweave, in order to output an individual document for each theme, we need to cheat noweave into thinking it is weaving a literate program. In reality, we simply output filtered content that was stored in several theme related files during the first execution. This output is then converted into a documentation format; T_EX in this case.

⟨manage Makefile⟩≡

```
all:    displacement.nw
        notangle -R"initiate application" displacement.nw > displace.pl
        notangle -R"package IOops" displacement.nw > IOops.pm
        notangle -R"package Nowebchunk" displacement.nw > Nowebchunk.pm
        notangle -R"package Displacement" displacement.nw > Displacement.pm
        notangle -R"manage Makefile" displacement.nw > Makefile
```

weave:

```
noweave -filter ../print.pl -filter ./displace.pl displacement.nw
noweave -filter "cat themeOne" b> themeOne.tex
noweave -filter "cat themeTwo" b> themeTwo.tex
noweave -filter "cat themeThree" b> themeThree.tex
```

traditional:

```
noweave -index displacement.nw > displacement.tex
```

D.3.2 A Summary of the Second Implementation

We have developed a filter that enables **Noweb** to conform to the displacement model shown in Section D.3. This model provides for a powerfully enabling tool. It substantially deviates from the traditional model offered by common literate programming tools. Essentially, it offers the ability to:

- store many documentation chunks for one code chunk in a central web,
- given a particular theme, output the relevant chunk with a code chunk. This is the same functionality offered by our initial implementation of the ‘switch’ based model.
- present code chunks along with theme related documentation chunks *in any order*. We have allowed the generation of multiple theme weavings; reader specific psychological ordering.

This model is similar to that offered by tools such as **Leo**. **Leo** facilitates multiple outlines, and hence, multiple psychological orders to be developed from the one web.

Separation of Content and Ordering

The expected input to the **Noweb** displacement filter is a file containing a number of lists, each of which represents a theme and the order of woven chunks. The format of the (CSV) input is:

theme name, code chunk names

An example of this input is:

```
thesisdoc,sub Displacement::displace,rearrange chunks,retrieve  
displacement lists,search backwards and find decrementing  
chunks,prepare theme info from input,sub  
Displacement::process\_doc,manage Makefile,
```

This example (used to generate a theme document used in Section D.3) is one line of input. It illustrates that the code chunks `<<sub Displacement::displace>>`, `<<rearrange chunks>>`, `<<retrieve displacement lists>>`, `<<search backwards and find decrementing chunks>>`, `<<prepare theme info from input>>`, `<<sub Displacement::process_doc>>`, and `<<manage Makefile>>`, are all to be presented, in the order implied by the list, in the `thesisdoc` theme weave. Alternatively, as in the example in Section D.4.1 on page 306, chunk numbers may be used as the comma separated values (CSV) list.

Effectively, a separation exists between the content (the web document) and the ordering of the display of chunks in the themed literate document.

A GUI-based approach will greatly enhance the benefits of the displacement filter.

Cross-Referencing and Indexing

Again, as with the initial model, the implementation of the second model does not offer the cross-referencing and indexing capability offered by **Noweb**. This functionality can be maintained by keeping track of `@xref` declarations in the **Noweb** source and only outputting the line of source if the referenced section is to be output in the theme document. For the sake of simplicity, the functionality has not been implemented.

Limitations of the Displacement Model Implementation

Inelegant theme markup: The markup that indicates inclusion of a documentation or code chunk is not elegant. We have reserved the first line of a documentation chunk as a declaration space for themes that the documentation and associated code chunk is to be included in. Thus, it is impossible utilise the **noweave** tool in its traditional non-themed manner. Amongst the documentation of the woven document will also exist the words used to describe a chunk's theme orientation.

Ugly weave commands: The bounds of **Noweb**'s capabilities are being pushed. So much so that we finally resort to use of a **Makefile** to

produce multiple documents⁴. This is because **Noweb** does not support the output of multiple documents. Indeed, the model and capabilities of **Noweb** do not support multiple input documents, and relies upon the functionality of the document formatter to offer this capability. We resort to cheating the **Noweb** parser into believing it has generated some **Noweb** low-level source code by **echoing** previously created source code files into the filter. **Noweb** was never intended for this type of manipulation; hence its ugliness.

Backwards compatibility: Such literate documents become largely incompatible with the traditional **Noweb** model. Not only will the traditional **Noweb** model not understand the newly developed markup, but, it will also be unable to weave the document and code chunks in a logical psychological order apart from the linear order in which they appear in the master document. To use the displacement model and filter means that reuse of literate code amongst those who do not possess the same filter is unfeasible.

D.4 Does Chunk Displacement Suffice?

The displacement model, presented in Section D.3, allows the multiple ordering of code chunks along with theme specific documentation chunks. It also enables the exclusion and inclusion of code chunks to a woven document. Does this functionality satisfy the needs of a theme-based approach to literate programming.

We present an example, through which we are able to lay strong foundations for enhanced model. Let's imagine that we wish to combine two themes together. Both of these themes will be extracted from the literate program which implements the displacement model filter. The theme is presented in Figure D.4 on page 352. The two sub-themes exist as separate theme documents in Appendices D.10 (theme-weaving: how **eliding** is performed), and Appendix D.11 (theme-weaving: how **displacement** is performed).

⁴equally, a shell script of some description would suffice

Note that some of the chunks in the two sub-themes appear twice. The **eliding** theme contains code chunks 4, 26, 79, 8, and 28. The **displacement** theme contains code chunks 4, 23, and 8. Both requirements make use of code chunks 4 and 8. However, both of these code chunks have a different set of associated document chunks.

This literate program contains three chunk types: (1) requirements, (2) code construction, and (3) code chunk types. Note that the two requirements chunks are orthogonal to the construction documentation and code chunks belonging to either theme.

The presentation implications of requiring such capabilities are that the literate tool support:

display of duplicate chunks: a chunk must be able to be output more than once in the woven document. This is to allow the display of code chunks 4 and 8, and construction documentation chunks 2 and 5.

differentiated document chunks: it must be possible to indicate which documentation chunk is presented with its neighboring code chunk in the woven document. This functionality is available in the initial switch filter.

multiple chunk types: it must be possible to differentiate amongst the code construction, requirement, and code chunks both in the web source and the woven document. The traditional, switch, and displacement model are all unable to do this.

D.4.1 Duplicate Chunks

The traditional model is not able to *reuse* a chunk such that it is presented more than once in the woven document⁵.

Contrastingly, the tangle process allows code chunk reuse by allowing a chunk to be referenced more than one time. In this case, it is possible to develop a literate program like the following:

⁵ some tools provide parameterised macros which do enable this, however, turning each and every chunk into a parameterised macro is an ugly solution

```

<<z>>=
  1
<<a>>=
  <<z>>, 2, 3
<<b>>=
  3, 2, <<z>>
<<*>>=
  <<a>><<b>>

```

likely to produce the tangled source:

```
1, 2, 3, 3, 2, 1
```

Notice that the body of chunks <<a>> and <> are composed of chunk <<z>>. Thus, chunk <<z>> has been reused.

The displacement model also able to include a code chunk more than once in the woven document. This is facilitated by the data file containing the code chunk in the relevant place in the comma separated list. The example in Figure D.4 would require the following to be written to the data file.

```
elide_and_displace,4,26,79,8,28,4,23,8
```

The file is then parsed by the **Noweb** filter which implements the displacement model. However, even the displacement model's implementation of code chunk replication is not very elegant. This is because the replicated chunk appears as an *additive* chunk in the literate document.

The current literate model uses the chunk name as a unique identifier for each code chunk. Including a chunk with the same name twice in the woven document will therefore indicate the existence of an additive code chunk.

The displacement filter is unable to present two identical code chunks with differing document sections. Although it is able to weave a code chunk multiple times, albeit as an additive code chunk, differentiation amongst the duplicated code chunks is not possible. It is impossible, therefore, to determine which code chunks the document section is associated with: it is not possible to associate different documentation chunks with duplicated code chunks.

Although it is possible to implement a filter which assumes the functionality required to conform to the duplicate chunk requirement, it stretches the boundaries of **Noweb**'s capabilities. **Noweb** becomes becoming unrecognisable, and the predominant processing begins to take place using filters. The implied reasoning behind the use of a filter is to enhance **Noweb**'s functionality — not to disproportionately affect **Noweb**'s 'normal' functionality. Surely not an intention of Norman Ramsey's, however, a testament to the extensibility of his tool.

Implications of Chunk Reuse In order to allow the display of a chunk's content to occur in more than one area, the chunk's implementation and its ordered display in a theme literate document must be stored separately⁶. Separating a chunk's storage from its placement in a theme allows us to associate particular attributes as to a chunk's treatment in that theme. These attributes are discussed further in Chapter 6.

D.4.2 Non-fixed Chunk Types — Higher Order Documentation

The traditional, switching, and displacement models maintain a fixed set of chunk types; namely the code chunk and the associated documentation chunk. These models suffer from the phenomenon of convergence of layers. Figure D.4 on page 352, however, presents the need for *three* chunk types:

1. code chunk
2. construction documentation chunk
3. requirements documentation chunk

It must be possible to differentiate amongst these chunk types. Currently, however, both the construction and requirement documentation chunks are treated as a singular chunk type by the displacement model — both chunk types fall under the general umbrella term of documentation chunk. The weaving process is not able to differentiate between the two.

⁶ there are other approaches to this problem, however this is the most elegant and unrestrictive

The inability for the author to specify chunk types, and therefore alter his perspective as necessary, has negative repercussions on literate program development. A fixed weaving, or formatting of each chunk type means that dynamic differentiation of chunks is not possible.

Differentiation of chunk types is necessary because:

1. either chunk type — code, construction, or requirement — may possess special formatting requirements (as do document and code chunks) depending on the nature of the literate document being woven: a chunk’s formatting is theme-dependant.
2. literate program *development* of multi-layered documents is not obvious, and furthermore, not encouraged. This non-orthogonal approach means that the current models restrict LP to the software construction phase of software engineering. What would naturally occur as layered abstractions are forced to be represented by one chunk type. Explicit differentiation and consideration of chunk types helps develop more logically sound literate documents⁷ because the author is able to separate between orthogonal concerns.

In order to treat each chunk type distinctly, a solution could be to mark each chunk’s content up with a particular set of formatting instructions. For example, a specific set of custom built L^AT_EX environments and macros could be developed for each chunk type. These environments and macros could be altered depending on the formatting necessities of each woven document.

Such a solution has a number of drawbacks however:

- it is limited to the formatting capabilities of the formatting language chosen, however. Also, it limits LP authors to use L^AT_EX only as their formatting language. This provides a sound basis to consider XML as a solution as a semantic markup tool.
- the capabilities that are endowed upon the formatting language to differentiate between each chunk type aid to exacerbate the three-syntax

⁷ we believe

problem. This is because the formatting language is used as well as the literate tool to differentiate between chunk types.

- the literate tool is unaware of the various chunk types that may be developed through the L^AT_EX macros. While the author may be able to differentiate in the woven document between chunk types, these chunk types may not be displaced and presented anywhere in the document.

The inability to differentiate amongst different chunk types, we term as *obscurity of an obscurity of hierarchical association*. This exists because the existing *hierarchical* models are not able to differentiate between different documentation types; *association* they have a fixed hierarchical representation. That is, a section can either be a document section or a code section — nothing else.

To add further concern to the capabilities of the current models, it is not necessarily the case where the developer would always treat source code as the first (and lowest) layer. What if source code *documentation* is desired to be treated as the primary layer? The source code layer would then become a higher level medium. This is not possible to represent using existing models.

The obscurity of hierarchical association is caused by what we term as the *convergence of layers*. Treating the third-layer document chunk as a second layer document chunk melds all layers above layer one into a singular, all-containing, layer. The transformation of converging what is a logically a three-layer literate *program* into a two layer literate *document* representation essentially causes is a *loss of information* due to this singular, flattened second layer.

Some issues arise from this convergence of layers:

Logical association of sections is depreciated: Representing the third layer chunk in a second layer chunk, it is forced to be associated with a first-layer section. It is this first-layer section that the third-layer section now describes; not what was initially intended. The intention was to associate third-layer sections with second-layer (document) sections.

A third-layer requirements section, for example, is unable to be correctly associated with the necessary second layer document sections.

The association is therefore expressed in the traditional model's capabilities by associating it with a code chunk (first-layer section). The code chunk in this case would possess a null body, and is simply used to denote the requirement section's hierarchical status. This process devalues the document and code section's ability to represent each other in a consistent fashion.

Loss of section semantic information: The third layer requirement section now becomes a hierarchically superior second layer document section. There is no clear indication that the third layer chunk is indeed a requirement, or just another second layer document section. There now exists a reliance on the chunk developer to explicitly mention that the content of the section is a requirement.

This means that the extraction of a requirement section, and the document sections it is associated with is impossible — unless the displacement model is distorted somewhat: the requirement section labelled with a theme name, and the associated document sections labelled with a sub-theme. The theme and sub-theme are then displaced and a literate document woven. This is not a full-proof solution however, especially if multiple third-layer sections are associated with the same document sections.

Overhead on document development: Traditional literate programming places undue overhead on document development. The literate document in the traditional sense is all-encompassing. It presents all the information available in the literate program to the reader. It is impossible to exclude, or re-order particular document sections. This means that the literate document must embody all information about the program. In our example, this means that the literate document must contain all the specifications, and blend these in a logically flowing manner, whereby part of the specification documentation, and the implementation of this specification is spread across the document sections that are associated with their code sections.

The problem with this approach is that it is difficult to follow a particular path throughout the literate document's development, and presentation of information, using the conventional model.

disproportionate literate programming: The emphasis begins to bear ever more towards the development of a comprehensible document, rather than application development.

Given multiple requirements, more effort will go into document presentation, and logical layout, editing, etc, rather than program development.

Including the (third layer) requirements in the literate program in this flat structure using the traditional model means that a much greater emphasis (arguably over-bearing and unnecessary) will be placed upon the logical flow of the document.

Using the enhanced model to omit unrelated (to the requirement) documentation sections from the literate document solves this problem, we are still faced with the other problems...

directly relevant information is obscured: Important items may become obscured by the surrounding context. This is due to the *convergence of documentation*. Convergence of documentation is directly related to lateral (on the same hierarchical layer) uncohesive documentation practices, whereby a given code chunk is documented with one document section which covers many points of interest. In our case, this may be the manner in which two (or more) requirements affect the construction of the application. Unless the displacement model is used to exclude unneeded document sections from the woven literate document, all document sections must be included, and therefore multiple themes will be incorporated in the document section.

Converging layers does not necessarily mean that documentation chunks themselves are converged. Converging documentation can be avoided by using the displacement literate model, however is unavoidable when using the traditional model.

*obsured points of
interest
convergence of
documentation*

Does the programmer write new chunks to explicitly mark a point of particular interest? Using the displacement approach, this is possible. Without it, the conventional model can only serve to represent all information, thereby obscuring points of interest of a particular theme.

false hierarchies: Due to the uni-typed documentation chunk, more use must be made of the code chunk delimiter as an abstraction tool. This has the effect of creating a hierarchy of chunks where there perhaps would not have been — false hierarchies — the requirements section, now treated as a second level section, imposes itself as a hierarchically superior document section.

1. a document section is not able to be excluded from the “all”, or global, weaving of a document.
2. the content, or implementation of a code chunk is not able to be excluded from the output of a chunk.
3. a documentation chunk is unable formed from other documentation chunks.

These items are explained further.

D.5 Theme 1 Example: Bubblesort

evaluation While bubblesort may have a cute name, its performance may leave a lot to be desired. Bubblesort has a notorious reputation for being inefficient. While this may be true for randomly sorted lists of numbers, bubblesort is one of the better performing sorting methods on near-sorted lists.

This program implements and evaluates the bubblesort algorithm. Primarily, we generate the number of comparisons and swaps performed given a set of arrays to be sorted. Specifically, four arrays are sorted:

- a randomly sorted array.
- a partially sorted array.
- a reverse sorted array.
- a sorted array.

```
314  <* 314>≡  
      <use packages 320g>  
      <test bubblesort 316a>  
      <define sub bubblesort 315>  
      <define sub bubbletest 319>
```

evaluation algorithm

Bubblesort makes multiple scans through the array to be sorted, swapping adjacent pairs of elements if they re in the wrong order, until no more swaps are necessary. The bubble is the element propogating through the array.

algorithm

This bubblesort routine requires, as an argument, a reference to an array of unsorted numbers. The array after being sorted, is then available outside the function.

e.g.,

```
@array = (1, 5, 9, 7);  
bubblesort(\@array);  
print "@array";
```

```
315  <define sub bubblesort 315>≡ (314)  
    sub bubblesort {  
        <initialise variables 318>  
        <iterate backwards over array 317b>  
        {  
            <iterate forwards over array up to current backwards position 317c>  
            {  
                <gather comparisons data 320b>  
                <swap adjacent members in array if necessary 317a>  
            }  
        }  
  
        <output results 316b>  
    }
```

Defines:

bubblesort, used in chunks 316b and 319.

evaluation

Running the `bubbletest` function 10 times gives the following results:

```
@a: 1000 elements, 499500 comparisons, 0 swaps
@b: 1000 elements, 499500 comparisons, 246925 swaps
@c: 1000 elements, 499500 comparisons, 499500 swaps
@d: 1010 elements, 509545 comparisons, 9945 swaps
```

The results show us that the bubblesort algorithm is, in the worst case (@c), ON^2 . There are as many swaps as there are comparisons. The algorithm is ON^2 , derived from the equation $N(N-1)/2$.

For a random set of data, there are approximately half the swaps that there are comparisons.

A near-sorted array, with only a few elements out of order, yields a far more tolerable result. In fact, bubblesort is one of the fastest algorithms of all in such situations.

Given its poor worst and average case results, bubblesort is a poor choice as a general sorting algorithm.

For all intensive purposes, bubblesort is an Order N^2 algorithm.

```
316a  <test bubblesort 316a>≡ (314)
      &bubbletest;
```

```
316b  <output results 316b>≡ (315)
      print "bubblesort: ", int @$array,
      " elements, $ncomp comparisons, $nswap swaps\n";
```

Uses `bubblesort` 315, `$ncomp` 318, and `$nswap` 318.

algorithm

A swap is performed if the current member in the array is greater than the member before it. In this fashion, we sift the bubble upwards through the array.

317a $\langle \text{swap adjacent members in array if necessary 317a} \rangle \equiv$ (315)

```
    if ($array->[ $j - 1 ] > $array->[ $j ] ) {  
        @array[ $j, $j - 1 ] = @array[ $j - 1, $j ];  
         $\langle \text{gather swap data 320a} \rangle$   
    }
```

Uses \$j 318.

algorithm

The first loop iterates over the array backwards, starting at the last element of the array in the first iteration. The next iteration starts at the element before the last element at the end of the array. In other words, we iterate backwards over the array storing the value in \$i.

317b $\langle \text{iterate backwards over array 317b} \rangle \equiv$ (315)

```
    for ($i = $#array; $i; $i--)
```

Uses \$i 318.

algorithm

317c $\langle \text{iterate forwards over array up to current backwards position 317c} \rangle \equiv$ (315)

```
    for ($j = 1; $j <= $i; $j++)
```

Uses \$i 318 and \$j 318.

\$ncomp : counts the number of comparisons.

\$nswap : counts the number of swaps.

\$array : a reference to an array of numbers to be sorted.

318 $\langle \textit{initialise variables 318} \rangle \equiv$ (315)
 my \$array = shift;

 my \$i;

 my \$j;

 my \$ncomp = 0;

 my \$nswap = 0;

Defines:

\$i, used in chunk 317.

\$j, used in chunk 317.

\$ncomp, used in chunks 316b and 320b.

\$nswap, used in chunks 316b and 320a.

evaluation We generate a random array of numbers to pass to the `bubblesort` function.

`bubbletest` builds the `a`, `textttb`, `textttc`, and `texttt d`, arrays, and then proceeds to sort them using the `bubblesort` algorithm.

```
319  <define sub bubbletest 319>≡ (314)
    sub bubbletest {

        my @a;
        my @b;
        my @c;
        my @d;
        for (1..1000) {
            <populate sorted array 320c>
            <populate randomly sorted array 320d>
        }

        <populate reverse sorted array 320e>
        <populate partially sorted array 320f>

        &bubblesort(\@a);
        &bubblesort(\@b);
        &bubblesort(\@c);
        &bubblesort(\@d);

    }
```

Uses `bubblesort` 315.

The maximum number of swaps that can occur is the same as the number of comparisons. This would occur if the array was presented in reverse sorted order to the bubblesort algorithm.

Otherwise, a swap only happens when an element is greater than its right-neighboring element.

320a $\langle gather\ swap\ data\ 320a \rangle \equiv$ (317a)
 `$nswap++;`

Uses `$nswap` 318.

320b $\langle gather\ comparisons\ data\ 320b \rangle \equiv$ (315)
 `$ncomp++;`

Uses `$ncomp` 318.

evaluation

320c $\langle populate\ sorted\ array\ 320c \rangle \equiv$ (319)
 `push @a, $_;`

evaluation

320d $\langle populate\ randomly\ sorted\ array\ 320d \rangle \equiv$ (319)
 `push @b, sprintf("%d",rand() * 100);`

evaluation

Make use of perl's reverse function to reverse the `@a` array.

320e $\langle populate\ reverse\ sorted\ array\ 320e \rangle \equiv$ (319)
 `@c = reverse @a;`

evaluation

Append ten numbers to the end of a sorted set of data.

320f $\langle populate\ partially\ sorted\ array\ 320f \rangle \equiv$ (319)
 `@d = (@a, 1..10);`

320g $\langle use\ packages\ 320g \rangle \equiv$ (314)
 `use strict 'vars';`

D.6 Theme 2 Example: Bubblesort Evaluation

While bubblesort may have a cute name, its performance may leave a lot to be desired. Bubblesort has a notorious reputation for being inefficient. While this may be true for randomly sorted lists of numbers, bubblesort is one of the better performing sorting methods on near-sorted lists.

This program implements and evaluates the bubblesort algorithm. Primarily, we generate the number of comparisons and swaps performed given a set of arrays to be sorted. Specifically, four arrays are sorted:

- a randomly sorted array.
- a partially sorted array.
- a reverse sorted array.
- a sorted array.

```
 $\langle * \rangle \equiv$   
 $\langle use\ packages \rangle$   
 $\langle test\ bubblesort \rangle$   
 $\langle define\ sub\ bubblesort \rangle$   
 $\langle define\ sub\ bubbletest \rangle$ 
```

Bubblesort makes multiple scans through the array to be sorted, swapping adjacent pairs of elements if they re in the wrong order, until no more swaps are necessary. The bubble is the element propogating through the array.

```

⟨define sub bubblesort⟩≡
  sub bubblesort {
    ⟨initialise variables⟩
    ⟨iterate backwards over array⟩
    {
      ⟨iterate forwards over array up to current backwards position⟩
      {
        ⟨gather comparisons data⟩
        ⟨swap adjacent members in array if necessary⟩
      }
    }
    ⟨output results⟩
  }

```

Running the `bubbletest` function 10 times gives the following results:

```
@a: 1000 elements, 499500 comparisons, 0 swaps
@b: 1000 elements, 499500 comparisons, 246925 swaps
@c: 1000 elements, 499500 comparisons, 499500 swaps
@d: 1010 elements, 509545 comparisons, 9945 swaps
```

The results show us that the bubblesort algorithm is, in the worst case (@c), ON^2 . There are as many swaps as there are comparisons. The algorithm is ON^2 , derived from the equation $N(N-1)/2$.

For a random set of data, there are approximately half the swaps that there are comparisons.

A near-sorted array, with only a few elements out of order, yields a far more tolerable result. In fact, bubblesort is one of the fastest algorithms of all in such situations.

Given its poor worst and average case results, bubblesort is a poor choice as a general sorting algorithm.

For all intensive purposes, bubblesort is an Order N^2 algorithm.

```
<test bubblesort>≡
    &bubbletest;
```

D.6.1 Test Environment

We generate a random array of numbers to pass to the `bubblesort` function.

`bubbletest` builds the `a`, `textttb`, `textttc`, and `texttt d`, arrays, and then proceeds to sort them using the `bubblesort` algorithm.

```
<define sub bubbletest>≡
  sub bubbletest {

    my @a;
    my @b;
    my @c;
    my @d;
    for (1..1000) {
      <populate sorted array>
      <populate randomly sorted array>
    }

    <populate reverse sorted array>
    <populate partially sorted array>

    &bubblesort(\@a);
    &bubblesort(\@b);
    &bubblesort(\@c);
    &bubblesort(\@d);

  }

<populate sorted array>≡
  push @a, $_;

<populate randomly sorted array>≡
  push @b, sprintf("%d",rand() * 100);
```

Make use of perl's reverse function to reverse the @a array.

⟨populate reverse sorted array⟩≡

```
@c = reverse @a;
```

Append ten numbers to the end of a sorted set of data.

⟨populate partially sorted array⟩≡

```
@d = (@a, 1..10);
```

D.7 Theme 3 Example: Bubblesort Algorithm

Bubblesort makes multiple scans through the array to be sorted, swapping adjacent pairs of elements if they re in the wrong order, until no more swaps are necessary. The bubble is the element propogating through the array.

This bubblesort routine requires, as an argument, a reference to an array of unsorted numbers. The array after being sorted, is then available outside the function.

e.g.,

```
@array = (1, 5, 9, 7);  
bubblesort(\@array);  
print "@array";
```

```
<define sub bubblesort>≡  
sub bubblesort {  
    <initialise variables>  
    <iterate backwards over array>  
    {  
        <iterate forwards over array up to current backwards position>  
        {  
            <gather comparisons data>  
            <swap adjacent members in array if necessary>  
        }  
    }  
  
    <output results>  
}
```

A swap is performed if the current member in the array is greater than the member before it. In this fashion, we sift the bubble upwards through the array.

```

<swap adjacent members in array if necessary>≡
    if ($array->[ $j - 1 ] > $array->[ $j ] ) {
        @$array[ $j, $j - 1 ] = @$array[ $j - 1, $j ];
        <gather swap data>
    }

```

The first loop iterates over the array backwards, starting at the last element of the array in the first iteration. The next iteration starts at the element before the last element at the end of the array. In other words, we iterate backwards over the array storing the value in \$i.

```

<iterate backwards over array>≡

    for ($i = $$array; $i; $i--)

<iterate forwards over array up to current backwards position>≡
    for ($j = 1; $j <= $i; $j++)

```

D.8 NowebTheme Converter

To develop a filter for **Noweb** that implements this ruleset is not a particularly formidable task. In fact, given the right language for the task, it is a rather simple process.

Using Perl as the programming language, and possessing knowledge of the **Noweb** filtering source code file, we can implement an algorithm to process the **Noweb** input:

We loop and process input from STDIN; line by line. We then scan each line to find documentation sections with the relevant theme attributes. If these theme attributes are present we carry on and include the documentation section, and its associated code section in the output. Otherwise, if the attributes are not present, we simply exclude the source from output to STDOUT.

```
328  <* 328>≡
    <set up operating environment 332d>
    <process initial input 332e>

while ($line = &readln) {
    if ( <find doc section beginning 329b> ) {
        <keep track of previous line 330b>
        for my $theme (@themes) {
            if ( <scan line for theme attributes 330a> ) {
                &println($prev_line);
                <include doc section 332c>
                $INCLUDE = 1;
                last;          # include doc chunk once only
            }
        }
    }

    if ( <find code section beginning 329c> && $INCLUDE == 1) {
        &println($line);
    }
}
```

```

    <include code section 332b>
        $INCLUDE = 0;
    }

}

```

<define subroutines 330c>

thesisdoc One disadvantageous repercussion from performing noweb theme based literate programming. All indexing and cross-referencing capabilities are lost.

We purposely exclude all cross references to chunks and definitions because, unfortunately, it is cumbersome to determine which chunks will be included in the L^AT_EX output. Cross-references are excluded via the `process_doc` and `process_code` functions.

To do this would require processing the entire input and keeping state which chunks have been included. We would then be required to reprocess the input and include cross-references to the chunks to that are included in the output document. This process can most definitely be done, however, it is rather ugly.

```

329a  <discard cross references 329a>≡ (330d 331a)
      next if $line =~ /$xref/;

```

Documentation chunks start with “@begin docs”. We use regular expression parsing to check for the occurrence of this string.

```

329b  <find doc section beginning 329b>≡ (328)
      $line =~ /$begin_docs/

```

```

329c  <find code section beginning 329c>≡ (328)
      $line =~ /$begin_code/

```

Using regular expression matching, we scan the first line of the documentation section for the existence of the current theme.

330a *⟨scan line for theme attributes 330a⟩*≡ (328)
 \$line =~ /\$theme/

330b *⟨keep track of previous line 330b⟩*≡ (328)
 \$prev_line = \$line;
 \$line = &readln;

`process_doc` and `process_code` are used to print to STDOUT until the end of the respective documentation or code section is encountered.

`println` and `readln` are used to commit the writing to STDOUT and reading from STDIN to one area. This can aid in testing (as it did for the author ;).

330c *⟨define subroutines 330c⟩*≡ (328)
 ⟨sub process_doc 330d⟩
 ⟨sub process_code 331a⟩
 ⟨sub println 331b⟩
 ⟨sub readln 331c⟩
 ⟨sub read_themes_file 332a⟩

330d *⟨sub process_doc 330d⟩*≡ (330c)
 sub process_doc {
 my \$line;
 while ((\$line = &readln) && (\$line !~ /\$end_docs/)) {
 ⟨discard cross references 329a⟩
 &println(\$line);
 }
 print \$line;
 }

331a $\langle \text{sub process_code 331a} \rangle \equiv$ (330c)

```

sub process_code {
    my $line;
    while ( ($line = &readln) && ($line !~ /$end_code/)) {
         $\langle \text{discard cross references 329a} \rangle$ 
        &println($line);
    }
    print $line;
}

```

shift the element off the array of arguments passed to this function and print it.

331b $\langle \text{sub println 331b} \rangle \equiv$ (330c)

```

sub println {
    my $line = shift;
    print $line;
    print FILE $line;
}

```

331c $\langle \text{sub readln 331c} \rangle \equiv$ (330c)

```

sub readln {
    my $line = <>;
    return $line;
}

```

332a *⟨sub read_themes_file 332a⟩*≡ (330c)

```

sub read_themes_file {
    open THEMES, "<themes";
    my @themes;
    for ( <THEMES> ) {
        chomp;
        push @themes, $_;
    }
    close THEMES;

    return @themes;
}

```

332b *⟨include code section 332b⟩*≡ (328)

```

&process_code();

```

332c *⟨include doc section 332c⟩*≡ (328)

```

&process_doc();

```

332d *⟨set up operating environment 332d⟩*≡ (328)

```

#! c:/perl/bin/perl.exe
⟨use packages 334a⟩
⟨define variables 333b⟩

```

332e *⟨process initial input 332e⟩*≡ (328)

```

if ($themes[0] eq "all") {
    ⟨transfer STDIN to STDIN 333a⟩
}
⟨output first line of input 334b⟩

```

In order to save on unrequired processing, we check to whether the keyword “all” exists. If it does, simply print STDIN to STDOUT and exit ruthlessly from the program.

Not so elegant, but efficiently practical.

```
333a  <transfer STDIN to STDOUT 333a>≡ (332e)
      print <>;
      exit;
```

```
333b  <define variables 333b>≡ (332d)
      my $begin_docs = "^\\@begin docs";
      my $end_docs = "^\\@end docs";
      my $begin_code = "^\\@begin code";
      my $end_code = "^\\@end code";
      my $xref = "^\\@xref";
      my @themes = <read themes from file 333c>;
      print STDERR "themes = '@themes'\n";
      my $INCLUDE = 0;
      my $line = "";
      my $prev_line;
      open FILE, ">file.out2";
```

Note that a file named `themes` must exist with all themes to be output line delimited.

```
333c  <read themes from file 333c>≡ (333b)
      &read_themes_file();
```

We firstly need to look out for a “@begin docs” line. Take note that each code chunk, given our previously stated rules, must have a documentation chunk. This means that *every* `@begin code` has one or more set of `@begin docs` and `@end docs` associated with it. This is because a code chunk is able to have multiple themes associated with it.

The package `strict` restricts us from using previously undeclared variables.

334a $\langle use\ packages\ 334a \rangle \equiv$ (332d)
 `use strict 'vars';`

The first line from STDIN is the name of the noweb file being processed, e.g., `@file themenoweb1.nw`. We print this to STDOUT.

334b $\langle output\ first\ line\ of\ input\ 334b \rangle \equiv$ (332e)
 `$line = &readln;`
 `&println($line);`

D.9 NowebDisplacement Theme Converter Overview

Approximately, the noweb filter input assumes the following format:

```
@begin docs 3
@text Documentation chunks start with ‘‘@begin docs’’. We use regular
@nl
@text expression parsing to check for the occurrence of this string.
@nl
@end docs 3
@begin code 4
@xref label NWKuRup-4U7dz4-1
@xref ref NWKuRup-4U7dz4-1
@defn scan line for doc section beginning
@xref beginuses
@xref useitem NWKuRup-1p0Y9w-1
@xref enduses
@nl
@text $line =~ /$begin\_docs/
@nl
@end code 4
```

Notice that the “docs” and “code” sections are incrementally numbered. We can make good use of this in order to help us store the documentation and code sections we see. We also make use of this in the output, and rearrangement, of the chunks.

Initially, we parse and store *all* the noweb filter input, and then rearrange this input according to a set of rules (attained from the programmer’s reordering requirements).

$\langle sub\ Displacement::displace \rangle \equiv$

The sections stored in the `$Data` object are rearranged according to the order denoted in the lists read in from $\langle retrieve\ displacement\ lists \rangle$; that is, the user-defined order.

$\langle rearrange\ chunks \rangle \equiv$

One list is a line of comma delimited integers, or chunk names. Each list is separated by a new line. Leading each list is the theme name.

⟨retrieve displacement lists⟩≡

Note that the perspective we take is one that suggests a documentation section references (or is associated with) a code section. The reverse is *not* true however. This is the reason why we give as theme lists in *⟨retrieve displacement lists⟩* only the code sections. We then rely on the doc section containing an argument relating to the theme it should be included in.

⟨search backwards and find decrementing chunks⟩≡

The line given as an argument to this method contains information about the number of the section. We extract this firstly, and then access the next line which will contain any possible themes a doc section is associated with.

⟨prepare theme info from input⟩≡

We also collate information about the theme that a particular documentation section is associated with. This information is contained in the first line of a documentation section. This is the second line of the doc section in the form of: `@text <data>....`

This means we must strip out this information from the first line of the of the documentation section. We extract the space delimited themes and store the theme name along with the doc section number in a “THEME” hash of the *Nowebchunk* class.

After reading this line from the input, we can effectively discard it as we dont want to process it further and include it in the weaved document.

⟨sub Displacement::process_doc⟩≡

One of the consequences of using filters to instill a theme based approach to noweb is that we require some external processing to take place in order to write the respective documents for each theme.

This cannot be done running noweave once. Although all the necessary filtering is performed during the first execution of noweave, in order to output an individual document for each theme, we need to cheat noweave into thinking it is weaving a literate program. In reality, we simply output filtered content that was stored in several theme related files during the first execution. This output is then converted into a documentation format; T_EX in this case.

⟨manage Makefile⟩≡

```
all:    displacement.nw
        notangle -R"initiate application" displacement.nw > displace.pl
        notangle -R"package IOops" displacement.nw > IOops.pm
        notangle -R"package Nowebchunk" displacement.nw > Nowebchunk.pm
        notangle -R"package Displacement" displacement.nw > Displacement.pm
        notangle -R"manage Makefile" displacement.nw > Makefile
```

weave:

```
noweave -filter ../print.pl -filter ./displace.pl displacement.nw
noweave -filter "cat themeOne" b> themeOne.tex
noweave -filter "cat themeTwo" b> themeTwo.tex
noweave -filter "cat themeThree" b> themeThree.tex
```

traditional:

```
noweave -index displacement.nw > displacement.tex
```

D.10 NowebDisplacement Theme Converter Elide

Approximately, the noweb filter input assumes the following format:

```
@begin docs 3
@text Documentation chunks start with ‘‘@begin docs’’. We use regular
@nl
@text expression parsing to check for the occurrence of this string.
@nl
@end docs 3
@begin code 4
@xref label NWKuRup-4U7dz4-1
@xref ref NWKuRup-4U7dz4-1
@defn scan line for doc section beginning
@xref beginses
@xref useitem NWKuRup-1p0Y9w-1
@xref enduses
@nl
@text $line =~ /$begin\_docs/
@nl
@end code 4
```

Notice that the “docs” and “code” sections are incrementally numbered. We can make good use of this in order to help us store the documentation and code sections we see. We also make use of this in the output, and rearrangement, of the chunks.

Initially, we parse and store *all* the noweb filter input, and then rearrange this input according to a set of rules (attained from the programmer's reordering requirements).

```
<sub Displacement::displace>≡  
sub displace {  
  use Data::Dumper;  
  <Displacement::displace variables>  
  
  $line = $IO->readln();  
  $IO->println($line);  
  
  <populate data structures>  
  <retrieve displacement lists>  
  <retrieve elision lists>  
  <rearrange chunks>  
  
  return $Data;  
}
```

This function is similar to the *<retrieve displacement lists>* chunk. Perhaps it suggests that we need a `read_file` method.

```
<retrieve elision lists>≡
my %elidedisplists;

open ELIDELISTS, "<displistselide" or die "Can't open file: $!\n";

for my $line (<ELIDELISTS>) {
    <omit comment and non content lines>
    @listarray = split (/,/, $line);
    $name = shift @listarray;

    for my $item (@listarray) {
        if ($item =~ /\d+$/) {
            push @{$elidedisplists{"$name"}}, $item;
        } else {
            push @{$elidedisplists{"$name"}}, $Data->{CODEREF>{"$item"};
        }
    }
}

close ELIDELISTS;
```

```

<sub Displacement::elide_chunk>≡
sub elide_chunk {
    my $num = shift;
    my $array = shift;

    #    print STDERR "matching: '$num' with '@$array'\n";

    for my $match (@{$array}) {
    #        print STDERR "matching: '$match' with '$num'\n";
        return 1 if $match == $num;
    }

    return 0;
}

```

The sections stored in the `$Data` object are rearranged according to the order denoted in the lists read in from *<retrieve displacement lists>*; that is, the user-defined order.

```
<rearrange chunks>≡
my $dochashref = $Data->get_doc();
my $codhashref = $Data->get_code();

my $array;
my @docnumbers = sort {$a <=> $b} keys %$dochashref;
my @codnumbers = sort {$a <=> $b} keys %$codhashref;
my $num;

#print STDERR "docnums @docnumbers\n";
for my $theme (keys %displists) {
    $list = $displists{"$theme"};
    #print STDERR "opening '$theme' @$list\n" if $DEBUG;
    open LS, ">$theme" or die "Couldn't open '$theme'. $!.\n";
    print LS "\@file $theme\n";

    for $num (@$list) {

        #print STDERR "Found doc for code '$num' docsections '";
        <find document section>
        $array = $$codhashref{$num};
        if (&elide_chunk($num, $elidedisplists{"$theme"}) == 1 ) {
            <do not output code chunk>
        } else {
            <output code chunk>
        }
        #print STDERR "''\n";

        $count ++;
    }
}
```

```
    close LS;
}
```

An elision lists file may contain empty lines. Comments may also take the form of `# text`. The ‘`#`’ character must appear at the very beginning of the line to be commented.

⟨omit comment and non content lines⟩≡

```
next if $line =~ /^\\s*$/;
next if $line =~ /^\\#/;
```

D.11 NowebDisplacement Theme Converter Displace

Initially, we parse and store *all* the noweb filter input, and then rearrange this input according to a set of rules (attained from the programmer's reordering requirements).

```
<sub Displacement::displace>≡
sub displace {
  use Data::Dumper;
  <Displacement::displace variables>

  $line = $IO->readln();
  $IO->println($line);

  <populate data structures>
  <retrieve displacement lists>
  <retrieve elision lists>
  <rearrange chunks>

  return $Data;
}
```

The %displists hash is used to store the array of lists indexed by the name of the theme.

If integer values are offered in the list, we store this number along with the theme name. If a chunk name is given instead, we resolve the chunk name to the chunk's position in the document, thereby returning a number. *We assume here that there can be no chunk names that consist solely of numbers.*

<retrieve displacement lists>≡

```
my %displists;
my @listarray;
my $name;

open LISTS, "<displists" or die "Can't open file: $!\n";

for my $line (<LISTS>) {
    next if $line =~ /\s*$/;
    @listarray = split (/,/, $line);
    $name = shift @listarray;

    for my $item (@listarray) {
        if ($item =~ /\d+$/) {
            push @{$displists{"$name"}}, $item;
        } else {
            push @{$displists{"$name"}}, $Data->{CODEREF>{"$item"};
        }
    }
}

close LISTS;
# print out the displacement list chunk numbers.
for (keys %displists) {
    print STDERR "$_: ", (join ", ", @{$displists{$_}}), "\n";
}
```

The sections stored in the **\$Data** object are rearranged according to the order denoted in the lists read in from *⟨retrieve displacement lists⟩*; that is, the user-defined order.

The displaced code chunks order is stored in the `$list` array reference. This is the order traversed in the inner ‘for’ loop.

```
<rearrange chunks>≡
my $dochashref = $Data->get_doc();
my $codhashref = $Data->get_code();

my $array;
my @docnumbers = sort {$a <=> $b} keys %$dochashref;
my @codnumbers = sort {$a <=> $b} keys %$codhashref;
my $num;

#print STDERR "docnums @docnumbers\n";
for my $theme (keys %displists) {
    $list = $displists{"$theme"};
    #print STDERR "opening '$theme' @$list\n" if $DEBUG;
    open LS, ">$theme" or die "Couldn't open '$theme'. $!.\n";
    print LS "\@file $theme\n";

    for $num (@$list) {

        #print STDERR "Found doc for code '$num' docsections '";
        <find document section>
        $array = $$codhashref{$num};
        if (&elide_chunk($num, $elidedisplists{"$theme"}) == 1 ) {
            <do not output code chunk>
        } else {
            <output code chunk>
        }
        #print STDERR "'\n";

        $count ++;
    }
}
```

```
        close LS;  
    }
```

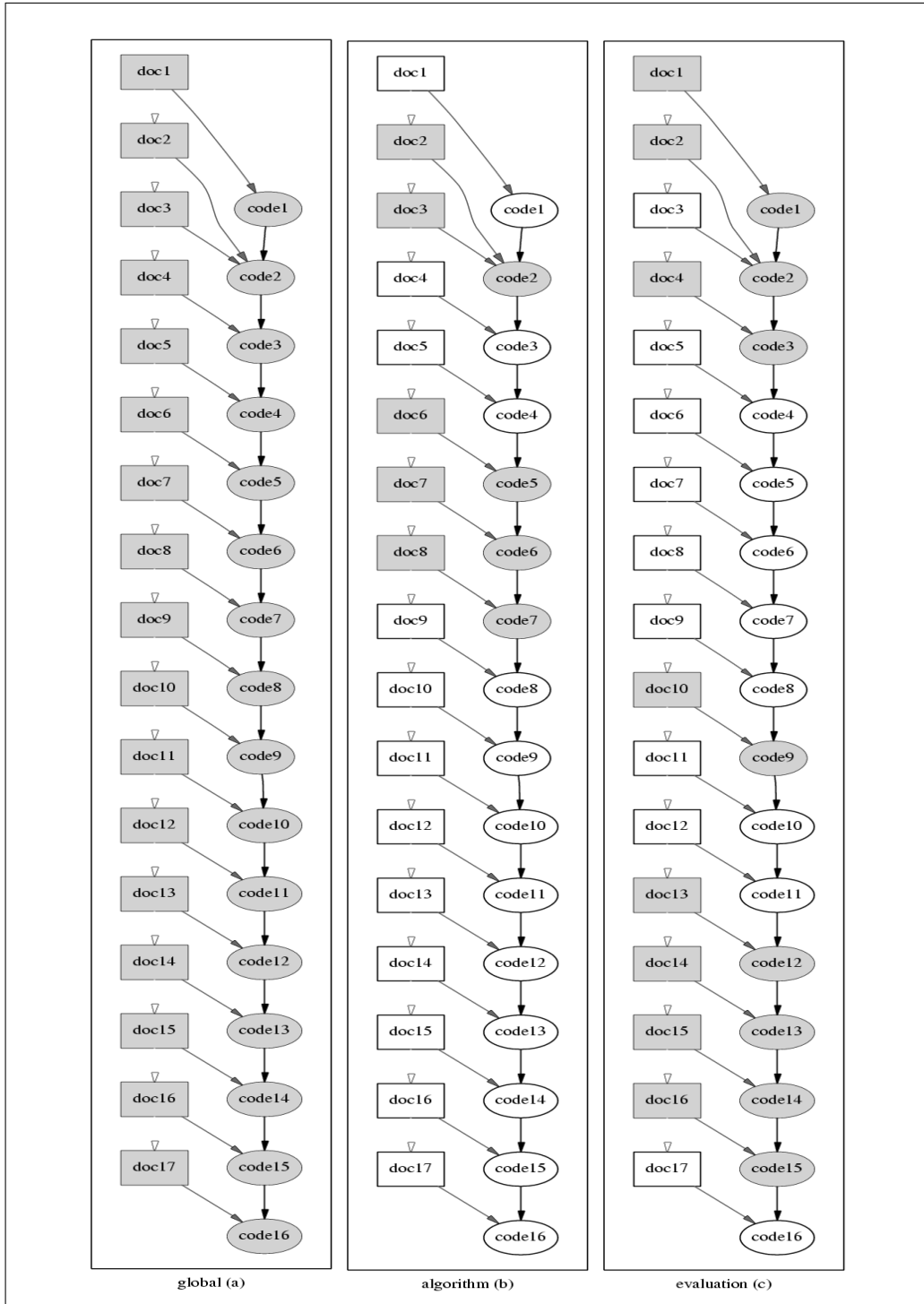


Figure D.1: Themes that a bubblesort literate program is required to presented in.

-
- a documentation chunk begins with an ‘@’ character followed by a space character ‘ ’.
 - the first line of a documentation chunk is reserved for the theme attributes that the documentation chunk should be included in. If the chunk does not belong to any specific theme, then this line *must* be kept empty.
 - themes are space delimited, and hence, should be one word.
 - single or double quotes are used for multi-word themes.
 - the theme keyword ‘all’ denotes that a documentation chunk should occur in all woven documents.
 - more than one documentation chunk may be associated with a code section.
 - the inclusion of a documentation chunk in the woven output will include the accompanying code chunk also. If more than one documentation chunk exists for a code chunk, then the code chunk will be included in the output even if only one accompanying document chunk is included in a theme.
 - a code chunk with zero accompanying documentation chunks will not be woven.
 - a code chunk may be included in a theme output without any documentation. This is performed by writing a documentation chunk with the necessary theme information, and no lines of documentation. e.g.,

```
@ tutorialtheme <<look out for nasty characters>>=
```

Figure D.2: A list of requirement specifications for the new literate model.

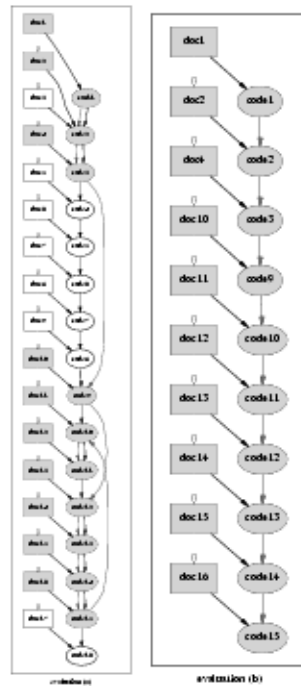


Figure D.3: A displaced version of the evaluation theme. **a** shows the literate program's complete set of originally ordered sections. **b** shows the included set of sections in their displaced order.

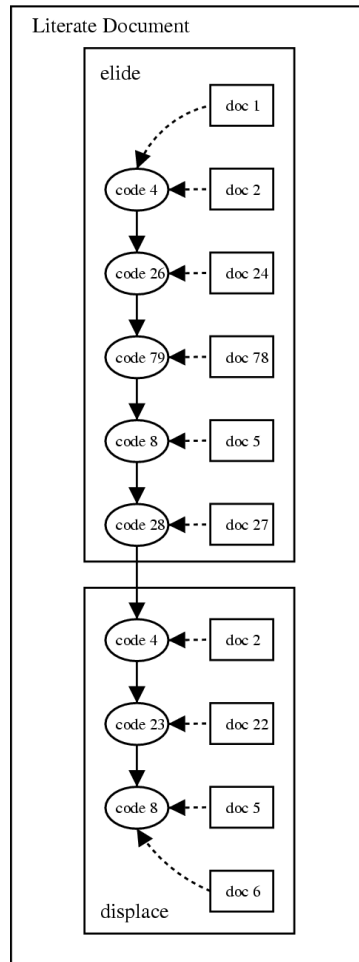


Figure D.4: Abstract diagram of included document and code sections in the eliding and displacing themes extracted from the web in Appendix D.9.

Appendix E

Theme-Based Version Control

E.1 Multi-chunk version control

Complete document rollbacks are not enabled in the CBDE. This is because a version relationship is not supported by the current chunk model, and therefore, does not exist between chunks.

Document rollbacks can be implemented by enhancing the theme model¹(see Section 5.2.5 on page 107 for a description of the theme model) (instead of the chunk model): we extend the theme model such that a theme is given a version attribute. This attribute is numbered according to the version numbering scheme presented in Section 95 on page 198. The XML theme source document (see Section 6.2.2 on page 125) is attributed this version number and stored as a *snapshot* of a theme in a point in time.

Storing theme documents as versions enables the author to rollback to a particular branch in the theme versioning tree and continue development — all chunk-references are resolved and authoring continues. In effect this is the same as saving multiple copies of a theme document and loading them as necessary.

We illustrate document rollback and show how isolated chunk versioning lends itself naturally to a more globally oriented document versioning. Figure E.1 on the next page presents an abstract versioned view of a TBLP theme set. In this literate program, three chunks exist; <<a>>, <>, and <<c>>. Each chunk has an established version tree. Each chunk's version tree is shown in the lower half of the figure. Two themes exist that represent complete theme versions. Theme version 1 contains version 1.3 of chunk <<a>>,

¹(

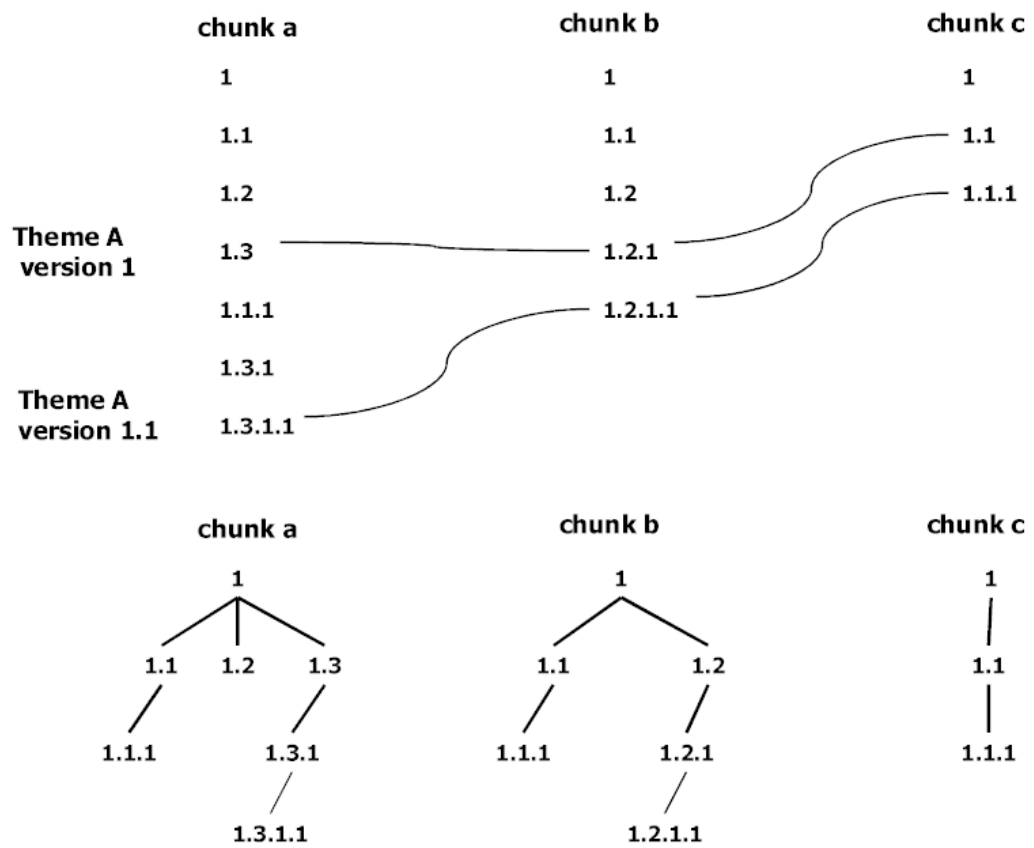


Figure E.1: An example of theme-based chunk version control. Three chunk version hierarchies are utilised to compose a two theme versions.

version 1.2.1 of chunk <>, and version 1.1 of chunk <<c>>. Theme version 1.1 contains version 1.3.1.1 of chunk <<a>>, version 1.2.1.1 of chunk <>, and version 1.1.1 of chunk <<c>>. The programmer is currently working on the chunks contained in theme version 1.1, but wishes to revert back to the theme's version 1 state. The chunk references made in theme version 1 are resolved from the repository, and the new theme attributed a version number of 1.2.

Note that this rollback is not universal. All other themes are not affected. Research should be conducted to determine the worthiness and appropriate methods with which to offer such functionality.

Investigation into chunk-based document rollbacks, whereby a theme, or set of themes, are rolled back according to a given chunk's version number is an area of further research. This is powerful functionality, however, it is likely to incur processing penalties and necessitate a more elaborate chunk and theme model. It is also uncertain whether such functionality is useful, or potentially catastrophic².

² far be it from us to judge

Appendix F

Reverse Engineering

In this chapter, we present a method to perform automated conversions from legacy source code to the literate equivalent. We are effectively concerned with identifying a consistent and automated manner of TBLP program construction.

We stress that this is a tentative proposal and requires further research to develop effective and useful conversion methods.

We firstly use the concept of psychological scope to motivate our methods of legacy source code conversion. We then propose a set of guidelines to convert specific programming language constructs to their literate equivalent.

F.1 Psychological Scope — Its Affect on LP

Psychological scope was introduced in Chapter A, Section A.3.2. In essence, it is about explicitly acknowledging the existence of the underlying programming languages' constructs and modular division of code into packages, classes, methods, functions, sub-routines, for example. These natural chunking tendencies are able to be exploited by mapping these over to a skeletal structure for literate representation. These types of chunks are largely related to the programming language itself, and are governed by the programming methodology employed by the programming language.

Just as auto-documenters create skeletal documentation sections for a language construct, so is it useful to apply a similar process when converting and developing source code. Indeed, when the following three factors are known the process of automatic translation between source code to a literate skeletal equivalent is possible, and relatively straight-forward:

1. the syntax of a particular programming language,
2. the semantic construct representation via the combination of key words,
and
3. which constructs are of relevance.

It is important to bear in mind that this is concerned with providing consistent and workable themes based upon a pre-determined template. The generated literate code from this operation will provide a workable and malleable set of literate programming chunks. The chunks created are not, by any means, the final chunked representation of the literate program. Nor is their initial order of representation. More chunks will be included with further development: chunks may be ‘chunked’ further and may be combined in an atomic relationship, or reordered to present a particular perspective. This phase adds a firm foundation from which further implementation can be performed.

By purposely chunking source code, the programmer’s ability to consider:

- the combination of chunks into multiple themes to represent ‘psychological ordering’ of chunks, or perspectives,
- the importance of the chunked unit, its relevance to the rest of the program, therefore facilitating ‘chunk-based perspective’, and
- the chunked unit as a separately documentable object

is enhanced.

Furthermore, although LP does increase abstraction towards the reader, the automatically chunked programming constructs of particular languages warrant particular attention. They are purposeful logical units that can be interfaced, implemented, or referenced. We must therefore explicitly recognise psychological scope. A theme should not detract from this fact.

F.2 Object Orientation

We use the Java programming language as a basis for the illustration of automated chunking. Note also that we will use the **Noweb** notation for the following examples because of its readable syntax. Each chunk created is assumed to contain a unique **chunkID**. Each chunk will also receive an initial **version** value of 1. Furthermore, a chunk's type is assumed to be of type **code**, however, the author can be more specific by using **Java-test-code**, for example. Each chunk's **name** is presented, and utilised as a chunk's reference, in the following literate examples.

To transform source code to the literate equivalent, we consider the following Java constructs warrant automatic chunking:

- package
- class
- method
- abstract
- interface
- try/catch

For example, the following Java code:

```
class Alpha {
    private int i;
    private void privateMethod() {
        System.out.println("publicMethod");
    }
}
```

can be converted to its literate equivalent by ‘chunking’ the class definition and class method:

```

<<class Alpha>>=
class Alpha {
    private int i;
    <<publicMethod>>
}

<<publicMethod>>=
    public void publicMethod() {
        System.out.println("publicMethod");
    }

```

F.2.1 Class-Level Chunking

The **Alpha** class definition has been chunked because it is an expression of a well formed and contained idea with a logical scope, very well suited to a literate code chunk, aptly enough.

F.2.2 Method-Level Chunking

The **publicMethod()** method has also been chunked because it represents a programmatically natural closure of thought. Rather than contain all methods in the confines of one chunk, we believe that the phenomenon of psychological scope dictates the need to distinctly ‘chunkify’ methods.

For example, several minor methods, most likely private to a particular class can be contained into one chunk (the naming of the chunk then becomes an issue):

```

class Alpha {
    private int i;
    <<publicMethod>>
    <<general purpose mathematic functions>>
}

<<publicMethod>>=
...
<<general purpose mathematic functions>>=

private void add (int j) {
    i += j;
}

private void subtract (int j) {
    i -= j;
}
...

```

However, this approach is not consistent. A reader wanting to know more of the methods contained in class **Alpha** may assume that all methods have been chunked. It may not be obvious, by perusing the content of the repository, which methods `<<general purpose mathematic functions>>` contains. The reader, knowing that the method `add()` exists may be lulled into wrongly looking for an `<<add>>` chunk.

Consistently recognising method-level scope also facilitates a granular theme composition, such that the inclusion and ordering of chunks is possible. Combining multiple methods under the umbrella of one chunk prevents this.

F.2.3 Abstract Class Chunking

Abstract classes in Java allow the development of template-like structures that are extendible by a given class. They are not directly instantiatable.

Methods implemented in an abstract class are inherited by the extending class. An abstract class' method, however, must be defined by the extending class. The outline of an abstract class follows:

```
abstract class GraphicObject {  
    int x, y;  
    . . .  
    void moveTo(int newX, int newY) {  
        . . .  
    }  
    abstract void draw();  
}
```

The makeup of of an abstract class is directly similar to a 'normal' class and should therefore be chunked.

Similarly, abstract methods should be chunked. They differ from normal methods because they do not receive direct implementation. An abstract method should still be singly chunked because:

- explanation via documentation is required. It is designed with a mindful intent of future implementation by each inheriting class. This intent should be documented.
- it promotes the composition of themes such that specifically group an abstract method and its implementations from inheriting classes, for example.

F.2.4 Interface Chunking

Interfaces define, but do not implement methods. They are not extendable by other classes, however may be implemented, in which case the implementing class must implement each method of an interface. A given class may implement more than one interface.

```

public interface Beta {
    final String word = "hello";
    void firstMethod(String partialWord);
    void secondMethod(String partialWord);
}

```

An interface is a device that unrelated objects use to interact with each other. It is a way of capturing similarities among unrelated classes without artificially forcing a class relationship.

Although not a class, an interface is an entity that has been similarly designed; a set of methods and data. And, for the same reasons as an abstract method construct's chunking, interface constructs are also chunked.

```

<<interface Beta>>=
public interface Beta {
    final String word = "hello";
    <<interface method firstMethod>>
    <<interface method secondMethod>>
}

<<interface method firstMethod>>=
    void firstMethod(String partialWord);

<<interface method secondMethod>>=
    void secondMethod(String partialWord);

```

F.2.5 Try/Catch Chunking

Error handling also deserves discrete chunking. **Exceptions** are specifically used to manage errors. They separate error handling content from 'regular' code and possess a strong similarity to chunk-based modularisation (effectively forcing chunk transparency — Appendix A.3 on page 241 discusses the

concept of literate programming transparency). Effectively, we treat try and catch statements as method level constructs.

The following example illustrates the result of chunking try/catch statements:

```
<<readFile>>=
public void parseFile() {
}
<<try to parse file>>=
    try {
        <<open the file>>
        <<determine number of lines>>
        <<allocate that much memory>>
        <<read the file into memory>>
        <<close the file>>
    }
<<catch memoryAllocationFailed>>=
    catch (memoryAllocationFailed) {
        ...
    }
<<catch readFailed>>=
    catch (readFailed) {
        ...
    }
<<catch writeFailed>>=
    catch (fileCloseFailed) {
        ...
    }
<<finally>>=
    finally {
        ...
    }
```

It is, thus, immediately obvious which errors are handled without obscur-

ing the central purpose of the method. Specific themes can thereafter be created to include these `catch` chunks in order to completely separate error handling from the method's core.

F.2.6 Sub-Method Level Chunking

We leave the discretion of sub-method level chunking to the discretion of TBLP authors because the chunking of artifacts such as looping constructs, for example, is a subjective process.

Further work must be performed to develop guidelines as to the chunking of class and method-level variable declarations, for example. It is likely that these declarations will require explicit chunking.

F.3 Imperative Languages

Imperative, or procedural languages, will, by their very nature, promote different thought processes by the programmer[73]. The structuring of information differs, and the semantic expressiveness of imperative languages compared to OO languages, for example, is different. In consideration of this, the following constructs are deemed as individually chunkable items:

- function definitions/declarations,
- `#define` macro definitions, and
- data structure constructs.

Functions are cohesive units, however differ from classes and methods in OO because there exists a less cohesive modularity between data and functions. Data may not be tightly associated within functions as it is with a class, for example. The concepts of data hiding are not prevalent in procedural languages.

Data structures central to a program's functionality are commonly defined as separate entities global to all functions, or functions within a particular file. If not global, pointers to these data structures are declared and passed

by reference to functions that need to make use of the data. These functions will commonly make use of, or transform, data available in these structures. This contrasts against the object oriented model whereby data and methods exist together as part of a particular entity — an encapsulated object: if a method has a direct influence on a set of data, that method will usually be contained within a class, along with the affected data.

In recognising the differences and similarities between the two paradigms, it is possible to translate these into rules to be applied to an automated literate chunking engine of an imperative language.

Whereas in OO, a hierarchical approach is taken in the chunking of program constructs i.e., package - class - method, imperative languages, like C, are more lateral in nature; they have data structure and function constructs. These two entities should be chunked.

More specifically, data structures in C are commonly created using three constructs:

structs: user-definable types.

unions: user-definable type whose members occupy the same space in memory.

enums: user-definable type whose members assume consecutive, unless otherwise stipulated, integer values.

Data structures are central to a good program's design. So much so that in imperative languages, the alteration of a data structure can render functions that are in some way linked, inoperable. The thought behind data structures can be expounded by promoting their careful documentation.

As an example of data structure chunking:

```

<<enum days>>
<<union earnings>>
<<employess struct>>

<<enum days>>=
enum {sun,mon,tues,wed,thur,fri,sat,sun} days;

<<union earnings>>=
union earnings{int wageperhour, double int weeklysalary};

<<employess struct>>=
struct employee {
    char    name[20];
    enum days offdays[2];
    union earnings income;
};

```

Functions in C, as recommended by the ANSI standard, should be declared as prototypes, and then defined separately. This ensures type-checking for return and parameter values. It also promotes forward thinking and design. It is common practice that external function prototypes are placed in a header file. Utilising TBLP's documentation reuse, the author is able to associate the documentation for the function with both the function prototype and the function definition.

Examples of function-level chunking are reflected in Section F.2.2 on page 359.

References

- [1] Cpan (comprehensive perl archive network). <http://www.cpan.org/>, 2002. Common repository for perl modules and documentation.
- [2] Lyx. <http://www.lyx.org/>, August 23 2002. A visual document processor that uses L^AT_EX.
- [3] Perlpod, perl plain old documentation. <http://www.perldoc.com/perl5.6.1/pod/perlpod.html>, 2002. Description of Perlpod utilisation.
- [4] BATTERY, D. Refinements and separation of concerns. Tech. rep., University of Texas at Austin, Department of Computer Sciences, 2000. Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000).
- [5] BECK, K. *Extreme Programming Explained : embrace change*. Addison-Wesley, 2000.
- [6] BENTLEY, J. Programming pearls — literate programming. *CACM* 29, 5 (May 1986), 364–369.
- [7] BO LEUF, W. C. *The Wiki Way: Collaboration and Sharing on the Internet*, 1st ed. Addison-Wesley Pub Co, April 3 2001.
- [8] BODANIS, D. *E=mc²*, 1st ed. Berkley Pub Group, October 9 2001.
- [9] BRIGGS, P. *Nuweb Version 0.92 A Simple Literate Programming Tool*. <http://sourceforge.net/projects/nuweb/>, Feb 1 2001.
- [10] BROOKS, R. E. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18, 6 (1983), 543–554.

- [11] BROWN, M. E., AND CHILDS, B. An interactive environment for literate programming. *Journal of Structured Programming* 11, 1 (1990), 11–25.
- [12] BROWN, M. E., AND CORDES, D. Literate programming applied to conventional software design. *Structured Programming* 3, 11 (1990), 85–98.
- [13] BROWN, M. E., AND CORDES, D. A literate programming design language. In *COMPEURO'90: Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering, May 8–10, 1990, Tel-Aviv, Israel* (Los Alamitos, CA, USA, 1990), IEEE CS Press, pp. 548–549.
- [14] BURDETT, P. S. Documentation: Effective and literate. In *Proceedings of the Fourth International Conference on Systems documentation* (Cornell University, Ithaca, New York, United States, 1986), SIGDOC : ACM Special Interest Group on Systems Documentation, ACM Press New York, NY, USA, pp. 110 – 113.
- [15] CANFORA, G., AND CIMITILLI, A. Program comprehension. 1998.
- [16] CAREY, V. J. litxml.shar. Internet document: <http://biosun1.harvard.edu/~carey/litxml.shar>, March 6 2001.
- [17] CHILDS, B. Literate programming, a practioner's view. In *TUGboat* (2001), vol. 13 of *Proceedings of the 2001 Annual Meeting*, Department of Computer Science, Texas A&M University, pp. 1001 – 1008.
- [18] COATES, A. xmlp - literate programming in xml. Internet document: <http://http://xmlp.sourceforge.net//>, August 11 2000.
- [19] COCKBURN, A. Supporting tailorable program visualisation through literate programming and fisheye views. <http://www.cosc.canterbury.ac.nz/~andy/papers/brow.pdf>, April 6 2000.

- [20] COCKBURN, A., AND CHURCHER, N. Towards literate tools for novice programmers. Tech. rep., Department of Computer Science, University of Canterbury, Christchurch, New Zealand, http://www.cosc.canterbury.ac.nz/research/reports/TechReps/tr_9705.pdf, May 1997.
- [21] CORDES, D., AND BROWN, M. The literate-programming paradigm. *Computer 24*, 6 (June 1991), 52–61.
- [22] DIDIER MARTIN, MARK BIRBECK, M. K. B. L. J. P. S. L. P. S. K. W. R. A. S. M. D. B. B. P., AND OZU., N. *Professional XML*, 1st ed. Wrox Press, 2000.
- [23] DRAKOS, N. L^AT_EX2HTML. <http://cbl.leeds.ac.uk/nikos/tex2html/doc/latex2html/latex2html.html>
- [24] EVANS, T. A meta-model for literate programming. Tech. rep., University of Canterbury, New Zealand, http://www.cosc.canterbury.ac.nz/research/reports/HonsReps/hons99_02.pdf, November 10 1999.
- [25] FOUNDATION, T. A. S. Xalan-c++ version 1.1. Internet document: <http://xml.apache.org/xalan-c/index.html>, 2000.
- [26] FOX, J. Webless literate programming. *TUGBOAT journal 11*, 4 (Nov. 1990), 511–513.
- [27] FRIENDLY, L. The design of distributed hyperlinked programming documentation. In *Proceedings of the International Workshop on Hypermedia Design* (Montpellier, France, June 1-2 1995), Springer, pp. 151 – 173.
- [28] FURNAS, G. W. Generalized fisheye views. In *Proc. of CHI-86* (Boston, MA, 1986), pp. 16–23.

- [29] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.
- [30] GARTNER, F. *The PretzelBook*, 2nd ed. Internet document: <http://www.informatik.tu-darmstadt.de/BS/Gaertner/pretzel/distribution/doc/pretzelbook/pretzelbook.dvi>, June 11 1998.
- [31] GERMAN, D. M., AND COWAN, D. Sgml-lite-an sgml-based programming environment for literate programming. In *Fourth International Symposium on Applied Corporate Computing* (Monterrey, Mexico, October - November 1996), pp. 175–184.
- [32] GURARI, E., AND WU, J. A WYSIWYG literate programming system: a preliminary report. Technical research report OSU-CISRC-7/90-TR17, Ohio State University, Computer and Information Science Research Center, Columbus, OH, USA, 1990.
- [33] HENRIK M MORGENSEN, K. R. T., AND VESTDAM, T. Object oriented documentation — construction and presentation using the docsewer tool. Master’s thesis, Aalborg University, Department of Computer Science, 1999.
- [34] HENRY, S., AND KAFURA, D. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering* (1981).
- [35] HURST, J. nutweb-1.12. Internet document: <http://www.csse.monash.edu.au/~%7Eajh/research/literate/nutweb-1.12.tar.gz>
- [36] HURST, J. Document technology interests - axe. Internet document: <http://www.csse.monash.edu.au/~ajh/research/doctech/index.htm>, April 12 2000.

- [37] HURST, J. John hurst's literate programming. Internet document: <http://www.csse.monash.edu.au/~%7Eajh/research/literate/>, August 11 2000.
- [38] JEFFREYS, C., AND HAEMER. Virtual threaded news reader. SunExpert, url: <http://www.alumni.caltech.edu/~copeland/work/thread2.html> date: and <http://www.literateprogramming.com/seaug98.pdf> date:, August 1998. news reader literate program.
- [39] KACOFEGITIS, A., AND CHURCHER, N. Theme-based literate programming. *APSEC 2002, 9th Asia-Pacific Software Engineering Conference* (November 4 2002). (accepted).
- [40] KAY, M. *XSLT Programmer's Reference*, 2nd ed. Wrox Press, 2001.
- [41] KERNIGHAN, B. W. Why pascal is not my favorite programming language, April 2 1981.
- [42] KERNIGHAN, B. W., AND PIKE, R. *The Practice of Programming*. Addison-Wesley, 1999. effective programming.
- [43] KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds., vol. 1241. Springer-Verlag, Berlin, Heidelberg, and New York, 1997, pp. 220–242.
- [44] KNUTH, D. E. The web system of structured documentation. Tech. rep., Stanford Computer Science Report CS980, Stanford University, Stanford, CA, September 1983.
- [45] KNUTH, D. E. Literate programming. *The Computer Journal* 27, 2 (1984), 97–111.

- [46] KNUTH, D. E. *T_EX : The Program (Computers & Typesetting ; B.* Addison-Wesley Publishing Company, Stanford Univ., Stanford, CA, 1986.
- [47] KNUTH, D. E. *The METAFONT book.* Addison-Wesley Publishing Company, Stanford University., Stanford, CA, 1989.
- [48] KNUTH, D. E. *Literate Programming.* CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.
- [49] comp.programming.literate faq. `news:comp.programming.literate` <http://www.faqs.org/faqs/literate-programming-faq/>, March 15 2000. literate programming mailing list.
- [50] MCDUGALL, S. Program pods. http://world.std.com/~swmcd/steven/perl/program_pod.html, June 2 1997. `perlpod`, `styleguide`, `perl`.
- [51] MENS, K. Multiple cross-cutting architectural views. Tech. rep., Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussel, Belgium, February 21 2000.
- [52] MICROSYSTEMS, S. Doclet overview. <http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/overview.html>, 1998.
- [53] MICROSYSTEMS, S. How to write doc comments for the javadoc tool. <http://java.sun.com/j2se/javadoc/writingdoccomments/>, 2000.
- [54] MUSSER, D. R. Text-line random shuffling program. url: <http://www.literateprogramming.com/rand.pdf> date: , September 28 2000. literate program using Nuweb.
- [55] NORMAN WALSH, L. M. *DocBook: The Definitive Guide.* O'Reilly, October 1999.

- [56] N, K. Requirements for an elucidative programming environment, June 2000.
- [57] OSSHER, H., AND TARR, P. Multi-dimensional separation of concerns and the hyperspace approach, 2000.
- [58] , K. Literate smalltalk programming using hypertext. *IEEE Transactions on Software Engineering* 21, 2 (1995), 138 – 145.
- [59] P. TARR, H. OSSHER, W. H., AND S.M. SUTTON, J. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering* (May 1999).
- [60] PAGE-JONES, M. *The practical guide to structured systems design*.
- [61] PARIKH, G., AND ZVEGINTZOV, N. Tutorial on software main-tenance. Silver Springs, MD: IEEE Computer Society, 1983.
- [62] PARNAS, D. Software aging. *Proceedings of the 16th International Conference on Software Engineering, IEEE Press* (May 1994), 279–287.
- [63] PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 12 (1972), 1053–1058.
- [64] PRESSMAN, R. S. *Software Engineering: A Practitioner’s Approach*, fifth ed. McGraw-Hill, New York, 2000.
- [65] RAMSEY, N. *A Spider User’s Guide*. Department of Computer Science, Princeton University, Internet document: <http://www.literateprogramming.com/spider.pdf>, July 1989.
- [66] RAMSEY, N. The noweb hacker’s guide. Tech. rep., Department of Computer Science, Princeton University, September 1992.
- [67] RAMSEY, N. Literate-programming can be simple and extensible. <http://tex.loria.fr/litte/ieee.pdf>, October 1993.

- [68] RAMSEY, N. Literate programming simplified. *IEEE Software Engineering* 11, 5 (sep 1994), 97–105.
- [69] RAMSEY, N. Weaving a language independent web. *Communications of the ACM* 32, 9 (September 1989 1995), 1051—1055.
- [70] RAMSEY, N. Noweb. <http://www.eecs.harvard.edu/~nr/noweb/>, July 2001. Noweb’s home page and latest news.
- [71] RAMSEY, N., AND MARCEAU, C. Literate programming on a team project. *Software, Practice and Experience* 21, 7 (1991), 677–684.
- [72] REAM, E. K. Leo’s home page, August 4 2001. Home page and download site.
- [73] RECHENBERG, P. Programming languages as thought models. *Structured Programming* 3, 11 (1990), 105–115.
- [74] REENSKAUG, T., AND SKAAR, A. L. An environment for literate Smalltalk programming. *SIGPLAN* 24, 10 (Oct. 1989), 337–345.
- [75] RYMAN, A. Foundations of 4thought. *Proceedings of the 1992 CAS Conference, Toronto, Ontario* (November 1992), 133–155.
- [76] SEVILLA, R. xml-lit. <http://xml-lit.sourceforge.net/>, August 2001. XML-based literate programming tool.
- [77] SEWELL, E. W. *Weaving a Program: Literate Programming in WEB*. Van Nostrand Reinhold, New York, NY, 1989.
- [78] SHNEIDERMAN, B. Direct manipulation, 1992.
- [79] SHUM, S., AND COOK, C. Using literate programming to teach good programming practices.
- [80] SMITH, M. Modernising literate programming. Tech. rep., University of Canterbury, 2001.

- [81] SMITH, S., BARNARD, D., AND MACLEOD, I. Holophrasted displays in an interactive environment. *International Journal of Man Machine Studies* 20, 4 (Apr. 1984), 343–355.
- [82] SUSANTI. Documentation definition in xml. Internet document: <http://www.csse.monash.edu.au/hons/projects/2000/Susanti/THESIS.doc>, November 2000.
- [83] TANENBAUM, A. S. *Modern Operating Systems*, 2nd ed. Prentice-Hall Inc, February 28 2001.
- [84] TICHY, W. F. RCS - a system for version control. *Software - Practice and Experience* 15, 7 (1985), 637–654.
- [85] TILLEY, S. R. Documenting-in-the-large vs. documenting-in-the-small. In the Proceedings of *CASCON '93*, (Toronto, Ontario) (October 1993), pp. 1083–1090.
- [86] VAN AMMERS, E. W. Clip, a universal literate programming tool. Internet document: http://www.literateprogramming.com/clip_ann.pdf, Feb 24 1993.
- [87] VAN AMMERS, E. W., AND KRAMER, M. R. The clip style of literate programming, Feb 26 1993.
- [88] VESTDAM, T. Pulling threads through documentation. Tech. rep., Aalborg University, Department of Computer Science, Fredrik Bajers Vej 7E, DK-9220 Aalborg, Denmark.
- [89] VIVTEK. xml-lit. <http://www.vivtek.com/lpml/>, July 2000. XML-based literate programming tool.
- [90] WILLIAMS, R. *FunnelWeb*. <http://www.ross.net/funnelweb/>, 1992.
- [91] WITH KARL BERRY, P. W. A., AND HARGREAVES, K. A. *T_EX for the Impatient*. Addison-Wesley, Reading, MA, USA, 1990. T_EX.

- [92] WROTH, M. An experiment in literate programming using sgml and dsssl. Internet Document: <http://ourworld.compuserve.com/homepages/markwroth/LitProg/SGMLWEB/experiment.pdf>, December 31 1999.
- [93] WROTH, M. Dblp: Docbook-based literate programming. Internet Document: <http://www.west-point.org/users/usma1978/36200/LitProg/SGMLWEB/dblp.htm> or <http://www.west-point.org/users/usma1978/36200/LitProg/SGMLWEB/DBLP.pdf>, April 12 2001.