

Structured Writing

Theory and Practice

Mark Baker

Structured Writing: Theory and Practice

Mark Baker

Printed in the United States of America.

Copyright © 2017 Mark Baker

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means without the prior written permission of the copyright holder, except for the inclusion of brief quotations in a review.

Credits

Cover Image:

Cover Background:

Foreword:

Disclaimer

The information in this book is provided on an “as is” basis, without warranty. While every effort has been taken by the author and XML Press in the preparation of this book, the author and XML Press shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

This book contains links to third-party web sites that are not under the control of the author or XML Press. The author and XML Press are not responsible for the content of any linked site. Inclusion of a link in this book does not imply that the author or XML Press endorses or accepts any responsibility for the content of that third-party site.

Trademarks

XML Press and the XML Press logo are trademarks of XML Press.

All terms mentioned in this book that are known to be trademarks or service marks have been capitalized as appropriate. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Table of Contents

1. Why structure writing?	1
I. Domains	6
2. How ideas become content	8
From ideas to dots	8
The three domains	11
3. Writing in the Media Domain	15
Partitioning font information	17
Partitioning the complexity of pagination	20
4. Writing in the Document Domain	21
Extensibility	23
Context and Structure	24
Constraining document structure	26
Extending the document domain	28
5. Writing in the Subject Domain	30
The role of the subject domain	35
Using subjects to establish context	36
6. The Management Domain: an Intrusion	37
Including boilerplate content	37
An alternative approach in the subject domain	39
Hybrid approaches	41
II. Algorithms	43
7. Quality in Structured Writing	48
Robots that read	48
Dumbing it down for the robots	48
Making humans better writers	49
We write better for robots than we do for humans	49
Structure, art, and science	51
Contra-structured content	52
Until the robots take over	52
Other sources of quality improvement	53
8. Writing	54
Functional lucidity	57
Simplicity and Clarity	57
9. Separating Content from Formatting	60
Separate out style instructions	60
Separate out formatting characters	61
Name your abstractions correctly	62
Make sure you have the right set of abstractions	62
Create containers to provide context	64
Move the abstractions to the containers	65
Separate out abstract formatting	66
10. Processing Structured Text	69

Two into one: reversing the factoring out of invariants	69
Adding back style information	69
Rules based on structures	71
The order of the rules does not matter	71
Applying rules in the document domain	72
Processing based on context	73
Processing container structures	75
Restoring factored-out text	75
Processing in multiple steps	77
Query-based processing	79
11. Single Sourcing	81
Basic single sourcing	81
Differential single sourcing	85
Differential organization and presentation	88
Conditional differential design	89
Primary and secondary media	90
Responsive Design	91
12. Reuse	93
Fitting pieces of content together	94
Common into variable	94
Variable into common	96
Variable into variable	97
Using IDs	98
Using keys	99
Common with conditions	101
Factor out the common	104
Factor out the variable	105
Assemble from pieces	106
Combining multiple techniques	108
Content reuse is not a panacea	109
Quality traps	109
Cost traps	110
Alternatives to reuse	112
13. Composition	114
Fundamental composability	115
Structural composability	115
Stylistic composability	116
Narrative composability	116
14. Normalization	118
The limits of content normalization	121
Examples of content normalization	122
Normalization and discovery	123
Proximity detection	124
Normalization via aggregation	125
15. Linking	126

Deflection in the media domain	127
Deflection in the document domain	128
Deflection in the management domain	129
The problem with IDs and Keys	130
Relationship tables	130
The problem with relationship tables	131
Conditional linking	132
Deflection in the subject domain	133
Finding resources to link to	136
Deferred Deflection	138
Different domain, different algorithm	140
16. Conformance	142
Completeness	145
Consistency	145
Accuracy	146
Semantic constraints	146
Entry validation constraints	149
Referential integrity constraints	150
Conformance to external sources	150
Conformance and change	152
Design for conformance	152
17. Auditing	155
Correctness of the definition of the content set	156
Bottom-up content planning	157
Ensuring the content set remains uncontaminated	157
Ensuring that the content set is well integrated	158
Making content auditable	159
Performance auditing	160
18. Relevance	162
Being relevant	162
Showing relevance	163
Showing relevance to algorithms	164
19. Extract	167
Tapping external sources of content	167
Information created for other purposes	168
The diversity of sources	171
20. Merge	173
21. Information Architecture	176
Top-down vs. bottom-up information architecture	178
Categorization	178
Linking	179
Tables of Contents	180
Lists	181
Personalized content	182
22. Change management	184

Content management systems	189
23. Content Management	191
Metadata is the foundation of management	192
The location of metadata	193
Managing the process	195
Conflicting constraints	196
Creating manageable content	198
24. Collaboration	200
Bridging silos	203
25. Timeliness	204
26. Repeatability	211
27. Publishing	214
The Rendering Algorithm	214
The Encoding Algorithm	215
The Formatting Algorithm	217
The Presentation Algorithm	218
Differential presentation algorithms	219
Presentation sub-algorithms	219
The linking algorithm	219
The information architecture algorithm	219
The Synthesis Algorithm	220
Differential synthesis	220
Synthesis sub-algorithms	221
The reuse algorithm	221
The extraction algorithm	221
The merge algorithm	221
Deferred synthesis	221
Combining algorithms	221
Architecture of a publishing tool chain	222
28. Active content	225
29. Translation	230
Extracting content for translation	230
Avoiding trivial differences	230
Isolating content that has changed	231
Continuous translation	231
III. Structures	232
30. Rhetorical Structure	234
Presentation-oriented rhetorical structures	239
Rhetorical metamodels	241
Meta-models vs generic models	243
Making the rhetorical structure explicit	244
Structure and repeatability	245
The importance of the rhetorical model	247
31. Mechanical Structure	248
Flat vs. nested structures	249

Annotating blocks	252
Agreeing on names	252
Different rules for intermediate languages	253
Secondary structures of interpretation	254
Child blocks vs. additional annotations	256
32. Blocks, fragments, paragraphs, and phrases	262
Semantic blocks	262
Information typing blocks	265
Rhetorical blocks	267
Granularity	270
Fragments	270
Paragraphs and phrases	271
33. Wide Structures	274
Tables	274
Alternatives to tables	279
Alternate presentation	279
Subject domain structure	279
Record data as data	280
Code	280
Pictures and graphics	281
Inline graphics	288
34. Subject domain structures	290
Keep it simple and lucid	296
35. Metadata	298
The recursive nature of metadata	299
Where should metadata live?	300
Ontology	302
36. Terminology	304
Top-down vs. bottom-up terminology control	305
IV. Languages	311
37. Markup	313
Markup vs. regular text	314
Markup languages	317
Concrete markup languages	317
Abstract markup languages	317
Instances of abstract markup languages	318
Concrete languages in abstract clothing	319
The ability to extend	320
The ability to constrain	321
Showing and hiding structure	321
Hybrid languages	322
Instances of hybrid markup languages	325
38. Patterns	326
Design implications	327
39. Lightweight Languages	332

Markdown	333
Wiki markup	334
reStructuredText	335
ASCIIDoc	336
LaTeX	337
Subject Domain Languages	340
40. Heavyweight markup languages	341
DITA	342
DocBook	346
S1000D	347
HTML	347
Subject domain languages	348
41. Extensible and Constrainable Languages	349
XML	349
DITA	350
Specializing between domains	354
DocBook	356
RestructuredText	357
TeX	357
SAM	358
SPFE	360
42. Constraint Languages	362
V. Design	365
43. System design	367
Identifying complexity	368
Current pain points	368
Unrealized possibilities	369
What's working now	370
Complexity that impacts others	370
Process overheads	371
Your style guide	372
Change management issues	372
What's coming down the road	373
Partition and direct complexity	374
Focus on complexity, not effort	376
Indivisible complexity	377
Subtract current tool complexity	377
Avoid tool filters	377
Select domains	378
Select tools	383
Count the costs and savings	384
Index	386
A. Copyright and Legal Notices	388

List of Figures

3.1. Objects vs. dots	17
3.2. A vector graphics text object	18
3.3. Merging text and font information	19
33.1. Broken table formatting	275
37.1. Printer's markup	313

Chapter 1. Why structure writing?

Why would we apply structure to writing? Creating content is a complex business. You have to research your subject matter and your audience. You have to figure out what needs to be said and how to say it to achieve your desired effect. You then have to record the content, manage it, and publish it, potentially to many different audiences and many different media. If you are part of an organization that creates a lot of content, you have to coordinate what you are doing with what everyone else is doing and make sure it all works together and serves the organization's goals. Each of these things are complex tasks in themselves and they become all the more complex when you have to coordinate them all. Not everyone has the skills or the knowledge to do all these things well, and if too much of the complexity falls on one person's shoulders, it will limit their ability to perform well or deliver on time. In particular, writing well is an activity that requires the whole of the writer's attention. Any part of the complexity of the content creation process that divides the writer's attention makes it more difficult for them to write quality content.

You can't reduce the fundamental complexity of content creation. All that stuff has to get done, and it is all difficult in one way or another. All you can do is to make sure that the inherent complexity is handled by the people or processes that are best equipped to deal with it. Managing a content creation process is all about partitioning and distributing the complexity of content creation in the right way.

Every content organization practices the partitioning and distribution of content creation. Newspapers do not make reporters drive the delivery trucks. Publishing houses do not let authors design book covers (much as they might want to). Recognizing that spotting your own mistakes is difficult, wise organizations partition the job of editing and proofreading content from the job of writing it.

What happens if the inherent complexity of content creation is not managed properly? Complexity cannot be destroyed, so it has to go somewhere. If no one in the publishing organization handles it, it falls through to the reader. If a piece of content is missing information the reader needs, then the burden of finding the missing information falls on the reader. If the reader can't understand the terms used, or read the font chosen, or find content to explain an unfamiliar concept, or get the information they need when they need it, or tell current information from outdated, that is complexity falling through the content creation process to land on the reader's head. The reader must then either handle the complexity themselves or abandon the content.

Another name for this is poor quality content. Poor quality content is the direct result of the complexity of content creation falling through to reader instead of being handled in the organization that produced it. The way for an organization to consistently create quality content is to make sure that every piece of the inherent complexity of content creation is directed to a competent and adequately resourced person or process so that none of it falls through to the reader.

To distribute complexity, you must partition it along clean lines without dropping any of the complexity in the process. For example, the style sheets in a word processor partition the complexity of consistently formatting a document by allowing writers to apply named styles to paragraphs. This partitions the task of designing the look of a document, and defining styles to implement that design, and directs it to the stylesheet designer. The writer is left with the nominally easier job of assigning named styles to paragraphs as they write. (I say nominally because, as we know, styles are rarely used correctly, or at all. Mere technical partitioning often fails because it does not fully encapsulate the complexity it is attempting to partition.)

Partitioning and redistributing complexity is not just about shifting complexity from one person to another. It is also about shifting complexity from people to algorithms. Most complex tasks have repeatable elements in them. A design question that has been settled once can be implemented over and over again without having to redo the design work. If one piece of content is formatted a certain way, chances are many similar pieces can be formatted the same way. Algorithms are great at doing the same task the same way over and over.

Writing an algorithm to do a repetitive task redistributes complexity from the person who used to do the task to the algorithm, but also to the person who writes and maintains the algorithm. For example, using a CSS stylesheet to format lists redistributes the task of designing list formatting from the writers, whose job is to know what they are talking about and how to say it, to a publications designer, whose job it is to know how to attractively format lists and how to code good CSS.

Using CSS partitions the complexity of formatting a page so that we can distribute the complexity of formatting away from the writer. Now the writer does not have to know about formatting or design, and the designer does not have to know about writing or the subject matter. This simplifies the writer's life, allowing them to focus on writing. It also simplifies the designers life, allowing them to concentrate on design. The result is better writing and better, more consistent, design.

CSS does an effective job of partitioning the complexity of formatting, but CSS is itself complex. Introducing CSS in to our process means introducing new complexity. All tools introduce new complexity into the processes they serve. People have to design the tools and build the tool and learn to use the tool, and maintain the tool. All this is new complexity that did not exist before. We accept this new complexity into our processes because a good tool allows us to distribute our existing complexity more efficiently, so that less of the complexity falls on the people and more on the tools, and so that each person in the system only has to deal with the complexity that they are able to cope with. Managing where the complexity falls is ultimately much more effective than merely minimizing total complexity.

Adding complexity to the system in order to distribute complexity better is pervasive today. The complexity of the technology stack that lets you send an email or a text message is staggering, but its directs the complexity of communicating over distance

away from the users towards systems and the people who maintain them for a huge net gain in productivity.

The content creation stack is similarly complex. Yet the content creation stack does not do such a good job of partitioning and distributing all of the complexity of content creation, with the result that complexity is pervasively being dumped on the reader in the form of poor quality content.

There is some very elaborate partitioning and distribution going on in the content creation stack, in the form of complex content management and publishing systems, as well as new roles for human contributors, such as information architects and content strategists. However, most of the content creation going on in the content stack today is taking place using WYSIWYG authoring interfaces from the desktop publishing era.

In its time, desktop publishing represented an interesting repartitioning and redistribution of the complexity of content creation. At the time, creating a formatted printed document generally meant handing off a manuscript from an author to a typesetter for re-keying, followed by mechanical pasteup by a page-layout artist, the preparation of proofs by a printer, correction of proofs, and the final printing. This was a complex and time consuming process that cost a lot of money and had many points where failure could occur or error be introduced. The various function were well partitioned from each other, but the overhead of passing information from one to another was cumbersome and time consuming.

Desktop publishing eliminated much of that overhead by putting all the tools of document creation – writing, design, layout, proofing – in the hands of a single operator. The problem with this model are twofold. First, it put too much of the complexity on one person: the writer. The writer’s attention is divided between multiple tasks, and since attention is a finite resource, this meant the quality of writing, design, layout, and proofing all suffered. The fact that desktop publishing skill became a major hiring criteria for writers (and remains so today) shows how the focus was taken off writing and knowledge of subject matter and placed on the ability to manage the complexity of the publishing process.

Secondly, while it removed a lot of complexity of vertical communication between writer, designer, and typesetter, desktop publishing left every writer on an island, making no provision for any kind of horizontal coordination between writers. Everyone worked on their own book as a separate project. The division of the writer’s attention and the lack of horizontal coordination meant the huge amounts of complexity were going unhandled in large content systems. Duplication omissions, and inconsistencies were very difficult to detect and fix, while providing effective navigation between books was virtually impossible. All this unhandled complexity was dumped on the reader in the form of poor quality content. Desktop publishing did not create this problem, but it did nothing to fix it.

Content management system have tried very hard to partition and redirect the complexity of that horizontal coordination, but with results are are, at best, incomplete. Meanwhile,

the advent of the Web had brought a new set of challenges. A modern website is not a library of independent volumes but a complex hypertext consisting of many smaller pieces of content related in much more complex ways than paper documents ever were. Search engines and social networks have profoundly changed how readers seek and use content. Meanwhile, many organizations are trying to deliver content to both the paper model and the Web model simultaneously. Content creation and delivery has got a lot more complex and reader's expectations far more demanding. The desktop publishing model is fundamentally unsuited to handling this complexity.

Structured writing offers a different way to partition and distribute the complexity of content creation and delivery that better address the modern content creation challenge. The fundamentals of these techniques are not new. In fact, they all exist in the desktop publishing paradigm. When we define a stylesheet in FrameMaker. When we use CSS to separate formatting from our HTML content, we are using structured writing techniques. What we call structured writing today is simply about taking those techniques one or two steps further to allow us to better partition and distribute all of the complexity of content creation in organizations.

Alas, not every structured writing system does a good job of distributing complexity appropriately for every organization. Many merely move it from the shoulders of one group to the shoulders of another without regard to whether those shoulders are broad enough to bear the additional burden. Far too often, content management installations are designed and implemented with an eye to solving one part of the content management complexity, rather than with an eye to how content complexity is distributed and handled throughout your whole content system. Tools and systems designed for one purpose are almost always careless about where they deflect the complexity they don't handle. In many cases, while they take the pressure off one group or function, they add to the total amount of unhandled complexity in the system. This is a big part of why so many content management systems fail and so many organizations hate their content management system.

This book seeks to provide a comprehensive overview of how different structured writing techniques can be used to partition and distribute the complexity of content creation. It does not advocate for any one system. Nor will it tell you exactly which combination of techniques and tools will be right for your organization. Different organizations have different kinds of complexity to deal with and need to distribute it in different ways to maximize the quality of the content they produce. Finding the right combination of techniques for your organization is up to you. Hopefully, though, this book will give you the information you need to figure out what will work best, whether that means adopting or adapting an existing system or building all or part of a system for yourself.

Most content systems have never seriously attempted to minimize the amount of complexity they dump on the reader. Quality and process have largely been treated as separate concerns, as if the process and its demands had no influence, for good or bad, on the quality of content that writers produced.

The idea that you can build quality into a process by managing the complexity of the design and production process, though well accepted in other fields, has not been widely considered in content. To a large extent, where we have sought process improvements, they have been in the areas of publishing and content management, not in content itself. Where structured writing tools have replaced desktop publishing or word processing tools it has largely been for process goals. But a more holistic view would show that process and quality are intimately related, and the treating them holistically can significantly improve both.

Part I. Domains

Table of Contents

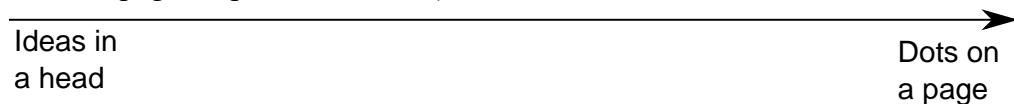
2. How ideas become content	8
From ideas to dots	8
The three domains	11
3. Writing in the Media Domain	15
Partitioning font information	17
Partitioning the complexity of pagination	20
4. Writing in the Document Domain	21
Extensibility	23
Context and Structure	24
Constraining document structure	26
Extending the document domain	28
5. Writing in the Subject Domain	30
The role of the subject domain	35
Using subjects to establish context	36
6. The Management Domain: an Intrusion	37
Including boilerplate content	37
An alternative approach in the subject domain	39
Hybrid approaches	41

Chapter 2. How ideas become content

An efficient structured writing system must consider the whole content creation process in order to distribute complexity appropriately without overloading anybody or letting any of the complexity slip through to fall on the reader. Let's start, therefore, by looking at how content gets from ideas in a head to dots on a page or screen.

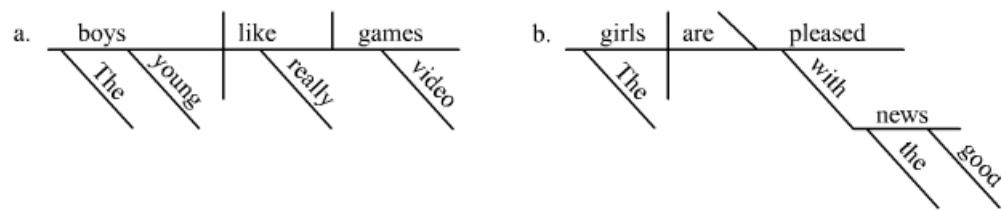
From ideas to dots

The process of creating and delivering content consists of translating ideas (stuff someone thinks or knows) into concrete physical form that can be read (dots or lines of ink on a page, or pixels on screen).



Structured writing is about applying a structured methodology to that process. It is a long road from ideas to dots, and structured writing techniques can be applied at many points along that road. Almost all writing done today uses structured writing techniques to one extent or another. As this book will show, the basic principles of structured writing apply across the spectrum, from the tools and techniques used in most offices today, to most sophisticated structured writing systems. It is not a matter of unstructured vs structured approaches, but the type of structures used.

All writing has structure in the literary sense of the word. Every comprehensible sentence has a grammatical structure. You may even have learned to diagram that structure in school.¹

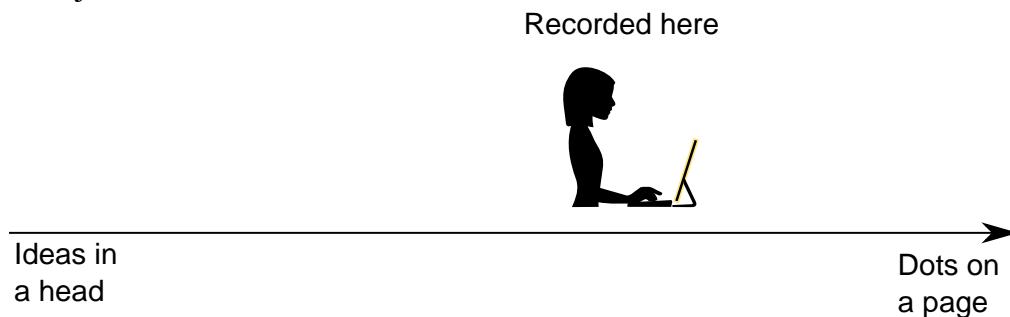


Just as all writing has at least a basic grammatical structure, all writing on a computer involves creating basic data structures. Thus the only case in which no structured writing techniques are involved in the writing process is when an author writes down their ideas with pen and paper, and gives that paper directly to the reader. In this case, the entire writing process, from an idea in the writer's head to words on paper takes place in the writer's head and any constraints that are imposed on the content are captured and imposed by the writer themselves as they inscribe their words with the pen.

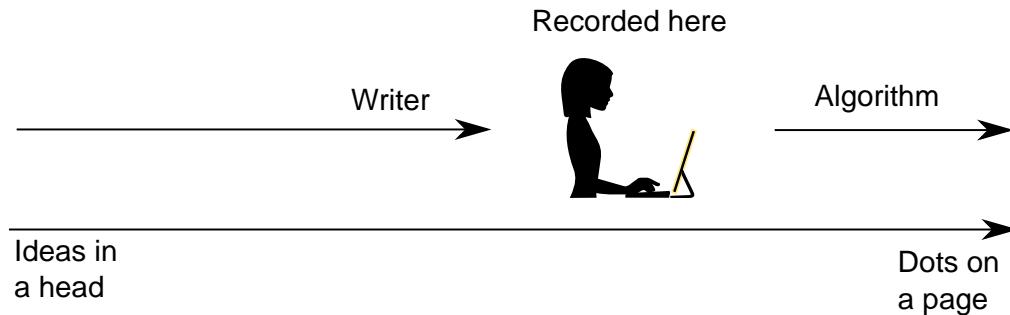
¹By Tjo3ya - Own work, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=18312612>



It is rare for writers to record their ideas directly in the final physical form these days. For instance, the writer may write in a word processor, edit the text on screen, and press Print to send the content to a printer to create the final form that the reader receives. Or they may press Send and have the final form rendered on someone else's monitor. The point at which the content is recorded, in the journey from ideas to dots, has been moved back just a little bit.



Word processing, desktop publishing, and various approaches to structured writing all establish a point between ideas and dots where the content will be recorded, and then provide algorithms to complete the journey from that point to dots on a page.



I'm going to use the word algorithm a lot in this book. An algorithm is a formalized and consistent way to do something. Basically, if you give an algorithm the same inputs, it should always produce the same outputs every time it is run. Computer software is an encoding of an algorithm that a computer can execute. A program describes an algorithm to a computer but you do not need to be a programmer to design an algorithm.

Algorithms are fundamental to structured writing. Algorithms and structures work together and you can't design one without the other. The reason you add structure is

to enable algorithms. The heart of this book is a description of the principal structured writing algorithms and the structures that support them. The structures exist to support the algorithms by constraining the content.

Algorithms apply rules to data that follows rules. If the data does not follow the rules, then the rules applied by the algorithms will not work. If you want algorithms to process your content, therefore, it has to follow the rules laid down for input to that Algorithm. If you want a browser to display your document, it needs to follow the rules of HTML. Structures constrain content to make sure it obeys the rules that the algorithms expect it to obey. But it is not sufficient to obey the rules, you also have to explicitly record that you have obeyed them. The algorithm uses that explicit recording to interpret the content so it can process it.

This leads us to a working definition of structured writing for purposes of this book:

Structured writing is writing that follows a set of constraints and records that it has done so.

For instance, lets say that we have a constraint that every section must start with a title. We can follow this constraint in just about any tool we choose. We could even follow in when writing by hand on paper. But if we choose a structured writing format, such as DocBook, we can not only follow this constraint but record that we have followed it:

```
<section>
    <title>The quick brown fox</title>
    <para>The quick brown fox jumps over the lazy dog.</para>
</section>
```

This markup records that we have followed the constraint by identifying the title as the first thing in the section. This constrains the interpretation of the string “The quick brown fox” by algorithms. Algorithms can now do things such as:

- Check that all sections start with titles (that is, check that the constraint has been followed)
- Format all section titles the same way
- Create a list of all the section titles in the document (perhaps to create a table of contents)

Recording the constraints that content follows is key to untangling and partitioning the complexity of content creation because it allows information to be passed reliably from one person or process to another.

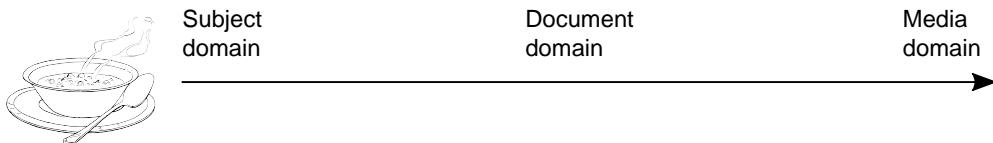
Generally speaking, the more we constrain the creation and interpretation of content with structure, the more accessible it becomes to algorithms, and the more opportunities we

gain to distribute complexity effectively. This means recording the content at a point nearer to ideas in the head and further from dots on a page. The nearer we are to ideas, the more we know about what the content says as opposed to what a page looks like, and the more effectively we can constrain the creation and interpretation of the text.

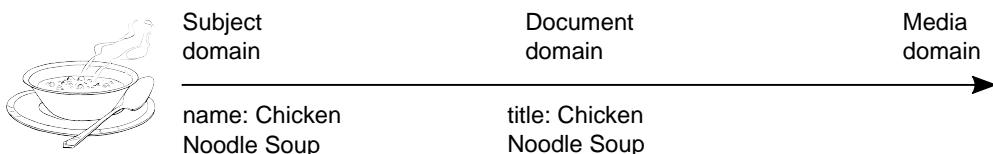
We can describe this process of earlier recording in terms of three domains, each domain reflecting a stage in the progress from ideas to dots. The domains are the media domain (which is concerned with lines and dots on paper or screen), the document domain (which is concerned with the expression and organization of documents), and the subject domain (which is concerned with the ideas that we write about).

The three domains

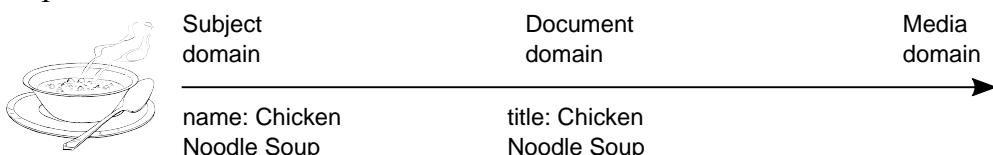
Let's suppose that an author is planning to write a recipe for chicken noodle soup. They start out with the idea of a soup made with chicken and noodles. This is an idea about the subject matter and not yet any form of content.



They then decide to give the dish the name “Chicken Noodle Soup.” They figure out which ingredients they want to use and how they want to make the dish. This is all information about making chicken noodle soup but it is not yet part of a document. It is information in the subject domain.

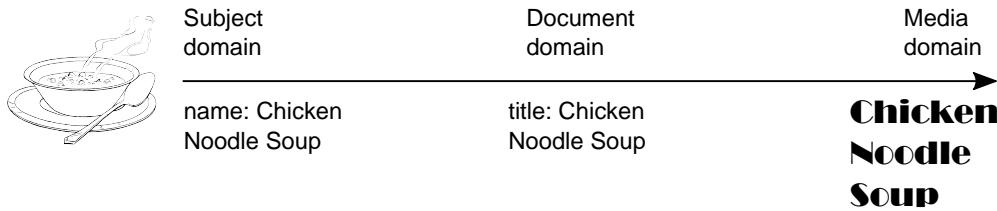


Then the author decides how they want to present this information to help other people make Chicken Noodle Soup. They decide they want to have a title, a picture, an introduction, a list of ingredients, and a set of preparation steps. They are no longer gathering information; now they are focused on how to present the information they have gathered. These are decisions in the document domain. Documents are how we organize and present information.



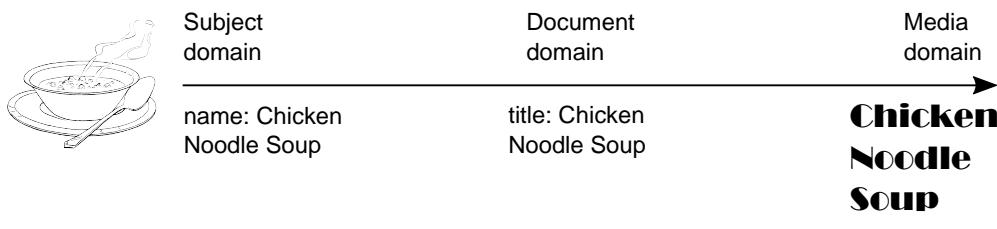
Then the writer starts to think about how the document will look on screen or on paper. What font will be used for the heading and the body text. How large will the heading and the body be. Will the quantity of the ingredients be flush right? Will there be leading dots?

Will the presentation steps be numberer or just presented sequentially? How big will the picture be? Will the text wrap around it? These are decision in the media domain.

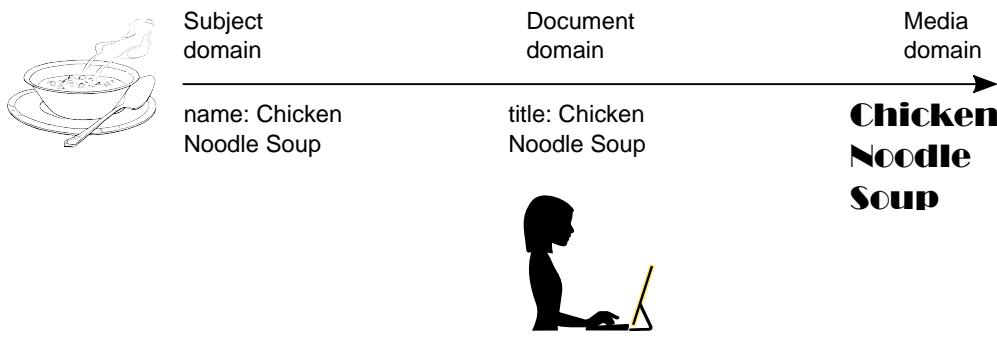


All content passes through the three domains. Content always begins with the author thinking about subjects in the real world. They then decide to express ideas about those subjects in words. They collect their ideas together and determine an order and structure to express them. Finally, they decide how they will be formatted in a particular media. The question is, where in this process does the author start recording the content?

Do they lay it all out in the formatted version as they write? Then they are working in the media domain.

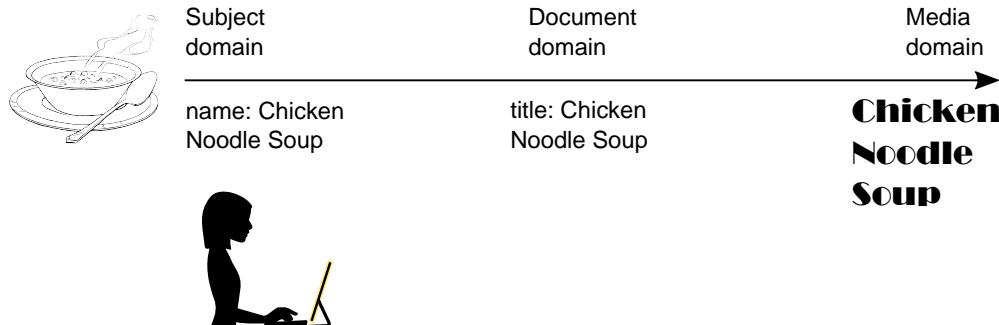


Do they record the presentation units like lists, heading, and step without associating specific formatting to them? Then they are working in the document domain.

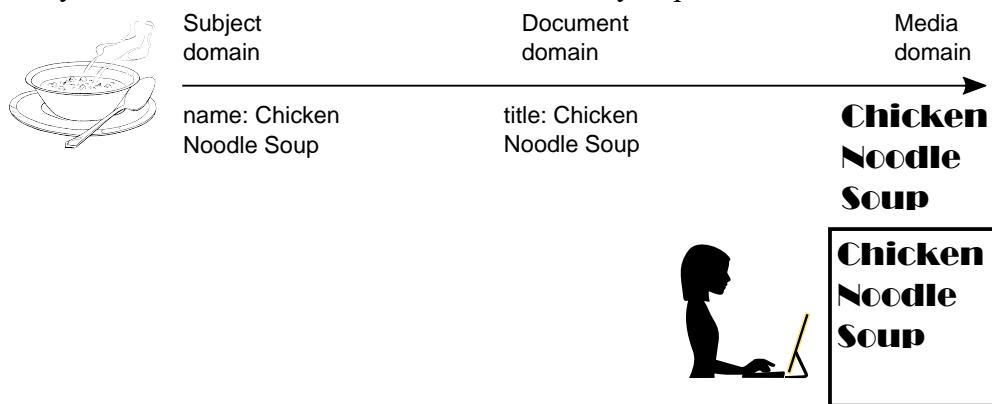


Do they record the raw information as data, for instance, recording each ingredient and its quantity as ingredients and quantities, rather than as list items? They they are working in the subject domain.

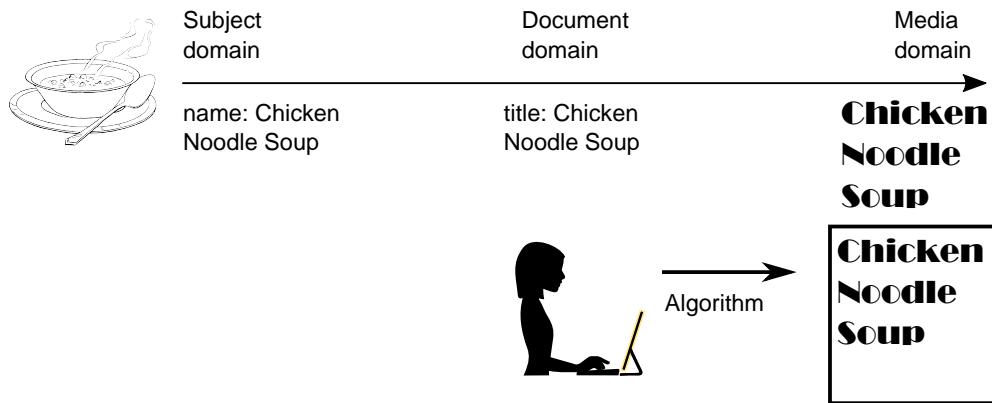
How ideas become content



If they recorded it in the media domain, it is ready to publish.

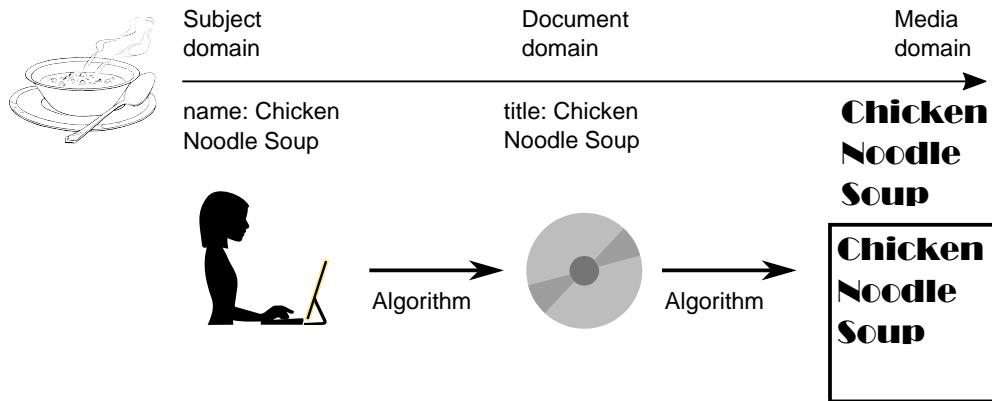


If they recorded it in the document domain, then it needs to be formatted before it can be published (and could potentially be formatted differently for different media or different publications).



If they recorded it in the subject domain, it needs to be organized into a document (and then formatted) before it can be published (but can potentially be organized into different kinds or documents or different collections).

How ideas become content

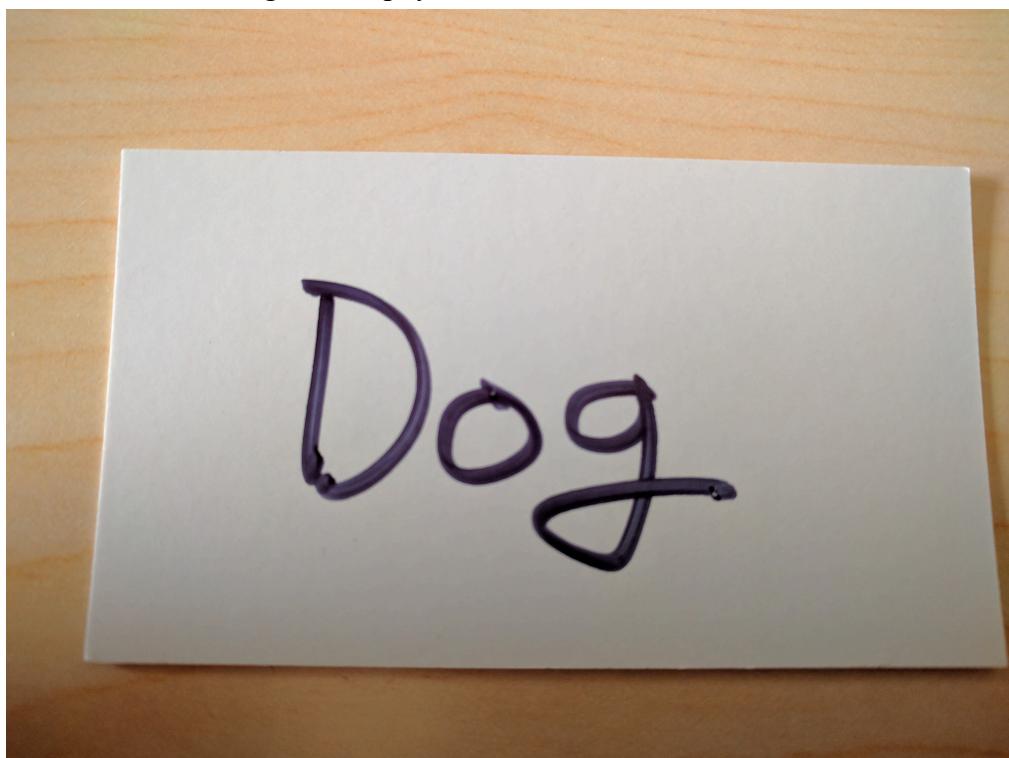


All content must pass through all three domains to get from ideas in a head to dots on a page. But the author can record that content in the media domain, the document domain, or the subject domain. In the next three chapters we will look at what it is like to write in each domain.

Chapter 3. Writing in the Media Domain

The media domain is the structured writing domain in which the structures relate to the media in which the content is displayed. Such content is often considered “unstructured”, but all content has structure, and we can actually find all the patterns and techniques of structured writing in the media domain. This makes it a good place to study the fundamentals of structured writing.

At its most basic, a hand guiding the pen over paper or chisel into stone is working in the media domain through direct physical interaction with the media.



The closest you can get to pen and paper in the computer world is to use a paint program to directly place dots on the screen. You can select the pen tool and use your mouse or a stylus to write your text. This will record the text as a matrix of dots.



There is very little structure here. We are recording a pattern of dots. Those patterns of dots are text characters only in the sense that the patterns are recognizable as characters to the human eye. The computer has no idea they are characters.

This is a pretty inefficient way to write. You can work faster if you use the paint program's text tool.



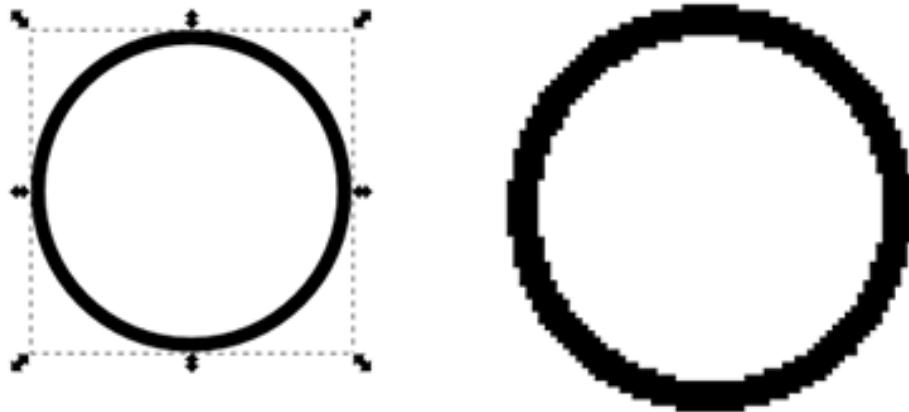
The text tool is our first step into structure. It partitions off the complexity of forming letter shapes from the task of writing letters and directs that complexity to an algorithm in the paint program, which does a better and more efficient job of it than the author does, as you can see from the much neater letter shapes in the sample above. However, those letters are still recorded as a set of dots, not as characters, so you can't go back and edit your text as text, only as dots. The act of forming letter shapes has been partitioned off, but the partitioning is not maintained in how the content is recorded.

This failure to maintain the partitioning of complexity in how content is recorded is a problem you may encounter many times in structured writing. For example, some wiki systems allow you to define content templates but do not record the structure of the template in the resulting page. The ability to maintain the partitioning of complexity is one of the key features of a mature and reliable content system, and one that is often neglected in the name of using simpler tools.

To partition the complexity of forming and editing letter shapes completely, you need to move away from dots and start working in a program that records characters as characters. You could go to a text editor, but a text editor does not keep any formatting information (unless you create markup – but that would be getting ahead of ourselves). That would not be partitioning the letter shape complexity, but dropping it. For most publishing purposes, plain text is inadequate. We need to maintain the ability to format the document.

One type of program that lets us record text as text and also lets us attach formatting to the text is a vector graphic program. A vector graphics (or “draw”) program creates graphics as a collection of objects (“vector” meaning the mathematical representation of a shape or line). For example, it allows you to create a circle as a shape, described mathematically in computer memory, rather than as a set of dots. Rather than recording an actual circle, the program records an abstraction of a circle: the essential properties needed to reproduce an actual circle on a media, such as its center, diameter, and line weight. In other words, it partitions all the complexity of circle drawing into the circle object. The computer then lets you manipulate that abstraction as an object, only rendering it as actual dots when the graphic is displayed on screen or paper.

Figure 3.1. Objects vs. dots



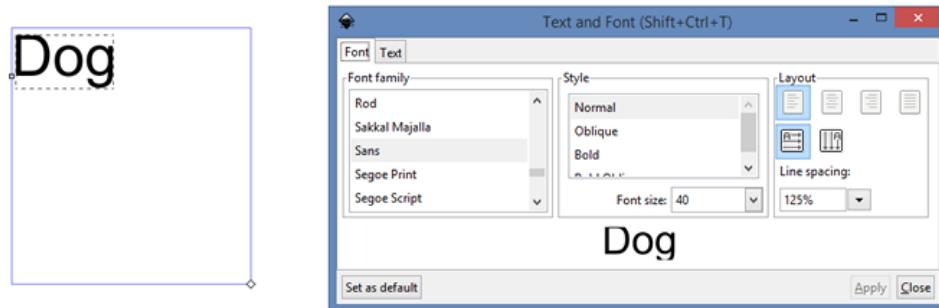
A circle as an object, displayed in a vector graphics program (Inkscape), left, vs. a circle as a set of dots in a raster graphics program (Microsoft Paint), right.

In a typical vector graphics program, a shape is rendered into dots on screen instantly as you draw or edit the shape. Nonetheless, the computer is storing data describing the shape, not a circular pattern of dots, as it would in a paint program. This is an instance of what in structured writing is called “separating content from formatting”. The mathematical abstraction of a circle is the content; the dots that represent it on screen are the formatting, or rather, the result of applying formatting to the object.

By partitioning aspects of the content into a form that algorithms can work with, we can make it easier to create and manage the content. All the principles of structured writing are present in this basic piece of computer graphics.

Partitioning font information

Just as a vector graphics program represents a circle as a circle object, it represents text as a text object. A text object is a rectangular area that contains characters. It has numerous media domain properties, such as margins, background and foreground colors, the text string, and the font face, size, and weight used to display that text, as in this example from InkScape:

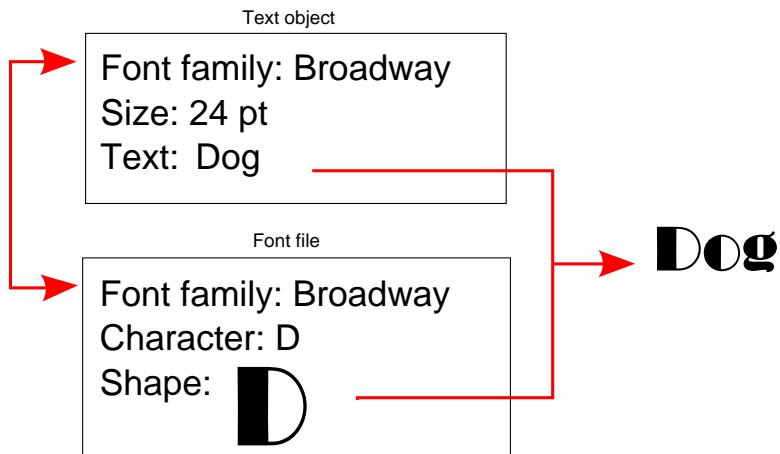
Figure 3.2. A vector graphics text object

A text object in a vector graphics program, with object properties shown.

A vector graphics program displays text in a chosen font. If you change the value of the text object's font attribute, it will immediately redraw the text in the new font. Not only has the drawing of letter shapes been partitioned, but the partitioning had been preserved in the way the content has been recorded in the source file, meaning we can redraw the letter shapes as much as we like without editing the text.

The shapes of the individual letters in the font are required information for rendering the text object in the media domain. However, they are not stored as part of the text object. The representation of the text in the paint program includes the shape of the letters. In the vector graphics program it does not. That information has been partitioned out.

The shape of the letters (technically, “glyphs”) that make up the font are stored separately in font files. Font files consist of a set of shape objects that describe each glyph, together with metadata such as the name of the font and the name of each glyph. To actually display the text block on screen, the graphics program (or rather the graphics system API to which it delegates this task) combines information from the font file with information from the text object by matching the metadata to find the right font and character, and then drawing the appropriate glyph on the current media.

Figure 3.3. Merging text and font information

The vector graphic text object factors out letter shapes to a separate font file.

The font system represents a partitioning of the complexity of rendering formatted text that has been built into every modern operating system. By partitioning this complexity and transferring it to the operating system, the OS designers made it easier for every developer who wants their application to work with formatted text. Rather than programming font handling themselves, they simply call operating system APIs to do it for them. It also makes it much easier for anyone who wants to design fonts, since it partitions off the problems of installing fonts and making them available to applications. These are a particularly powerful example of partitioning and transferring complexity because they mean that the various sets of people, all working in the content domain – font-designers, tool-developers, and writers – no longer have to know anything about how the others work and no longer need to communicate or coordinate in order to make their individual contributions to the overall content delivery process.

This is a pattern we will see over and over again in the partitioning of content complexity. In order to simplify the objects that we create to store our content, we take part of the information needed to do the final rendering, and partition it to a separate file. By partitioning out information that is constant for a given application (the shape of a capital D is the same for all capital D's for text in a given font), we simplify the format of the information we are preparing and keep the downstream presentation more consistent.

Designing a content structure, regardless of the domain you choose to work in, essentially consists of identifying the places in the content where we can partition out these invariant properties into separate structures.

Partitioning the complexity of pagination

Writing a document in a vector graphics program is certainly better than in a paint program, but you quickly run into a problem if you try to write a document that covers more than one page. A vector graphics program works purely in the media domain, and pretty much lets you put shapes and text boxes anywhere you like. This certainly allows you to create a new page whenever you need one, simply by creating a new drawing. But what happens when you want to edit the text and need to change how it flows from one page to the next? Pagination is a complex process, and we need a way to partition that complexity from the writing task and distribute it elsewhere, preferably to an algorithm.

Word processors and desktop publishing programs partition the pagination process from the writing process by introducing some document domain constraints. A document is made up of a series of pages that have margins and contain text flows. Text flows are made up of blocks (paragraphs, headings) inside of which text can flow, even from one page to the next. Common features like tables are supported as objects than can exist in text flows. New pages are created automatically as text expands. This leaves authors free to focus on writing and leave pagination to the program.

Pages, paragraphs, headings, and tables, are all document domain objects. Rather than working on a blank slate, as you do in a graphics program, you are now working within the constraints of these document domain objects. These constraints remove or constrain decisions about positioning of elements, which makes creating documents faster and more consistent. Structured writing is about making content that obeys constraints, and these basic document domain constraints are the next step in that journey.

Partitioning off the pagination problem is not without its drawbacks, however. There are certain page layout effects that are difficult or impossible to achieve in Word or FrameMaker because you have given up some of the liberty of a vector graphics program. We also gave up some liberty when we moved from raster to vector graphics, which is why photo editing, which requires adjusting individual pixels, is done in raster rather than vector format.

There is an important lesson here. We are starting to see that different partitioning of the content creation process is appropriate for different kinds of content. There is no one universal correct partitioning and distribution of content complexity that works for every type of content or every organization. This will continue to be the case as we look at more advanced forms of structured writing.

Chapter 4. Writing in the Document Domain

Word processors and desktop publishing applications tend to straddle the divide between the media domain and the document domain. While they are built on a basic set of document domain objects – pages, paragraphs, tables, etc. -- they use a WYSIWYG display, which unifies, rather than partitions, the writing and formatting of a document, to keep the author working and thinking mostly in terms of styles and formatting – the concerns of the media domain. This makes it difficult to apply meaningful document domain constraints to the author’s work, or to record which constraints the author has followed. This makes it impossible to redistribute formatting complexity away from the author. For that we need to move more fully into the document domain.

The simplest reason for moving to the document domain is actually to enforce media domain constraints that are hard to enforce in the media domain itself. In fact, the reason for moving to the next domain is often to enforce – or factor out – constraints in the previous domain. This is one of the consistent patterns we find in structured writing.

Consider a list. You may want to impose a constraint that the spacing above the first item of a list must be different from the spacing between other items of the list. This is a media domain constraint – it is about formatting, not the structure of the document – but it is hard to enforce in the media domain. Most media domain writing applications create lists by applying styles to ordinary paragraphs. The usual way to apply the extra space above the first item is to create two different styles, which we can call first-item-of-list style, and following-item-of-list style. The first-item-of-list style is defined with more spacing above.

This creates a constraint that the author is required to follow. They are required to apply first-item-of-list style to the first item of a list and following-item-of-list style to the following items. Nothing in the word processor enforces this constraint. The author simply has to remember it. This creates two problems:

1. It makes authoring a little bit harder (and all the little bits add up). New authors may not know to do this and may just use the list button. These errors then have to be caught and fixed.
2. The author can get it wrong without anyone noticing. They may forget to apply first-item-of-list style or they may add a new first item to the list and not realize that the second item in the list now has first-item-of-list style.

As we noted before, structured writing works by factoring out invariants. Most constraints are invariants – that is, they are rules that apply to all instances of the same content structure (such as all lists must have extra space before the first item). The easiest way to enforce a constraint is not to enforce it on the author, but to partition it out altogether.

To do this, we create a list structure – not a styled paragraph, but an structure that is specifically a list. The word processor version of a list is structured like this, as a flat list of styled paragraphs:

```
p{first-item-of-list}: Carrots  
p{following-item-of-list}: Celery  
p{following-item-of-list}: Onions
```

An explicit list structure looks like this:

```
list:  
  list-item: Carrots  
  list-item: Celery  
  list-item: Onions
```

A list structure partitions the idea of a list from the physical formatting of a list by creating a container that did not exist before – the `list` structure. That `list` container has no media-domain analog. It is an abstraction created to instantiate the partitioning of the idea of a list from the formatting of list items. It is a document domain structure.

Once we have a `list` structure, we can create rules – in a separate file – about how lists are formatted. We are familiar with this from HTML and CSS. Here's that same structure in HTML (actually, this is a slightly more specific structure, but we'll get to that):

```
<ol>  
  <li>Carrots</li>  
  <li>Celery</li>  
  <li>Onions</li>  
</ol>
```

Now that we have a distinct list object, we can factor out our invariant list formatting rule into a separate file. For HTML, this is usually done with a CSS stylesheet:

```
li:first-child  
{  
  padding-top: 5pt;  
}
```

Now the correct spacing-above-lists constraint has been partitioned off and assigned to an algorithm. To achieve this, though, a little bit of new structure has been added to the author's world: the list structure. That structure is essential to partitioning the list from the formatting rule. It is required to communicate the fact that the text is a list from

the author to the formatting algorithm. This better distributes the complexity of content creation by asking the author to express a simple idea in terms that are known to them – this is a list – instead of a complex set of formatting instructions.

While introducing the concept of a list structure has added a little bit to the total complexity of our content system, and has required our authors to learn a little bit of new complexity, we are more than compensated for this by the fact that a much larger piece of complexity has been partitioned off and assigned to an algorithm (which will be much more consistent about it than any author ever was), and the total amount of complexity vying for the author's limited store of attention has been reduced.

But wait! That's fine if all lists are formatted exactly the same way, but we know that is not true. At very least, some lists are bulleted and some are numbered. And then there are nested lists, which are formatted differently from their parents, and specialized lists, like lists of ingredients, definitions, or function parameters. If we are going to create list structures in the document domain rather than applying list styles in the media domain, how do we make sure each of these types of lists gets formatted appropriately?

Extensibility

At this point it is worth looking at a very important feature of all structured writing systems: extensibility. In media-domain word processing and desktop publishing programs, authors may need many different styles to format their documents, and these applications do not attempt to anticipate or provide all the styles every author might need. Some, like Word, come with a basic set of styles that may meet some basic needs, but all these programs let authors define new styles as well. The set of document domain structures in these programs is small and fixed, but the set of media domain styles is extensible – you can create as many as you need.

A word processor or desktop publishing application that supports the definition of styles is essentially creating an extensible media domain environment. Styles are media domain structures that partition out a set of style information that you can attach to a block of text to specify how it will be displayed. Every time you create a new style you are extending your set of media domain structures.

This need for extensibility is another common pattern in structured writing. If you are working in the media domain, you may need to extend your set of styles. If you are working in the document domain, you may need to extend your set of document domain structures.

How many types do we need? One obvious formatting difference between lists is that some are numbered and some are bulleted. How does a formatting algorithm tell whether to use bullets or numbers to format a given list? One way would be to add a style attribute to specify bullets or numbers, but then the author would be working in the media domain again, breaking the partitioning we had hoped to achieve. To keep the author in the

document domain, we need to create document domain structures that expresses different list types at the doc domain level.

The common way to handle bullets vs. numbers is to create two different list structures, the ordered list and the unordered list. Different markup languages call them by different names – `ol` and `ul` in HTML, `orderedlist` and `itemized-list` in DocBook, for example – but they are conceptually the same thing. (Thus the HTML example above is a little more specific than just being a list structure. It is an ordered list structure.)

The choice of the terms “unordered” and “ordered” is important, because it focuses on the document-domain properties of a list – whether its order matters – rather than on its media domain properties – bullets or numbers. Whether an ordered list should be formatted with numbers or letters or Roman numerals, belongs entirely to the media domain. It has been partitioned out of the document domain structures.

Context and Structure

Does the need for separate ordered and unordered list objects imply that we will need a separate document domain list structure for every possible way a list could be formatted in the media domain? No. In fact, that would really just be working in the media domain by proxy. When we work in the document domain we are specifically thinking in terms of document structures, not formatting, and so each document domain object we create needs to make sense in document domain terms, not media domain terms. Otherwise, the partitioning of the problem falls apart.

For example, consider nested lists. Items in a nested list are formatted differently from the list that contains them. At minimum they are indented more and usually they have different number and bullet styles as well. In the media domain, we would need a different style for each level. However, we don’t need a separate nested list structure in the document domain. Instead, we express the fact that a list is nested by actually nesting it inside its parent list. For instance, we can nest one ordered list inside another ordered list:

```
<ol>
  <li>
    <p>Dogs</p>
    <ol>
      <li>Spot</li>
      <li>Rover</li>
      <li>Fang</li>
      <li>Fluffy</li>
    </ol>
  </li>
  <li>
    <p>Cats</p>
```

```
<ol>
    <li>Mittens</li>
    <li>Tobermory</li>
</ol>
</li>
</ol>
```

In the document domain the inner and outer list are both identical ol/li structures. In the media domain, one will be formatted with Arabic numerals and the other with letters.

1. Dogs
 - a. Spot
 - b. Rover
 - c. Fang
 - d. Fluffy
2. Cats
 - a. Mittens
 - b. Tobermory

Both the inner and outer lists are ordered list items in the document domain, but in the media domain they are formatted differently based on context.

In this case, the algorithm that formats the page distinguishes the inner and outer lists by looking at their parentage. For instance in CSS:

```
ol>li>ol>li
{
    list-style-type: lower-alpha;
}
```

This ability to distinguish structures by context is vital to structured writing. It enables us to reduce the number of structures we need to fully describe our content, particularly in the document and subject domains. It also allows us to name structures more logically and intuitively, since we can name them for what they are, not how they are to be formatted or for where they reside in the hierarchy of the document as a whole.

It also points out another important difference between the way media domain and document domain writing is usually implemented. The media domain almost always uses a flat structure with paragraphs, tables, etc. following one after the other in sequence. For instance, a nested list in Word is constructed as a flat sequence of paragraphs with different styles. Inner and outer lists are expressed purely by the indent applied to the paragraphs. (Word tries to maintain auto-numbering across such listed nest structures, but does not always get it right.)

In the document domain, document structures are usually implemented hierarchically. List items are *inside* lists. Nested lists are *inside* list items. Sections are *inside* chapters. Subsections are sections *inside* other sections. Where the media domain typically only has before and after relationships (except in tables), the document domain typically adds inside/outside relationships to the mix. This use of nested, rather than flat, structures helps to create context, which helps to reduce the number of different structures you need. Just as we saw with the basic list structure, these nested structures help partition logical document structures so that we can distribute the complexity of their formatting to algorithms.

Some document domain languages are more hierarchical than others. HTML is relatively flat in structure. For example, it has six different heading styles H1 through H6. Docbook, a widely used document domain structured writing language, is much more hierarchical in structure and has only one element for the same purpose: `title`. But DocBook's `title` element can occur inside 84 different elements, and therefore can potentially be formatted in 84 different ways based on context. In fact, it can potentially be formatted in more ways than that, since some of the elements that contain it can also be nested in other elements, creating even more contexts.

There is a balance to be struck here, however. Nested structures are harder to create and can be harder to understand. Often they require the writer to find just the right place in a hierarchical structure to insert a new piece of content, which is more difficult than simply starting a new paragraph in a word processor. The partitioning they provide also introduces complexity into the writer's world, and we need to be vigilant to make sure that we are not introducing more complexity to the author than we are taking away.

Constraining document structure

There are other reasons for working in the document domain beside partitioning out formatting rules. One of the main ones is to constrain how documents are structured. For example, let's say that you want to make sure that all graphics inserted into your documents have a figure number, a title, and a caption. This is a document domain constraint rather than a media domain constraint. The requirement for a graphic to have a figure number, title, and caption is one of document structure and organization and does not say anything about how the title or caption should be formatted.

In the media domain, you can make styles available for figure-numbers, titles, and captions, but you can't enforce a rule that says all graphics must have these elements (which is, by its very nature, a document domain rule). In the document domain, you can express these constraints. You can literally make sure that the only way to include a graphic is to make it a figure and give it a title and a caption by making it illegal to place an `image` element anywhere else in the document structure. A structure that implemented this constraint might look like this:

```
figure:  
    title: Cute kitty  
    caption: This is a cute kitten.  
    image: images/cute.jpg
```

If the only way to include an image is to use the image element, and the only place where the image element is allowed is inside the figure element, and if the title and the caption elements are required and must have content, then there is no way for a writer to add a graphic without a figure, title, and caption. A document that lacked these elements would be rejected by the conformance algorithm (see Chapter 16, *Conformance*).

This is an example of a constraint being enforced rather than factored out. But it is still a form of partitioning because we have created a partition for figures which is the only place images are allowed. This partitions off the design question about image handling. With the list formatting we could partition out the formatting constraint completely by separating the formatting information from the list structure. Here we enforce the image constraint by providing a specific structure that must be used to insert images.

There are many ways in which you might want to constrain the structure of a document. In a typical media domain application, there is no restrictions on the order in which paragraph styles can be applied. If you want to put a level two heading between two steps in a procedure, nothing other than common sense will stand in the way of your doing so. In a document domain language, however, that kind of thing will usually not be allowed.

Instead, the document domain language will have a set of constraints on how procedures are structured. Procedure structures will have step structures nested within them. Step structures will only be allowed to appear inside procedure structures. Only certain text elements – such as paragraphs, lists, or code blocks – will be allowed to occur inside a step. There will be no way to place a second level heading inside a procedure.

Constraints like these are important to document domain languages. If you want to control how procedure are written, or how graphics are labeled, you need to create specific document domain structures for these things, and constrain them to avoid them being misused. Without such constraints, it is easy for a language to slip back into the media domain, something that has happened to HTML.

Today, structured writing advocates often dismiss HTML as an unstructured language. They will point to other languages, such as DocBook or DITA, as being structured by contrast, despite the fact that all three languages are document domain languages at heart with many similar structures between them.

And while it was not always so, HTML has largely become a set of basic document domain structures that authors can hang styles on (using CSS). In other words, it has come to be used much like traditional media-domain word processing and desktop publishing applications. When people write in HTML, they largely do so in WYSIWYG environments, using style-oriented tools that mimic traditional word processing very

closely. The result is often an HTML document that formats more or less correctly, but that is coded very inconsistently from the point of view of the document domain, and which is thus very hard to work with – either to edit by hand or manipulate with algorithms.

Extending the document domain

Another factor in HTML’s slide into the media domain is that it provides only a fairly basic set of document domain structures. As we have seen, enforcing or factoring out media domain constraints requires specific document domain structures. But the possible list of such structures is quite large. There are a few basic features that are common to all documents, such as paragraphs, lists, and titles. But these structures alone are not enough to hang meaningful and useful document domain constraints on, or to fully partition the content of a document from its formatting, which is why, as we noted above, extensibility is an important part of all structured writing domains.

For example, think about a bibliography. A bibliography is a document structure for listing works cited in or recommended by a document. It generally consists of a heading “Bibliography” followed by a set of paragraphs listing the cited works. In the media domain, it is not a particularly complicated structure to create. It is just a sequence of paragraphs with some bold and italic formatting for author names, book titles, etc.

In your media domain stylesheet, you may have some character styles for things like author-name or book-title. You may even have a specific paragraph style for bibliography entries, but it is unlikely to be more complicated than that.

But these few styles don’t really cover all the rules for creating bibliographies that your organization or publisher is likely to insist on. There are rules for the presentation of a bibliography entry which go into detail about how each work and its authors are listed and how the listings are presented. These are constraints on the writing of the bibliography that the author has to follow, but which are not modeled by the media domain styles they are working with. The authors have to learn and follow these constraints for themselves, and when they have finished writing, these constraints will not be recorded in the content in a machine-readable way. If the constraints are not recorded, you won’t be able to write an algorithm to pull information out of bibliography entries because all the algorithm can see is a bunch of paragraphs with some bold and italic formatting.

If you want to control how bibliographic information is presented, and enable algorithms to find and extract data from bibliographic entries, you need a document domain structure for a bibliography. DocBook has one:

```
<biblioentry id="bib.xsltrec">
  <abbrev id="bib.xsltrec.abbrev">REC-XSLT</abbrev>
  <editor>
```

```
<firstname>James</firstname>
  <surname>Clark</surname>
</editor>
<title>
  <ulink url="http://www.w3.org/TR/xslt">XSL Transformations
    (XSLT) Version 1.0</ulink>
</title>
<publishername>W3C Recommendation</publishername>
<pubdate>16 November 1999</pubdate>
</biblioentry>
```

The example is in XML, which can be hard to read, so here is the same structure the markup notation that I have used in earlier examples:

```
biblioentry: (#bib.xslttrec)
  abbrev: (#bib.xsltrec.abbrev) REC-XSLT
  editor:
    firstname: James
    surname: Clark
  title: XSL Transformations (XSLT) Version 1.0
  publishername: W3C Recommendation
  pubdate: 16 November 1999
```

This structure does not just constrain how bibliography entries are presented and formatted, it actually factors out many of those constraints by breaking down the components of a bibliography entry into separate labeled fields. Given a `biblioentry` structure like this, you could create an algorithm to present and format a bibliography entry almost any way you wanted to. This means that this structure not only partitions the formatting of the bibliography from the presentation of the bibliography, it also partitions the presentation from the underlying bibliographical data from the presentation. This means you could write an algorithm to extract bibliography information from a document by looking for `biblioentry` structures and extracting the desired information from them. For instance, if you want to build a list of authors cited in the document, you could do so by searching the `biblioentry` records and extracting the name of the authors from the structures that record them in the bibliography structure.

Chapter 5. Writing in the Subject Domain

Structured writing in the subject domain means creating structures that are specific to the subject matter being discussed. In essence, the subject domain partitions the subject matter of the text from its presentation. The utility of that partitioning may not be immediately obvious, but as we will see, it provides powerful solutions across a range of problems in the content system.

A recipe is a useful example for illustrating the use of the subject domain. Here is a recipe written in reStructuredText, a lightweight general purpose document domain markup language:

```
Hard Boiled Eggs
=====
A hard boiled egg is simple and nutritious.
Prep time, 15 minutes. Serves 6.

Ingredients
-----
=====      =====
Item      Quantity
=====      =====
eggs     12
water    2qt
=====      =====

Preparation
-----
1. Place eggs in pan and cover with water.
2. Bring water to a boil.
3. Remove from heat and cover for 12 minutes.
4. Place eggs in cold water to stop cooking.
5. Peel and serve.
```

In reStructuredText, a line underlined with equals signs is a major heading and one underlined with dashes is a minor heading. A table is created by using equals signs to mark the beginning and end of the table and the boundary between the table head and the table body. Rows are separated by new lines and columns by spaces in the rows of equals signs. Numbered lists are created by putting numbers in front of lines of text. The equivalent HTML document would look like this:

```
<html>
  <h1>Hard Boiled Eggs</h1>

  <p>A hard boiled egg is simple and nutritious.  
Prep time, 15 minutes. Serves 6.</p>

  <h2>Ingredients</h2>
  <table>
    <thead>
      <tr>
        <th>Item</th>
        <th>Quantity</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>eggs</td>
        <td>12</td>
      </tr>
      <tr>
        <td>water</td>
        <td>2qt</td>
      </tr>
    </tbody>
  </table>
  <h2>Preparation</h2>
  <ol>
    <li>Place eggs in pan and cover with water.</li>
    <li>Bring water to a boil.</li>
    <li>Remove from heat and cover for 12 minutes.</li>
    <li>Place eggs in cold water to stop cooking.</li>
    <li>Peel and serve.</li>
  </ol>
</html>
```

This document follows the normal rhetorical pattern of a recipe. That is, it has the things a recipe normally has, in the order they normally occur in a recipe: introduction, list of ingredients, preparation steps. However, it does not record the fact that it follows this rhetorical pattern. There is nothing in the markup to say that this is not a novel, a car manual, or knitting pattern. Nor would the markup constrain an author to follow the normal rhetorical pattern of a recipe while writing.

Moving this document to the subject domain would allow us to impose these rhetorical constraints, and to record that we have done so. Neither reStructuredText nor HTML

give us a way to do that, so we will need a different markup language. Here's what that might look like:

```
recipe: Hard Boiled Egg
    introduction:
        A hard boiled egg is simple and nutritious.
        Prep time, 15 minutes. Serves 6.
    ingredients:
        * 12 eggs
        * 2qt water
    preparation:
        1. Place eggs in pan and cover with water.
        2. Bring water to a boil.
        3. Remove from heat and cover for 12 minutes.
        4. Place eggs in cold water to stop cooking.
        5. Peel and serve.
```

This structure breaks the document up into a collection of named structures. Those structures are “introduction”, “ingredients”, and “preparation” and they are contained in an overall structure called “recipe”. This is the basic rhetorical structure of a recipe. This markup make it explicit that this is a recipe (not a novel, a car manual, or a knitting pattern). The author is explicitly guided to follow this pattern. They are also constrained to present the ingredients as an unordered list and the preparation as a numbered list. (Chapter 16, *Conformance* will look at how such constraints are expressed and enforced.)

One of the common patterns of structured writing is the partitioning out of invariants. One of the invariants of the recipe pattern is that it has sections titled “Ingredients” and “Preparation” (or words to that effect). Notice that these titles been partitioned out here. The titles are part of the rhetorical structure of a recipe, and since the markup now models the rhetorical structures, we can factor out the titles themselves by creating structures called `ingredients` and `preparation`. Since we need those sections to record the rhetorical pattern we have followed, the titles are redundant in our source and can be partitioned out and can be added back into the content by an algorithm at publishing time.

Partitioning requires that we pass enough information to the partition to do its job. Here the presence of the `ingredients` and `preparation` sections in the recipe structure provide the information that the algorithm needs to insert the appropriate titles.

By factoring out titles, we factor out the constraint on what those titles must be. An author no longer has to remember the standard titles to use, and a whimsical author can no longer decide to title these sections “Stuff you need” and “Stuff you do”, or any other variant of the standard that the organization has chosen. If we want to change these titles across all the recipes, on the other hand, we only have to change the algorithm.

If your organization publishes a lot of recipes, you probably have a lot more constraints on the rhetorical structure of your recipes. For instance, you might have a constraint that every recipe must state its preparation time and the number of people it serves. In our subject domain markup, we can enforce and record that constraint by moving the information from the introduction section to separate fields:

```
recipe: Hard Boiled Egg
introduction:
    A hard boiled egg is simple and nutritious.
ingredients:
    * 12 eggs
    * 2qt water
preparation:
    1. Place eggs in pan and cover with water.
    2. Bring water to a boil.
    3. Remove from heat and cover for 12 minutes.
    4. Place eggs in cold water to stop cooking.
    5. Peel and serve.
prep-time: 15 minutes
serves: 1
```

This approach changes the partitioning of the prep-time and serves information. Previously, writers had to remember the requirement to mention these facts in the text of the introduction. Now we have partitioned them off into simple fields that the writer is required to fill out. The writer no longer has to remember the presentation rules that dictate that this information is required and where it should appear. They are prompted for it directly and errors will be raised if they forget, and the decision about where to include it has been transferred to an algorithm.

Does this mean that the preparation time will now be displayed as separate fields in the output, rather than in-line? Not necessarily. It might be a good idea to call it out in separate fields so that readers can find the information more easily, but if you really wanted that information at the end of the introduction in every recipe, it would be a simple matter for the presentation algorithm (see Chapter 27, *Publishing*) to construct the sentences “Prep time, 15 minutes. Serves 6.” from the prep-time and serves field values.

So, something interesting has happened here. In order to enforce a rhetorical constraint – that prep time and number of servings must be specified – we have moved away from markup that specifies presentation to markup that merely records data. In other words, prep-time and serves are data-oriented subject domain structures that do not specify presentation at all. We are now partitioning the presentation of the recipe from the information the recipe must contain. This allows us to shift responsibility for conformance to our requirements from the author to a conformance algorithm. This

makes our design more testable and more repeatable – a welcome redistribution of complexity.

This is a recurring pattern in structured writing, and one of the most important things to understand about how structured writing works. It is almost always better to factor out a constraint than to enforce it. This can be a difficult idea to adjust to. If we have a particular form of presentation we want to achieve, our first instinct naturally is to specify it in detail. But this is not always easy to do, especially if you want to specify that a paragraph should always contain certain pieces of information. Nor is it flexible if you want to vary the presentation for any reason (we will look at some reasons why you might in Chapter 11, *Single Sourcing*). It is important to condition yourself so that when you look at these kinds of problem the first question you ask yourself is, is there a way I can factor out this constraint. Only resort to trying to impose a constraint if it is not possible to factor it out.

By the way, using data-oriented subject-domain markup like this also offers some interesting publishing possibilities. For instance, with this markup in place, you could easily query your set of recipes to create a cookbook of recipes you can make in 30 minutes or less because you can query your set of recipes and pull out just those in which the prep-time field has a value of 30 minutes or less.

Are there other elements of presentation that we can factor out of the recipe structure? As we noted, the reStructuredText version above uses a table presentation for the ingredients. Our recipe structure currently specifies a simple list. The block that contains it is labeled “ingredients”, but the contents of the block is just an ordinary unordered list.

There is a constraint here about how ingredients are expressed, one which authors need to follow, but one that our markup does not yet impose or record. The specification of ingredients in a recipe generally requires three pieces of information, the name of the ingredient, the quantity, and the unit of measure used to express this quantity. These can be presented as a list or a table. To factor out the presentation choice, we can create an ingredient structure that calls out each piece of information separately:

```
ingredients:  
    ingredient:  
        name: eggs  
        quantity: 12  
        unit: each  
    ingredient:  
        name: water  
        quantity: 2  
        unit: qt
```

There are some shortcuts we can take to make this markup less verbose. (This is a markup syntax named SAM that I will talk about later):

```
ingredients:: ingredient, quantity, unit
eggs, 12, each
water, 2, qt
```

This markup turns the ingredients into a set of records with named fields for ingredient, quantity, and unit. This enforces and records that the constraint on ingredients is followed. And because the way they are recorded is independent of any one form of presentation, we are now free to use an algorithm to present them either as a table or as a list.

By adding and recording these constraints, we get similar benefits as before. We can better enforce any constraints we have about how ingredient lists are structured and formatted, and we gain access to the specific data involved, meaning, for example, that we could write an algorithm to convert our units from imperial to metric for publication in markets where metric units are preferred.

The role of the subject domain

We don't always use the subject domain for an entire document. Sometimes the use of the subject domain is a minor part of a primarily document domain language. For example, DocBook includes subject domain tags like `GUILabel` for references to part of a computer display.

Conversely, most subject domain languages also contain some document domain text structures like paragraphs and lists. It is usually not possible to break your treatment of a subject down completely into named fields, after all, and the subject domain cannot represent the entire argument of a piece of content. For the rest we use text, and we organize that text in document domain structures.

When we do create a subject domain document, we may go to different lengths in factoring out content into subject-domain fields. Some subject domain document types are mainly focused on dictating the rhetorical structure to be used for handling a particular subject. It is defining the rhetorical construction of a recipe document. This is what we did in the first subject domain refactoring of our recipe example. This approach does not typically give us much access to the information in the document, it merely governs its shape.

When we start to pull out the prep and serves numbers and turn the ingredients list into a fielded data structure, however, we are factoring out the specifics of recipe presentation and focusing on the elements of recipe as data. This gives us presentation independence, but it also gives us access to the data in the document. Getting access to the data in a document can be key to many of the structured writing algorithms, as we will see in subsequent chapters.

Using subjects to establish context

In Chapter 4, *Writing in the Document Domain*, we noted that we can use context to identify the role that certain structures play in a document, which allows us to get away with fewer structures. For instance, we can use a single `title` tag for all titles because we can tell what kind of title each one is from the context in which it occurs. The same is true with subject domain structures. They can provide context that allows us to treat basic text structures differently.

Consider our markup language for recipes:

```
recipe: Hard Boiled Egg
    introduction:
        A hard boiled egg is simple and nutritious.
    ingredients:: ingredient, quantity, unit
        eggs, 12, each
        water, 2, qt
    preparation:
        1. Place eggs in pan and cover with water.
        2. Bring water to a boil.
        3. Remove from heat and cover for 12 minutes.
        4. Place eggs in cold water to stop cooking.
        5. Peel and serve.
    prep-time: 15 minutes
    serves: 6
```

With the ingredients, we saw that we needed to add additional structure to factor out whether the ingredients would be presented as a list or a table. For the preparation, the steps are currently marked up as a numbered list. Suppose we want to present the steps as steps, rather than just as a generic numbered list (for instance, by labeling them as **Step 1.**, etc, rather than just **1.**). Do we need to create an additional `step` structure to do this? Not necessarily. In this case we can tell the difference between an ordinary ordered list and a set of preparation steps based on context. We can write a rule in the presentation algorithm that creates special formatting just for ordered lists that are the children of `preparation` elements that are children of `recipe` elements (just as we could create rules to format a nested list differently from its parent list based on its context in Chapter 4, *Writing in the Document Domain*). This is another example of how partitioning creates context which simplifies various publishing functions.

Chapter 6. The Management Domain: an Intrusion

So far I have talked about three domains that content passes through and in which it can be recorded: the media domain, the document domain, and the subject domain. But there is a forth domain that intrudes into this picture: the management domain.

Why do I call the management domain an intrusion? Because while the subject, document, and media domains are all about recording the content itself, the management domain is not about the content, but about the process of managing it.

Let's suppose we run a publishing company that publishes a number of magazines. We want to create a common store of recipes for use in all the magazines. But different magazines have different requirements. *Wine Weenie* magazine needs to have a wine match with every recipe. *The Teetotaler's Trumpet*, naturally, wants a non-alcoholic suggestion.

Here is how that might be handled in the document domain:

```
<section publication="Wine Weenie">
    <title>Wine match</title>
    <p>Pinot Noir</p>
</section>
<section publication="The Teetotaler's Trumpet">
    <title>Suggested beverage</title>
    <p>Lemonade</p>
</section>
```

This is an example of what we call conditional text. The publication attribute on the `section` element says, display this text only in this publication. This makes it management metadata, which means these structures are in the management domain.) It does not specify the formatting, organization, or subject matter of the document. It specifies which publication the content should appear in, which is a content management decision.

Including boilerplate content

Management domain structures can be used to do a number of things. For example, let's say you have a standard warning statement that you are required to include in a document wherever you have a dangerous procedure. Structured writing partitions complexity by factoring out invariants, and the invariant here is that this warning statement must appear whenever you describe a dangerous procedure.

Just as we extracted formatting information into a separate file when we moved content from the media domain to the document domain, we now extract the invariant warning from the document and place it in a separate file. Any place we want this warning to occur, we insert an instruction to include the contents of the file at that location.

```
procedure: Blow stuff up
    <<<(files/shared/admonitions/danger.sam)
    step: Plant dynamite.
    step: Insert detonator.
    step: Run away.
    step: Press the big red button.
```

In the SAM markup above the <<< is an include command. It includes the content of the file located at files/shared/admonitions/danger.sam in the source file as it is read.

The equivalent in XML would look something like this:

```
<procedure>
    <title>Blow stuff up</title>
    <xi:include href="files/shared/admonitions/danger.xml" />
    <step>Plant dynamite.</step>
    <step>Insert detonator.</step>
    <step>Run away.</step>
    <step>Press the big red button.</step>
</procedure>
```

Why is this operation part of the management domain, rather than the document domain? Because it deals with a system operation: locating a file in the system and loading its contents. If we were purely in the document domain, the author would be the one performing this operation: finding the file with the warning in it, opening it, and copying the contents into the document. The include instruction is just that: an instruction. It is not a declaration about the subject matter or structure of a document, such as we find in subject domain or document domain markup. It is an instruction to a machine to perform an operation. The management domain consists of instructions or the declaration of data required to perform management functions.

Different structured writing systems have different instruction sets for handling the situation described above. In DITA, for instance, this use case is handled using something called a conref or a conkeyref. In Docbook it can be handled using a generic XML facility called XInclude, as in the XML example above.

An alternative approach in the subject domain

The downside of the management domain include instruction is that it introduces system complexity into the author's world. Adding complexity to the author's task is always a problem because the author's attention is a limited resource and everything that takes away their attention from writing has a direct impact on content quality. But the use of a conditional expression or an include instruction is not just a problem for the author, it also makes change management and content management more complicated by distributing management domain structures like conditions, and file paths throughout the content. Changing management decisions means changing management structures all over the content set.

Is there a way to partition the management complexity from the authoring and keep management metadata out of the content? In many cases this can be accomplished used the subject domain.

This is how we might approach the two magazines problem in the subject domain:

```
<wine-match>Pinot Noir</wine-match>
<beverage-match>Lemonade</beverage-match>
```

This markup says nothing about which documents should contain either of these pieces of information (a clear example of partitioning). Nor does it contain the subheadings what would introduce either of them in the appropriate publication. All these decisions are now left to algorithms (and the people who write them). This allows us to do far more things with this content without having to rewrite the source files in any way (a partitioning that significantly reduces the complexity of change management, an algorithms we will look at in Chapter 22, *Change management*).

As we saw in Chapter 5, *Writing in the Subject Domain*, factoring out invariant text is a feature of the subject domain. Since we are dealing with invariant text here, the subject domain may provide the solution. To understand the subject domain approach to this problem, remember what the invariant rule is here: A dangerous procedure must have a standard warning.

The management domain approach to this is to allow authors to insert the standard warning so that it is only stored once instead of being repeated in every procedure (something that is often called content reuse) an algorithms we will look at in Chapter 12, *Reuse*. Notice that the management domain markup does not encapsulate our invariant rule that dangerous procedures must have a standard warning. It just provides a generic mechanism for inserting content as a reference to a file. The requirement has not been partitioned, we have merely provided a mechanism for implementing each case of the

requirement. It leaves it entirely up to the author to remember and enforce the rule about dangerous procedures – complexity is not being distributed away from the author.

The subject domain approach, on the other hand, is all about the invariant rule itself. Specifically, it expresses the aspect of the subject domain that triggers the rule: whether a procedure is dangerous or not:

```
procedure: Blow stuff up
  is-it-dangerous: yes
  step: Plant dynamite.
  step: Insert detonator.
  step: Run away.
  step: Press the big red button.
```

This markup simply records that this procedure is dangerous, a fact about the subject matter. This partitions the requirement, by retaining the information on which our invariant rule is based, but factors out the action to be taken. Rather than asking authors to remember to include the file (and how to included it, and how to find it) we delegate that responsibility to the presentation algorithm. It is now the algorithm, not the writer, that needs to remember to include the material in `files/shared/admonitions/danger` whenever the `is-it-dangerous` field of a procedure structure is set to “yes”. This is the sort of task that algorithms are much better at than humans.

Of course, the human writer does still have a job to do here. They have to remember set `is-it-dangerous` to “yes”. But we can make remembering to do this much easier if we make `is-it-dangerous` a required field in the procedure structure. In other words, we set up our structured writing language in such a way that an error will occur if `is-it-dangerous` is not specified for a procedure. This transfers the complexity of remembering the requirement and fulfilling it from the writer and editor to the conformance algorithm and the publication algorithm.

This approach makes the writer’s job much easier because they not only get reminded of the need to address the question of danger with every procedure, they are also asked it in a way that does not require them to know anything about how the content management system works, what warning text is required, or where it is located. They are recording a fact about the subject, not giving an instruction or interpreting a style guide. This is a much more efficient partitioning of the requirement that does a better job of distributing complexity of this important rule to the person or process best able to handle it, thus minimizing the possibility that this complexity could get dumped on the reader, for whom there might be potentially painful consequences.

One the other hand, this approach only factors out the reuse of one particular piece of content – the warning for dangerous procedures. If you had multiple such invariant rules about different kinds of subject matter you would need separate subject domain structures for each of them, whereas a single management domain include instructions would let authors handle them all.

On the other other hand, if you have many such invariant rules, and you expect all of them to be enforced by authors from memory, you are dumping an awful lot of complexity on your authors and you are going to limit your pool of authors to a few highly trained individuals. Even then they are still likely to miss some instances, dumping this dangerous complexity on the reader again.

Hybrid approaches

It is not always an either/or decision to use pure management domain or pure subject domain approaches. Management domain structures tend to be used in generic document domain languages, since such languages are not designed to be specific to any particular subject matter. Nonetheless, such languages often have roots in particular fields and sometimes include subject-domain structures from those fields. Both DocBook and DITA, for instance, originated in the field of software documentation and both include structures related to the subject of software, such as code blocks and structures for describing user interface elements.

In some cases, such languages can mix subject domain structures into their management structures. One example is the `product` attribute, which is part of DITA's conditional text processing system.

In DITA, you can add the `product` attribute to a wide variety of elements. You can then set a value for products in the build systems and any element with the `product` attribute will only be included in the final output if it matches one of the product values specified in the build.

```
<p>The car seats
<ph product="CX-5">5</ph><ph product="CX-9">7</ph>
</p>
```

DITA can afford to use this bit of subject domain markup for products because product variations are an extremely common reason for using conditional text processing in technical communication, the area for which DITA was created. (Through a process called “specialization”, DITA can add other subject domain attributes for conditional processing in other subject areas.)

The reason I call this a hybrid approach is that the DITA `product` attribute does not exist merely to declare that a piece of text applies to a particular product. It is specifically a conditional processing attribute. That is, it is an instruction, even though it is phrased as a subject domain declaration.

To appreciate the difference, consider that there is another approach to documenting multiple versions of a product. Rather than generating a separate document for each product variant, you could create a single document that covered all product variants

and highlighted the differences between them. A pure subject domain approach would support either approach by simply recording the data for each variant:

```
seats:  
  CX-5: 5  
  CX-9: 7
```

This information could be presented as data similar to its source format or it could be used to algorithmically construct a sentence like this:

The CX-5 seats 5 and the CX-9 seats 7.

That is not something that the product attribute supports:

```
<p>The car seats  
<ph product="CX-5">5</ph><ph product="CX-9">7</ph>  
</p>
```

This markup is only designed to produce a CX-5 or CX-9 specific document. It is not designed to support the production of a document that covers both cars at once because it does not specify that the values 5 and 7 are numbers of seats. That information is in the text, but not in a form that a publishing algorithm could reliably locate and act on.

Also, creating a single document covering both cars is not the expectation that goes with creating the markup. It is not what the author is told the markup means. The markup is not a simple declaration of facts about each car, it is markup of a document intended to produce a document about one car or the other, not both. It is conditional text markup, and therefore an instruction.

Really, is it a contraction of the more explicitly imperative form (not actually used in DITA):

```
<ph condition="product=CX-5">5</ph>
```

While the introduction of subject domain names into management domain structures is an appropriate bit of semantic sugar for authors, this hybrid approach really remains firmly in the management domain.

We will see many more examples of the management domain, and the cases in which there is and is not a subject domain alternative, in the following chapters.

Part II. Algorithms

Table of Contents

7. Quality in Structured Writing	48
Robots that read	48
Dumbing it down for the robots	48
Making humans better writers	49
We write better for robots than we do for humans	49
Structure, art, and science	51
Contra-structured content	52
Until the robots take over	52
Other sources of quality improvement	53
8. Writing	54
Functional lucidity	57
Simplicity and Clarity	57
9. Separating Content from Formatting	60
Separate out style instructions	60
Separate out formatting characters	61
Name your abstractions correctly	62
Make sure you have the right set of abstractions	62
Create containers to provide context	64
Move the abstractions to the containers	65
Separate out abstract formatting	66
10. Processing Structured Text	69
Two into one: reversing the factoring out of invariants	69
Adding back style information	69
Rules based on structures	71
The order of the rules does not matter	71
Applying rules in the document domain	72
Processing based on context	73
Processing container structures	75
Restoring factored-out text	75
Processing in multiple steps	77
Query-based processing	79
11. Single Sourcing	81
Basic single sourcing	81
Differential single sourcing	85
Differential organization and presentation	88
Conditional differential design	89
Primary and secondary media	90
Responsive Design	91
12. Reuse	93
Fitting pieces of content together	94
Common into variable	94
Variable into common	96

Variable into variable	97
Using IDs	98
Using keys	99
Common with conditions	101
Factor out the common	104
Factor out the variable	105
Assemble from pieces	106
Combining multiple techniques	108
Content reuse is not a panacea	109
Quality traps	109
Cost traps	110
Alternatives to reuse	112
13. Composition	114
Fundamental composability	115
Structural composability	115
Stylistic composability	116
Narrative composability	116
14. Normalization	118
The limits of content normalization	121
Examples of content normalization	122
Normalization and discovery	123
Proximity detection	124
Normalization via aggregation	125
15. Linking	126
Deflection in the media domain	127
Deflection in the document domain	128
Deflection in the management domain	129
The problem with IDs and Keys	130
Relationship tables	130
The problem with relationship tables	131
Conditional linking	132
Deflection in the subject domain	133
Finding resources to link to	136
Deferred Deflection	138
Different domain, different algorithm	140
16. Conformance	142
Completeness	145
Consistency	145
Accuracy	146
Semantic constraints	146
Entry validation constraints	149
Referential integrity constraints	150
Conformance to external sources	150
Conformance and change	152
Design for conformance	152

17. Auditing	155
Correctness of the definition of the content set	156
Bottom-up content planning	157
Ensuring the content set remains uncontaminated	157
Ensuring that the content set is well integrated	158
Making content auditable	159
Performance auditing	160
18. Relevance	162
Being relevant	162
Showing relevance	163
Showing relevance to algorithms	164
19. Extract	167
Tapping external sources of content	167
Information created for other purposes	168
The diversity of sources	171
20. Merge	173
21. Information Architecture	176
Top-down vs. bottom-up information architecture	178
Categorization	178
Linking	179
Tables of Contents	180
Lists	181
Personalized content	182
22. Change management	184
Content management systems	189
23. Content Management	191
Metadata is the foundation of management	192
The location of metadata	193
Managing the process	195
Conflicting constraints	196
Creating manageable content	198
24. Collaboration	200
Bridging silos	203
25. Timeliness	204
26. Repeatability	211
27. Publishing	214
The Rendering Algorithm	214
The Encoding Algorithm	215
The Formatting Algorithm	217
The Presentation Algorithm	218
Differential presentation algorithms	219
Presentation sub-algorithms	219
The linking algorithm	219
The information architecture algorithm	219
The Synthesis Algorithm	220

Differential synthesis	220
Synthesis sub-algorithms	221
The reuse algorithm	221
The extraction algorithm	221
The merge algorithm	221
Deferred synthesis	221
Combining algorithms	221
Architecture of a publishing tool chain	222
28. Active content	225
29. Translation	230
Extracting content for translation	230
Avoiding trivial differences	230
Isolating content that has changed	231
Continuous translation	231

Chapter 7. Quality in Structured Writing

When I talk to programmers about what I do, they sometimes ask me why structured content is important any more. Machines are getting so good at reading human language, they argue, that semantic markup to assist the machine is increasingly becoming pointless.

Robots that read

Indeed, machines are getting better and better at understanding human language. An approach called Deep Learning is increasingly becoming a key technology for companies like Facebook, Google, and Baidu for both language comprehension¹ and speech recognition².

The semantic web initiative has long sought to create a Web that is not just people talking to people but also machines talking to machines. This has traditionally involved an essentially separate communication channel – semantic markup embedded in texts but not presented to the human reader. It has also involved the creation of specialized semantic data stores with query languages to match, to teach computers to understand relationships that humans would express in ordinary language.

But this two-channel approach – one text for the human, another for the machine – only makes sense if we assume that the machine cannot read human language. If machine and human can both read the same text then we shouldn't need two channels. The human Web becomes the semantic Web.

After all, the human text always was semantic. Semantics is simply the study of meaning. All meaningful texts have semantics. It is just that it has been difficult to build algorithms that could read and understand like humans do. Semantic technologies are about dumbing the semantics down for the machine because the machine is not bright enough to read the regular semantics.

Dumbing it down for the robots

This dumbing down necessarily involves omitting a great deal of the semantics of the text. Fully expressing all the meaning and implications of even the simplest text in RDF triples, for instance, would be daunting. This has always created a problem for semantic technologies: which semantics do you select to dumb down to the machine's level, and for what purpose? This is why there is no universal approach to structured writing that works for all purposes and all subject matter. You can only represent a fraction of the human

¹<http://www.technologyreview.com/featuredstory/540001/teaching-machines-to-understand-us/>

²<http://www.technologyreview.com/news/544651/baidus-deep-learning-system-rivals-people-at-speech-recognition/>

semantics to the machine, and which you choose depends on what specific functions you want the machine to perform.

But if the machine could read the text as well as you can, then these limitation vanish. Deep learning is moving us in that direction.

Why then should we bother with structured writing? Quite simply because while machines may be rapidly learning to read human text, that text is still written by humans, and most humans are not good writers.

Making humans better writers

By that I don't just mean that they use poor grammar or spelling or that they create run-on sentences or use the passive voice too much, though all those things may be true, and annoying. I mean something more fundamental than that: they don't say the right things in the right way for the right audience. They leave out stuff that needs to be said, or they weigh it down with stuff that does not need to be said, or they say it in a way that is hard to understand.

We all suffer from a malady called the Curse of Knowledge³ which makes it difficult for us to understand what it is like not to understand something we know. We take shortcuts, we make assumptions, we say things in obscure ways, and we just plain leave stuff out.

This is not a result of mere carelessness. The efficiency of human communication rests on our ability to assume that the person we are communicating with shares a huge collection of experiences, ideas, and vocabulary in common with us.⁴ Laboriously stating the obvious is as much a writing fault as omitting the necessary. Yet what is obvious to one reader may be obscure to another. The curse of knowledge is that as soon as something becomes obvious to us, we can no longer imagine it being obscure to someone else.

Thus much of human to human communication fails. The recipient of the communication simply does not understand it or does not receive the information they need because the writer left it out. Machines may learn to be better readers than we are, but even machines are not going to learn to read information that just isn't there.

We write better for robots than we do for humans

Actually, one of the advantages of the relative stupidity of robots is that it forces us to be very careful in how we create and structure data for machines to act on. The computer

³https://en.wikipedia.org/wiki/Curse_of_knowledge

⁴<http://everypageispageone.com/2015/08/04/the-economy-of-language-or-why-we-argue-about-words/>

science community coined the phrase “garbage in, garbage out” very early, because the machines were, and to a large extent still are, too stupid to know when the information they were taking in was garbage, and did not have the capacity, like human beings, to seek clarification or consult other sources. They just spit out garbage.

This meant that we had to put a huge emphasis on improving the quality and precision of the data going in. We diligently worked out its structures and put elaborate audit mechanisms in place to make sure that it was complete and correct before we fed it to the machine.

We have never been as diligent in improving the quality of the content that we have fed to human beings. Faced with poor content, human beings do not halt and catch fire; they either lose interest or do more research. Given our adaptability as researchers and our tenacity in pursuing things that really matter to us, we often manage to muddle through bad content, though at considerable economic cost. And the distance that often separates writers from readers means that the writers often have no notion of what the poor reader is going through. If readers did halt and catch fire, we might put more effort and attention into content quality.

Even today, when a huge emphasis is being placed on enterprise content management and the ability to make the store of corporate knowledge available to all employees, most of the emphasis is on making content easier to find, not on making it more worth finding. (This despite the fact that the best thing you can do to make content easy to find is to make it more worth finding.) People trying to build the semantic web spend a lot of time trying to make the data they prepare for machines correct, precise, and complete. We don’t do nearly as much for humans. Until we do, deep learning alone may not be enough to make the human web the semantic web.

Part of the problem has always been that improving content quality runs up against the curse of knowledge. Both the authors who create the content and most of the subject matter experts who review it suffer from the curse, meaning that there are few effective ways to audit written content. Style guides and templates can help remind authors of what is needed, but their requirements are difficult to remember and compliance is hard to audit, meaning there is little feedback for an author who strays.

The curse of knowledge and the distance separating writers from readers are a major source of complexity in the content creation process. Structured writing provides a way to guide and audit content for quality. My reply to the people who ask me whether structured writing is relevant, therefore, is “garbage in, garbage out”. Structured writing enables us to write in the subject domain (wholly or partially) and this allows us to guide and audit in ways not otherwise possible. It also allows us to factor out many constraints, simplifying the author’s task, and therefore allowing them to give more of their attention to the writing task. These things make content better, whether that content is going to be read by people or by robots.

Structure, art, and science

To many writers, the idea that imposing constraints can improve quality is controversial. Many see quality writing as a uniquely human and individual act, an art, not a science, something immune to the encroachment of algorithms and robots. But I would suggest that the use of structures and algorithms as tools does not diminish the human and artistic aspects of writing. Rather, it supplements and enhances them.

And I would suggest that this is a pattern we see in all the arts. Music has always depended on the making and the perfecting of instruments as tools of the musician. Similarly the mathematics of musical theory gave us well tempered tuning, on which modern Western music is based.

Computer programming is widely regarded as an art⁵ among its practitioners, but the use of sound structures is recognized as an inseparable part of that art. Art lies not in the rejection of structure but in its mature and creative use. As noted computer scientist Donald Knuth observes in his essay, *Computer Programming as an Art*, most fields are not either an art or a science, but a mixture of both.

Apparently most authors who examine such a question come to this same conclusion, that their subject is both a science and an art, whatever their subject is. I found a book about elementary photography, written in 1893, which stated that “the development of the photographic image is both an art and a science”. In fact, when I first picked up a dictionary in order to study the words “art” and “science,” I happened to glance at the editor’s preface, which began by saying, “The making of a dictionary is both a science and an art.”

—<http://dl.acm.org/citation.cfm?id=361612>

As writers we can use structures, patterns, and algorithms as aids to art, just like every other profession.

Of course, few writers would claim that there is no structure involved in writing. We have long recognized the importance of grammatical structure and rhetorical structure in enhancing communication. The question is, can the type of structures the structured writing proposes improve our writing, and if so, in what areas? Traditional poetry is highly structured, but it is doubtful that using an XML schema would help you write a better sonnet. On the other hand, it is clear that following the accepted rhetorical pattern of a recipe would help you write a better cookbook, and using structured writing to create your recipes can help you both improve the consistency of your recipes and to produce them more efficiently and exploit them as assets in new ways.

The question then becomes, how much of our work is like recipes and would benefit from structured writing, and how much is like sonnets and would not. The answer, I believe,

⁵<http://ruben.verborgh.org/blog/2013/02/21/programming-is-an-art/>

is that a great deal of business and technical communication, at least, can benefit greatly. If you look at much of that communication and see no obvious structure, I would suggest that this is not evidence that structure is inappropriate, but that appropriate structure has not been developed and applied to the content.

Contra-structured content

We must also acknowledge that many writers have had a bad experience with structured writing. In many of these cases, the structured writing system was not chosen or designed by the writers to enhance their art; it was imposed externally for some other purpose, such as to facilitate the operation of a content management system or make it easier to reuse content. In other words, they were designed to shift complexity away from some other function without sufficient thought about where that complexity would go, and it ended up being dumped on the writers. In some cases, these systems actively interfere with the author's art and directly hinder the production of quality content.

Writers who have had bad experiences with structured content have usually been faced with structures that were not designed for the writer's purpose. Such content is not merely unstructured for these author's purposes, it is actually contra-structured. It has an enforced structure that actively gets in the way of the author doing their best work.

I talk to authors all the time who show me page designs and layouts that make no sense, lamenting that the system does not give them any other choices. Content structure is not generic, and you cannot expect to simply install the flavor of the month CMS or structured writing system and get a good outcome.

Properly applied, however, as a means to guide and enhance the work of authors, structured writing can substantially improve content quality. In upcoming chapters, we will look at the algorithms of structured content, many of which relate directly to the enhancement of content quality.

Until the robots take over

Of course, this all supposes that the machines are not becoming better writers than us as well. Companies like Narrative Science are working on that, but I don't know if they are as far along that path as the deep learning folks are in teaching computers to read.

Do robots suffer from the curse of knowledge? Maybe not. But current writing robots certainly work with highly structured data, so structured writing is still key to quality content even when the robots do come for our keyboards.

Actually, according to James Bessen's recent article in *The Atlantic*, The Automation Paradox⁶, automation does not decimate white collar jobs the way we have been told to

⁶<http://www.theatlantic.com/business/archive/2016/01/automation-paradox/424437/>

fear. By reducing costs, it increases demand, resulting in net growth of jobs, at least for people who learn to use the new technology effectively.

That said, all the semantic technology and content management in the world is not going to make the difference it should until we improve the quality of content on a consistent basis. Structured writing, particularly structured writing in the subject domain, is one of our best tools for doing that.

Other sources of quality improvement

Besides assisting writers to produce better and more consistent rhetorical structures, there are other ways in which structured writing can contribute to content quality. In fact, most of the algorithms we are going to look at make a contribution to content quality in one way or another. In particular, the ability to hand certain complex tasks off to algorithms can provide a big boost to quality by enabling us to produce content structures that would be too laborious or too complex to produce by hand. The subject-domain linking methods that I will describe in Chapter 15, *Linking*, for instance, allow you to produce a consistent and rich linking scheme with minimal management overhead. Reproducing this using unstructured methods would be very costly and error prone. Indeed, the producing the rich and complex relationships between content pieces that create what I call a bottom-up information architecture, as described in Chapter 21, *Information Architecture*, would be very difficult without the right set of structured writing techniques.

The scale of the content manangement problem, and our awareness of the consequences for our readers when we fail to handle all the complexity of content creation, have increased dramatically in recent years, greatly raising the bar on content quality. To handle any process on this scale successfully without a significant application of algorithms to the content system and its processes, is not feasible. To enable those algorithms, we need structured writing.

Chapter 8. Writing

A comprehensive approach to managing the content creation and delivery process means partitioning and directing the complexity of that process to the right people and processes. While there are parts of that process that are separate from the tasks of the writer, most of the complexity of the content process flows through the writer. And because the writer's attention is a limited and valuable resource that is indispensable for the creation of quality content, much of the effort in designing the content process should be focused on partitioning and directing complexity away from the writer.

However, because complexity cannot be destroyed, only redirected, it is vital that we don't drop any of that complexity in the process of directing it away from the writer, because then it falls on the reader. This means that when we move a chunk of complexity out of the writer's world, we always have to ask the writer to supply enough information to complete the transfer without loss. Thus when we use styles to partition and direct the complexity of document formatting to a designer and an algorithm, we have to ask the writer to apply those styles correctly to the content they create.

Structured writing can partition and redirect a huge amount of the complexity of content creation – including complexity that you are just not handling now, and which is therefore falling on your readers – but it all starts with writers creating the right structures as they write.

Structures are the interfaces between the partitions in which we contain the complexity of our processes. Just as software developers partition software complexity behind an API, so information architects and content engineers partition content complexity behind content structure expressed as structured writing markup.

These structures are a system of constraints. They tell writers what is and is not a valid way to structure their writing. Constraints are the substance of partitioning. They tell you both what you cannot do and what you must do both to maintain the integrity of the partition, and to pass the required information into the partition so that it can play its proper part in the overall content system, ensuring that no complexity goes unhandled.

Our structured writing system, therefore, can only be as good as the structured content our writers create. Getting the best possible structure from our writers is key to all of the algorithms and all of the benefits of structured writing. This is not something we can afford to leave to chance. We need to be systematic about it.

All forms of structured writing, even in the media domain require writers to do something other than simply write. Since writing is an intellectually challenging activity that requires full attention, adding structured writing requirements into the mix necessarily takes away some of the attention that should otherwise be given to content, which obviously has the potential to reduce the quality of the content. Clearly, therefore, we need to make sure that the intrusion of structured writing requirements onto the writing process is as minimal as possible and that it removes more complexity than it introduces.

Quality writing does not result from a writer simply spilling words onto a page or screen in stream of consciousness fashion. Writing is a design activity. It creates a structure of words that conveys complex ideas and information about the real world. It very much matters that the writer says the right things using the right words in the right order. If structured writing techniques can help with this literary design work, they can lessen the intellectual burden on the writer, and thus potentially improve the quality of the content. This is a major area in which structured writing can redirect complexity away from the writer, particularly when the writer is an occasional contributor.

Of course, structured writing can improve content quality in other ways. Most of the algorithms we will look at pertain to content quality in one way or another, such as improved linking or better management of terminology. Still, these algorithms work on the structures that the writers create. If those structures are incorrect or inconsistent, there is a limit to what downstream algorithms can do to improve quality. It really all begins with getting writers to reliably create the required structures as they write.

In media-domain systems like word processors and desktop publishing systems, the writer is asked to think about formatting structures while writing. One of the traditional arguments for structured writing is to relieve the writer of the burden of thinking about (and manipulating) formatting so they can focus on writing. This means moving to the document domain. But in the document domain, the writer has a new set of structures to think about: document domain structures. Is it easier on the writer to think about and manipulate document domain structures rather than media domain structures? In some cases, yes. For instance, writing a blog post or a web page in MarkDown may be less cumbersome for some writers than using a WYSIWYG HTML editor.

However, Markdown does not contain enough structure, or enough constraints on its structure, to enable many of the structured writing algorithms we will be looking at. If we want to support these algorithms, we will need something more structured, and this can easily mean something that requires more of the writer's attention. If we are proposing to implement management-intensive algorithms, such as reuse, it can mean that writers need to learn and manipulate an entire management system and the management policies that the organization puts in place around it. Depending on how complex these policies are and how foreign they are to the writer's experience, this can create a burden far greater than that of creating and manipulating formatting according to a style guide.

We could look at this and say, okay, yes, writing is now more difficult and more complex than it was before because of all this additional structure and all that writers need to learn to apply that structure, but we are getting additional advantages as well, such as reuse and content management, so it is worth it overall. The problem is, attention is a finite resource. Writing is a task that takes full attention, so any requirement added to the task takes attention away from writing. As writing gets more difficult, writers do all of the component tasks less well. The more of the writer's attention is required on structure, the less is available for writing, and the quality of the writing suffers. The writer is dealing with more complexity than they are equipped to handle and some of it inevitably falls through to the reader.

As the quality of the writing suffers, the writer becomes frustrated with the system, and becomes more interested in getting their ideas down than in obeying the onerous structural rules that are getting in their way. When that happens, the quality of the structure suffers as well. And if both the quality of the writing and the quality of the structures decline, all of your algorithms become less reliable, compromising all of benefits you hoped to obtain. Too much complexity dumped on any one person or process compromises all downstream persons and processes.

To look at this another way, the more complex your system becomes, and the more algorithms you are attempting to support, the more important conformance to its constraints becomes. But conformance is fundamentally a human activity. It is that writer who must conform. Good conformance results from creating structures that are easy to conform to. It all begins with writing. Unfortunately, writing is often the last thing people think about in designing content management and structured writing systems. It is the place that complexity get dumped before it eventually gets dumped on readers.

One of the most familiar tropes of the content management industry is that problems with content management systems are not technology problems, they are human problems. The solution, this trope suggests, lies in better change management and more training. The presumption here is that the tools work fine if only you give them correct input. If the input is not correct, that is the fault of the humans who created the input. But this is an argument we would not accept for any other kind of system. For any other kind of system we would say, “this system is too hard to use”, not “the problem is everybody needs to be better trained and more accepting of change”. The real fault here is poor system design. If humans cannot conform to the structures that the system requires, the fault is in the system design. It is not partitioning and distributing complexity correctly. The structures should be redesigned to be easier to conform to. (We will look at this in more detail in Chapter 16, *Conformance*.)

How is designing structures for ease of writing consistent with designing them to match the specific constraints that we want to impose for the sake of quality and efficient processing? As we have seen, moving from the media domain to the document domain allows us to factor out or impose certain structural constraints, but often require the introduction of the management domain to impose content management constraints. These complexities detract from ease of writing. But we have also seen that by moving to the subject domain we can factor out many of the document domain and management domain structures. Designing structures for writing, therefore, often consists of factoring out complex publishing and content management structures using subject-domain structures that are lucid for writers.

One of the most important consequences of this, both for ease of writing and reliability of data, is that in the subject domain you are not asking the writer to think, and to structure content, in terms of algorithms. In this sense, the move to the subject domain not only factors out specific constraints from the writer, it factors out the need to think in algorithms at all, leaving the writer free to think in terms of subject matter. This freedom to focus on content is a property I call functional lucidity.

Functional lucidity

Functional lucidity means the way that you actually use language when you are writing, which is to say the way that you use language when you are in the throes of figuring out what you want to say and how you want to say it. If you are asked to add markup to your content as you write, if you are asked to shape your content according to the constraints that a structured writing language dictates, then the lucidity of that markup and the structures it defines are vital to your success. The names of the structures, the order in which they occur should spring into your mind as readily (if not more so) that the words and phrases and ideas you are trying to record.

If you have ever tried to learn a language, you know how painful it is even to write a paragraph in a language you are not fluent in. The effort of finding words and correct grammatical structures takes all of the attention that should be reserved for what you are trying to say. Writing in a markup language where the structures don't make intuitive sense, where they don't seem to fit the thoughts you are trying to express, is very much like this. Lucidity is essential to avoid having the markup absorb all of the attention that should be focused on the content.

Functional lucidity is not an absolute property, of course. What is lucid for one writer may be opaque to another. In particular, professional technical writers who have been used to writing in structured document domain templates in applications like FrameMaker may find a markup language like DocBook functionally lucid, whereas someone not used to thinking in these terms would find it difficult and distracting. On the other hand, those writers used to FrameMaker often find DITA's structure difficult to get used to because they do not find its approach to topics lucid. To still others it seems very natural.

But while different writers may have different degrees of experience and familiarity with abstract document structures, all writers should have familiarity with the subject matter they are writing about. Thus a well-designed subject domain language tends to naturally have functional lucidity for everyone who is likely to use it.

Simplicity and Clarity

One of the biggest benefits of subject domain markup for writers is a much higher degree of functional lucidity compared with a typical document domain language.

While a general document domain language like DocBook needs to have a wide range of document structures, a recipe markup language such as we developed in Chapter 5, *Writing in the Subject Domain*, has only a few simple structures. Better still, there are very few permutations of those structures.

Because subject domain structure describe the subject matter they contain, they are also much clearer to writers, who may not understand complex document structures (or, more

often, the subtle distinctions between several similar document structures), but who do (we hope) understand their subject matter.

The combination of simplicity and clarity mean that in many cases you can get writers to create subject-domain structured content with little or no training. For instance, even if we add some additional fields to our recipe markup, you could still hand a sample like the one below to an writer and ask them to follow it as a template, without giving them any training or any special tools.

```
recipe: Hard Boiled Egg
introduction:
    A hard boiled egg is simple and nutritious.
ingredients:: ingredient, quantity
    eggs, 12
    water, 2qt
preparation:
    1. Place eggs in pan and cover with water.
    2. Bring water to a boil.
    3. Remove from heat and cover for 12 minutes.
    4. Place eggs in cold water to stop cooking.
    5. Peel and serve.
prep-time: 15 minutes
serves: 6
wine-match: champagne and orange juice
beverage-match: orange juice
nutrition:
    serving: 1 large (50 g)
    calories: 78
    total-fat: 5 g
    saturated-fat: 0.7 g
    polyunsaturated-fat: 0.7 g
    monounsaturated-fat: 2 g
    cholesterol: 186.5 mg
    sodium: 62 mg
    potassium: 63 mg
    total-carbohydrate: 0.6 g
    dietary-fiber: 0 g
    sugar: 0.6 g
    protein: 6 g
```

Of course, the downside is that recipe markup is only good for one thing: recipes. Complexity is never destroyed, only moved somewhere else. So this approach moves complexity away from the writer to the person who has to design and maintain these structures and the algorithms that process them. (Information Architect and Content Engineer are both titles sometimes used for the person with this responsibility.)

This can seen scary because we are not used to partitioning complexity in this way. But then, we also recognize that the way we have partitioned complexity in the past has not been as successful as we would like – that it has, in fact, left much of the complexity of content creation unhandled, resulting in that complexity being dumped on our readers. If a new partitioning of the complexity of the content creation process is required, we have to accept that some of these new methods of partitioning will be unfamiliar. They are the place where previously unhandled complexity that used to get dumped on the reader is now being handled.

Chapter 9. Separating Content from Formatting

If there is one phrase that most people associate with structured writing, it is “separating content from formatting”. This is the most basic and well-known form of partitioning and redirecting of complexity that we do in structured writing, and it illustrates the basic method by which we do all the rest. But partitioning content from formatting is not as simple as it may seem. So let’s take a look at it one step at a time.

Separate out style instructions

Let’s start with a simple bit of content that includes a description of its format. I’m going to use CSS syntax to describe the format, but that is not significant. This is just to show what we are doing when we do the separation. I’m also going to represent certain characters by their names in square brackets, just so we can see exactly where everything is going:

```
{font: 10pt "Open Sans"}The box contains:  
{font: 10pt "Open Sans"}[bullet][tab]Sand  
{font: 10pt "Open Sans"}[bullet][tab]Eggs  
{font: 10pt "Open Sans"}[bullet][tab]Gold
```

This file contains content and formatting, so let’s separate the two. Of course, when we remove the formatting from the content we are going to have to add something in its place so we can add the formatting back later. The algorithm is not really “separate content from formatting” but “separate formatting from content and replace it with something else”.

The simplest thing to replace it with is a style:

```
{style: paragraph}The box contains:  
{style: paragraph}[bullet][tab]Sand  
{style: paragraph}[bullet][tab]Eggs  
{style: paragraph}[bullet][tab]Gold
```

Then, of course, we need to record the style (we are separating it, not eliminating it altogether):

```
paragraph = {font: 10pt "Open Sans"}
```

Now that they are separated, we have the choice of substituting different formatting by changing the definition of the style, while leaving the content alone:

```
paragraph = {font: 12pt "Century Schoolbook"}
```

Separate out formatting characters

Cool, but suppose we would like to change the style of the bullet we use for lists. The style of bullet used is certainly part of what we would consider “formatting”, but bullets are text characters. To change them you don’t just have to change the font applied to the characters, you have to change the characters themselves.

Sometimes the typed characters in your text are part of the content, and sometimes they are part of the formatting. So now we need to extend our idea of a style to include characters.

```
paragraph = {font: 12pt "Century Schoolbook"}  
bullet-paragraph = {font: 12pt "Century Schoolbook"}[bullet]
```

Now our content looks like this:

```
{style: paragraph}The box contains:  
{style: bullet-paragraph}[tab]Sand  
{style: bullet-paragraph}[tab]Eggs  
{style: bullet-paragraph}[tab]Gold
```

Except that now the writer will be starting the bulleted lines with a tab, which is awkward and probably error prone, so we move that character to the style as well.

```
paragraph = {font: 12pt "Century Schoolbook"}  
bullet-paragraph = {font: 12pt "Century Schoolbook"}[bullet][tab]
```

Now our content looks like this:

```
{style: paragraph}The box contains:  
{style: bullet-paragraph}Sand  
{style: bullet-paragraph}Eggs  
{style: bullet-paragraph}Gold
```

And now you can change the bullet style:

```
bullet-paragraph = {font: 12pt "Century Schoolbook"} [em dash] [tab]
```

And then we maybe realize that “bullet-paragraph” is not the best name any more, because the style is now a dash, not a bullet. In other words we discover that we had not done as good a job as we thought of partitioning content and formatting, because the content actually still contains formatting information in the form of a style that is named for a particular piece of formatting.

Name your abstractions correctly

When we separate formatting from content, we have to insert something in its place, and it matters what that something is and what it is called. If we call it the wrong thing we set up a false expectation, and that will lead to authors using it incorrectly, which will mean we can't format it reliably. This is simply poor partitioning.

So the first lesson about the algorithm of separating content from formatting is that it matters what you call things. When you do this, you are creating an abstraction, and you need to figure out what that abstraction is and name it appropriately.

So what is the abstraction here? It is a list, of course. So maybe we do this:

```
{style: paragraph}The box contains:  
{style: list-item}Sand  
{style: list-item}Eggs  
{style: list-item}Gold
```

and

```
list-item = {font: 12pt "Century Schoolbook"} [em dash] [tab]
```

Make sure you have the right set of abstractions

But then, of course, we run into this problem:

```
{style: paragraph}To wash hair:  
{style: list-item}Lather  
{style: list-item}Rinse  
{style: list-item}Repeat
```

Here our list should have numbers, not dashes or bullets. So we realize that the abstraction we are after is not so broad as all list items. We look at the differences between the

Separating Content from Formatting

different kinds of list items we use and try to group them into abstract types and come up with names for those types. Maybe we come up with “ordered-list-item” and “unordered-list-item”. Then we have:

```
{style: paragraph}The box contains:  
{style: unordered-list-item}Sand  
{style: unordered-list-item}Eggs  
{style: unordered-list-item}Gold
```

and

```
{style: paragraph}To wash hair:  
{style: ordered-list-item}Lather  
{style: ordered-list-item}Rinse  
{style: ordered-list-item}Repeat
```

And the style for ordered-list-items now looks something like this:

```
ordered-list-item = {font: 12pt "Century Schoolbook" }<count>. [tab]
```

And then we realize that we need a way to increment the count and to reset it to 1 for a new list. So we have:

```
{style: paragraph}To wash hair:  
{style: first-ordered-list-item}Lather  
{style: ordered-list-item}Rinse  
{style: ordered-list-item}Repeat
```

and

```
first-ordered-list-item =  
    {font: 12pt "Century Schoolbook" }<count=1>. [tab]  
ordered-list-item =  
    {font: 12pt "Century Schoolbook" }<++count>. [tab]
```

(++count here means add one to count and then display it.)

And this is pretty much how you do lists in FrameMaker or Microsoft Word today, as well as a number of other tools. But the reason for going through it in such detail is to point out what is involved in even this simple bit of partitioning. We began by simply removing formatting commands, but then started to remove characters as well, which forced us

to include characters in our style definitions, and then to be able to actually calculate characters in our style definitions. And we saw that in performing these separations, we were creating abstractions, and that it was important to consider all the cases we might run into and create the appropriate abstractions to handle them.

Create containers to provide context

As we noted in Chapter 4, *Writing in the Document Domain*, one problem with this approach is that the writer has to remember to apply a different style to the first item of a list. It would be better if they could use the same style for each list item and have the numbering just work. But this is hard to do because there is nothing in the content to say where one numbered list ends and the next begins. For this we need a new abstraction. So far we have abstractions for two kinds of list items: ordered and unordered list items. But we don't have an abstraction for lists themselves.

So far, we have been separating content from formatting purely in the media domain. We have replaced direct formatting definitions with indirect definitions through styles. The only thing that abstracts any of this beyond the media domain is the names that we have given to the styles that we have created. But now we start to venture into the document domain, creating the abstract idea of a list and inserting that abstract idea into our content.

```
paragraph: To wash hair:  
list:  
    ordered-list-item:Lather  
    ordered-list-item:Rinse  
    ordered-list-item:Repeat
```

There are a number of significant changes here. First, our structure is no longer flat. We have introduced the idea of a container. A list is a container for list items. In creating this container we have added something to the content that was not there before. Previously it was a series of paragraphs with different styles attached. Now we have a container, which, as far as the formatting is concerned, simply never existed in the original. The writer and reader knew that the sequence of bulleted paragraphs formed a list, but that was an interpretation of the formatting. Now we have taken that interpretation and instantiated it in the content itself. Those lines starting with bullets constitute a list, and now we have made it explicit.

By creating the idea of a list, we are able to further separate list formatting from the content of the list – because now an algorithm, one I will call the formatting algorithm, can recognize it as a list and can make formatting decisions based on that knowledge.

The second important thing that has happened here is that the content no longer contains references to style names. Instead we have structures. `list` is a structure and so are `paragraph` and `numbered-list-items`.

We have replaced styles with structures because the same structure may get a different style depending on where it is in the document. The formatting algorithm is responsible for determining if an ordered-list-item is the first one inside a list and formatting it accordingly. (Which is just how list formatting works in CSS¹.)

Now authors no longer apply styles to content, even ones with abstract names. Rather they place content in structures and allow the formatting algorithm to apply styles appropriately. The content is separated even more from the formatting.

Move the abstractions to the containers

But there is an obvious problem here. What if an author inadvertently does this:

```
paragraph: To wash hair:  
list:  
    ordered-list-item:Lather  
    unordered-list-item:Rinse  
    ordered-list-item:Repeat
```

To avoid this, we move the abstraction outward. Instead of ordered and unordered list items, we create ordered and unordered lists:

```
paragraph: To wash hair:  
ordered-list:  
    list-item:Lather  
    list-item:Rinse  
    list-item:Repeat
```

and

```
paragraph: The box contains:  
unordered-list:  
    list-item:Sand  
    list-item:Eggs  
    list-item:Gold
```

And, of course, the list-item structure can be used in either an unordered list or an ordered list, because it is a list item in either case, and the formatting algorithm can tell

¹<https://css-tricks.com/numbering-in-style/>

the difference based on which type of list it belongs to. The name `list-item` describes its role in the document in a way that is entirely separated from how it will be styled.

Moving the abstraction out to the container is an important part of the algorithm of separating content from formatting. It partitions ordered and unordered lists making the construction of each simpler and more reliable. This helps keep things consistent and reduces the number of things authors have to remember.

Creating containers and abstracting out the differences between their contents is an important piece of separating content for formatting. For example, HTML and Markdown both provide six different levels of headings. But content under an `H2` or an `H5` heading is not in any container. The content simply comes after the heading. This means that it is perfectly possible and legal in these languages to place different heading elements in any order you want. Writers have to pay attention to which heading level they are creating and how it fits in the structure of the document they are creating.

By contrast, in DocBook, we have a `section` structure. Like a list, a section is part of the writer's interpretation of what they are creating in the document, but it is only implied, not instantiated by the formatting. The `section` structure instantiates the concept of a section. And once we have the instantiation of a section, we don't need six levels of heading. We can have one structure called `title`. Sections can be nested inside other sections, and the formatting algorithm can apply the correct style to the title based on context:

```
section:  
  title:  
  paragraph:  
  section:  
    title:
```

This will eliminate incorrect heading element choices, ensuring that the headings in the output consistently reflect the section and subsection structure of the document.

(It must be said that not everyone holds to the view that headings in a text do or should reflect a hierarchy of sections. Sometimes they may be simply signposts along the way, and like any signpost, the size of the sign reflects the size of the town, not a strict hierarchy of sign sizes. So if that is how you look at document structures, you should choose a different way to separate content from formatting in your content.)

Separate out abstract formatting

We noted that in the case of ordered and unordered lists, separating content from formatting actually involves separating out some of the content as well. Or rather, that it involves separating out some of the characters. In other words, the distinction between

Separating Content from Formatting

what is represented in a document using character codes and what is represented in other data structures is not necessarily the same as the logical distinction between content and formatting from a structured writing point of view.

Consider a structure that we might call a labeled list:

Street	123 Elm Street
Town	Smallville
Country	USA
Code	12345

The generic structure of a labeled list might look like this:

```
labeled-list:  
    list-item:  
        label: Street  
        content: 123 Elm Street  
    list-item:  
        label: Town  
        contents: Smallville  
    list-item:  
        label: Country  
        contents: 123 USA  
    list-item:  
        label: Code  
        contents: 12345
```

But what if you have hundreds of addresses, all with the same labels. In this case, are the labels really content, or are they formatting? Since they don't change from one list to another, we could look at them as being part of how the content is presented, rather than being part of the content itself. So we look for a way to separate them from the content.

As always, when we separate something from our content, we have to replace it with something else, and that something is generally an appropriately named structure. So that gets us a structure like this:

```
address:  
    street: 123 Elm Street  
    town: Smallville  
    country: USA  
    code: 12345
```

Separating Content from Formatting

Now, of course, we have moved our content into the subject domain. At this point we should make a distinction between formatting and presentation. Though these two words are often used interchangeably, we need to make a distinction between them for structured writing purposes. For purposes of this discussion, the formatting of text is the precise details of its appearance: the font chosen, the width of the text column, the size of the characters, the spacing between line, the size and shape of the bullet characters. Presentation, on the other hand, is about how we organize the content. Deciding to use a list to present a certain piece of information is a presentation decision, and it is independent of the formatting details we apply to lists. When we move content from the media domain to the document domain, we separate the formatting of the content from the presentation of the content. The decision to present the information as a list remains; the decision about what that list will look like has been separated out.

When we move the content from the document domain to the subject domain, however, we separate the information from the presentation. The subject domain structure above is not a labeled list. It is a data record that could be turned into many different forms of presentation.

The job of turning such structures into a specific form of presentation is the job of the presentation algorithm, which we will look at in more detail in Chapter 27, *Publishing*. The presentation algorithm could turn it into a labeled list, a table, a paragraph (with the fields separated by commas), or the address label for an envelope.

In the subject domain, with the content separated from both formatting and presentation, we also gain the ability to query and reorganize the content in various interesting and useful ways (which we will explore in later chapters).

This is as far as we can go in separating content from formatting, and we can't separate all content from formatting to quite this extent. It should be clear at this point that separating content from format is not a binary thing. There are various stages of separation that we can achieve for various reasons. It is important to understand exactly which degree of separation will best serve your needs.

Chapter 10. Processing Structured Text

In the last chapter we looked at separating content from formatting. But of course they cannot stay separated. Content must be formatted before it is presented to the reader. So let's look at the algorithm for putting them back together again. Understanding the basics of algorithms is important to understanding structured writing even if you don't intend to code the algorithms yourself. Understanding how things can be put back together is a big part of understanding how to separate them. Indeed, it is only by describing how you would put them back together that you know for certain that you did not lose anything when you did the separation. This is essential to ensuring a clean partitioning that does not allow any complexity to fall through to the reader.

Two into one: reversing the factoring out of invariants

Moving content from the media domain to the document domain or the subject domain involves progressively factoring out invariants in the content. Each step in this process creates two artifacts, the structured content and the invariant piece that was factored out.

Processing structured text is about putting the pieces back together: combining the structured content with the invariants that were factored out. If factoring out the invariants moves content toward the document or subject domains, recombining the content with the invariants moves it in the opposite direction, toward the media domain. This could mean moving the content from the subject domain to the document domain or from the document domain to the media domain, or simply from a more abstract form in the media domain to a more concrete form, which will be our first example.

Adding back style information

The first example of separating content from formatting that we looked at involved factoring out the style information from this structure:

```
{font: 10pt "Open Sans"}The box contains:  
{font: 10pt "Open Sans"}[bullet][tab]Sand  
{font: 10pt "Open Sans"}[bullet][tab]Eggs  
{font: 10pt "Open Sans"}[bullet][tab]Gold
```

We replaced the style information with style names:

```
{style: paragraph}The box contains:  
{style: bullet-paragraph}Sand
```

```
{style: bullet-paragraph}Eggs
{style: bullet-paragraph}Gold
```

And then we defined the styles:

```
paragraph = {font: 10pt "Open Sans"}
bullet-paragraph = {font: 10pt "Open Sans"}[bullet][tab]
```

To unite the styles with the appropriate paragraphs, we can write as set of simple search and replace rules:

```
find {style: paragraph}
replace {font: 10pt "Open Sans"}

find {style: bullet-paragraph}
replace {font: 10pt "Open Sans"}[bullet][tab]
```

I said at the beginning that the basic processing algorithm was to combine two source of information to create a new one. Where are these two sources? The first source is the structured text. The second source is the style definitions, and they are embedded in the rules themselves. This is how it is usually done. In some cases, though, the rules may pull content from a separate file. We will see examples of this later.

The result of applying these rules is that we get back the original content:

```
{font: 10pt "Open Sans"}The box contains:
{font: 10pt "Open Sans"}[bullet][tab]Sand
{font: 10pt "Open Sans"}[bullet][tab]Eggs
{font: 10pt "Open Sans"}[bullet][tab]Gold
```

If we want to change the styles, we can apply a different set of rules:

```
find {style: paragraph}
replace {font: 12pt "Century Schoolbook"}

find {style: bullet-paragraph}
replace {font: 12pt "Century Schoolbook"} [em dash][tab]
```

Applying these rules will result in the a change in the formatting of the original content:

```
{font: 12pt "Century Schoolbook"}The box contains:
{font: 12pt "Century Schoolbook"} [em dash][tab]Sand
{font: 12pt "Century Schoolbook"} [em dash][tab]Eggs
```

{font: 12pt "Century Schoolbook"}[em dash][tab]Gold

Rules based on structures

The tools that do this sort of processing do not literally use search and replace like this. Rather, they parse the source document to pull out the structures and allow you to specify your processing rules by referring to those structures.

We are not concerned here with the actual mechanism by which a processing tool recognizes structures. We are concerned with what to do when a structure is found. So let's rewrite our rules to match structures rather than find literal strings in the text:

```
match paragraph
    apply style {font: 12pt "Century Schoolbook"}

match bullet-paragraph
    apply style {font: 12pt "Century Schoolbook"}
    output "[em dash][tab]"
```

The result of applying these rules is just the same as before:

```
{font: 12pt "Century Schoolbook"}The box contains:
{font: 12pt "Century Schoolbook"}[em dash][tab]Sand
{font: 12pt "Century Schoolbook"}[em dash][tab]Eggs
{font: 12pt "Century Schoolbook"}[em dash][tab]Gold
```

The way I have written these rules is an example of what is called pseudocode. It is a way for human being to sketch out an algorithm to make sure that they understand what they are trying to do before they write actual code. There is no formal grammar or syntax to pseudocode. It is intended for humans, not computers, and you can use whatever approach you like as long as it is clear to your intended audience. But pseudocode should clearly lay out a set of logical steps for accomplishing something. It should make clear exactly how the pieces go together.

Writing algorithms in pseudocode is a great way to make sure that we understand the algorithms we are creating without worrying about the details of code – or even learning how to code. They are also a great way to communicate to actual coders about what we need a program to do.

The order of the rules does not matter

You may have noticed that what these rules are doing is pretty much exactly what style sheets do in an application like Word or FrameMaker. In fact, it is exactly what a style

sheet does. If you understand stylesheets, you understand a good deal of how structured writing algorithms work.

One important thing to notice is that when you create a stylesheet in Word or FrameMaker, you don't specify the order in which styles will be applied to the document. The same is true when you create a CSS stylesheet for the Web. The stylesheet is just a flat list of rules. The order in which the rules are applied to the document depends entirely on the order in which the various structures occur in the document.

This may seem very obvious, but it is key to understanding how structured text is usually processed. It is a subject that is sometimes quite confusing to people who have been trained to write procedural computer programs, which is why I am making a point of calling it out.

Things get a tiny bit more complex when we move into processing the nested structures of the document domain and subject domain, but the basic pattern of a set of unordered rules to describe a transformation algorithm still applies.

Applying rules in the document domain

Suppose we have a piece of document domain structured text that contains this `title` structure:

```
title: Moby Dick
```

We want to transform this document into HTML. When our rule matches a structure in the source document, it outputs the equivalent HTML structure. Here is the pseudocode for this rule (it is in a slightly different format from the pseudocode above):

```
match title
    create h1
        continue
```

This says, when you see a `title` structure in the source, create an `h1` structure in the output, and then continue applying rules to the content of the `title` structure.

The `continue` instruction is indented under the `create h1` instruction to indicate that the results of continuing will appear inside the `h1` structure.

In our pseudocode, we are assuming that the text content of each structure will be output automatically (as is the case in many tools), so the output of this rule (expressed in HTML) is:

```
<h1>Moby Dick</h1>
```

But suppose that there is another structure inside the title in our source. In this case it is an annotation of part of the title text:

```
title: Review of {Rio Bravo} (movie)
```

Here the annotated text is set off with curly braces and the annotation itself in parentheses immediately after it. (This is a feature of the SAM markup syntax that I am using for most of the examples in this book.) So the annotation says that the words “Rio Bravo” refer to a movie. The annotation is a content structure, just like the title structure, and is nested inside the text of the title.

So what do we do with our rule for processing titles to make it deal with `movie` annotations embedded in the title text? Absolutely nothing. Instead, we write a separate rule for handling `movie` annotations no matter where they occur:

```
match movie
    create i
    continue
```

When the processor hits `continue` in the `title` rule, it processes the content of the title structure. In doing so, it encounters the `movie` structure and executes the `movie` rule. The result is output that looks like this:

```
<h1>Review of <i>Rio Bravo</i></h1>
```

The `continue` instruction is really all we need to add to our rules to allow them to deal with nested structures. They remain an unordered collection of rules, just like a stylesheet. (In fact, XSLT, a language that implements this model, calls a set of processing rules a “stylesheet”.)

Processing based on context

When we move to the document domain, we use context to reduce the number of structures that we need. For example, where HTML has six different heading structures, H1 through H6), DocBook has only one: `title`, which can occur in many different contexts. So how do we apply the right formatting to a title based on its context? We create different rules for the `title` structure in each of its contexts. We express the context by listing the parent structure names separated by slashes:

```
match book/title
    create h1
    continue
```

```

match chapter/title
    create h2
    continue

match section/title
    create h3
    continue

match figure/title
    create h4
    continue

```

Now here is the clever bit. You don't have to change the `movie` rule to work with any of these versions of the `title` rule. Suppose our title is the title of a section, like this:

```

section:
    title: Review of {Rio Bravo}(movie)

```

When we process this with our rules, the `section/title` rule will be executed to deal with the title structure, and the `movie` rule will be executed when the `movie` structure occurs in the course of processing the content of the `title` structure, with the following result:

```
<h3>Review of <i>Rio Bravo</i></h3>
```

This is the basic pattern for most structured writing algorithms. An algorithm consists of a set of rules.

- For each structure, you create a rule that says how to transform that structure into the structure you want.
- Each rule specifies the new structures to create and where to place the content and any nested structures.
- In each rule, you specify where the processor should process any nested structures and apply any rules that apply to them.
- If you want a different rule for a structure occurring in different contexts, write a separate rule for each context.

Why is it important for you to understand this? Because when you are going through the process of abstracting out invariants to move content to the document domain or subject domain, it is really useful to understand how those invariants will be factored back in. In fact, understanding how this works can help you recognize invariants in your source and

give you the confidence to factor them out. Writing down the pseudocode for processing the structures you are creating can help you validate that you have factored things out correctly and that the structures you are creating will be easy to process and that the processing rules will be clear, consistent, and reliable.

Obviously there is more involved in a complete processing system, and we are not going to get into the gritty details here, but let's look at few a cases that come up frequently.

Processing container structures

When we move content to the document domain or the subject domain, we often create container structures to provide context. These container structures don't have any analog in the media domain, so what do we do with them when it is time to publish? We obviously use them to provide context for the rest of our processing rules, but what to we do with the containers themselves?

In the previous example the content was contained in a `section` structure. So how does the `section` structure get processed?

```
match section
    continue
```

Yes, it's that simple. Just don't output any new structure in its place. The section container has done its work at this point so we simple discard it. We still want the stuff inside it though, so we use the `continue` instruction to make sure the contents get processed. In short, the container is a box. We unpack the contents and discard the box.

Restoring factored-out text

Sometimes when we factor out the invariants in content, we are not only factoring out styles, we are also factoring out text. To process the content we need to put the text back (obviously we can put back different text depending on our needs – which was why we factored it out in the first place).

As we saw, a simple example of factoring out text is numbered and bulleted lists, where we factor out the text of the numbers and bullets. Let's look at how we create rules to put them back.

Suppose we have a document that contain these two different kinds of lists:

```
paragraph: To wash hair:
ordered-list:
    list-item:Lather
```

```

list-item:Rinse
list-item:Repeat

paragraph: The box contains:
unordered-list:
    list-item:Sand
    list-item:Eggs
    list-item:Gold

```

Let's write a set of rules to deal with this document. Converting this to HTML lists won't tell us much, since HTML handles list numbering and bullets itself, so we'll create instructions for printing on paper. We won't use real printing instructions (they get tediously detailed). Instead we will use the same style specification shorthand we used above. The `paragraph` rule is simple enough:

```

match paragraph
    apply style {font: 10pt "Century Schoolbook" }
    continue

```

Now let's deal with the `ordered-list`. The ordered list structure is just a container, so we don't need to create an output structure for it. But because this is an ordered list, we need to start a count to number the items in the list. That means we need a variable to store the current count. We will use a `$` prefix to indicate that we are creating a variable:

```

match ordered-list
    $count=1
    continue

```

Then the rule for each ordered list item will output the value of the variable and increment it by one:

```

match ordered-list/list-item
    apply style {font: 12pt "Century Schoolbook" }
    output $count
    output ".[tab]"
    $count=$count+1
    continue

```

Every time the `ordered-list/list-item` rule is fired, the count will increase by one, resulting in the list items being numbered sequentially.

If a new numbered list is encountered, the `ordered-list` rule will be fired, resetting the count to 1.

This rule will not match `list-item` elements that are children of an unordered-list element, so we need a separate set of rules for unordered lists:

```
match unordered-list
    continue

match ordered-list/list-item
    apply style {font: 12pt "Century Schoolbook"}
    output "[em dash][tab]"
    continue
```

Applying the combined set of rules will produce output like this:

```
{font: 10pt "Century Schoolbook"}To wash hair:
{font: 10pt "Century Schoolbook"}1.[tab]Lather
{font: 10pt "Century Schoolbook"}2.[tab]Rinse
{font: 10pt "Century Schoolbook"}3.[tab]Repeat
{font: 10pt "Century Schoolbook"}The box contains:
{font: 10pt "Century Schoolbook"} [em dash][tab]Sand
{font: 10pt "Century Schoolbook"} [em dash][tab]Eggs
{font: 10pt "Century Schoolbook"} [em dash][tab]Gold
```

Note how the structure has been flattened and all of the abstractions of document structure have been removed. We are back in the media domain, with a flat structure that specifies formatting and text.

Processing in multiple steps

We do not always want to apply final formatting to our content in a single step. When we separated content from formatting, we did the separation in several stages. It is often desirable to put them back together in several stages. Not only are the algorithms involved easier to write and maintain if they only do one step of the process, we can often reuse some of the downstream steps (nearer the media domain) for many different types of document domain and subject domain content.

So far we have looked at examples from the media domain and the document domain. Let's look at one from the subject domain. We used an example of completing the separation of content from formatting by moving a labeled list from the document domain to the subject domain.

```
address:
  street: 123 Elm Street
```

```
town: Smallville
country: USA
code: 12345
```

Now let's look at the algorithm (the set of rules) for getting it back to the document domain, where it should look like this:

```
labeled-list:
    list-item:
        label: Street
        contents: 123 Elm Street
    list-item:
        label: Town
        contents: Smallville
    list-item:
        label: Country
        contents: 123 USA
    list-item:
        label: Code
        contents: 12345
```

Here is the set of rules to accomplish this transformation:

```
match address
    create labeled-list
    continue

match street
    create list-item
    create label
    output "Street"
    create contents
    continue

match town
    create list-item
    create label
    output "Town"
    create contents
    continue

match country
    create list-item
    create label
```

```
        output "Country"
create contents
continue

match code
    create list-item
    create label
        output "Code"
create contents
continue
```

Notice that the text of the labels, which we factored out when we moved to the subject domain, are being factored back in here, and are specified in the processing rules. As we moved the content from the media domain to the document domain to the subject domain, we first factored out invariant formatting and then invariant text. In the algorithms, we put back the text and the formatting, each at a different processing stage.

Processing content in multiple steps can save us a lot of time. The subject domain address structure is specific to a single subject and we might have many similar structures in our subject domain markup. But it is presented as a `labeled-list` structure. A labeled list is a document domain structure that can be used to present all kinds of information, and that can be formatted for many different media. By transforming the address structure into a `labeled-list` structure, we avoid having to write any code to format the address structure directly. We can format the address correctly for multiple media using the existing `labeled-list` formatting rules.

Query-based processing

The rule-based approach shown here is not the only way to process structured writing. There is another approach which we could call the query-based approach.¹ In this approach, you write a query expression that reaches into the structure of a document and pulls out a structure or a set of structures from the middle of the document.

This is a useful technique if you want to radically rearrange the content of a document or if you want to pull content out of one document to use in another. Any reference in these pages to using an algorithm to go through a content set and pull out certain pieces of data is an example of the query based approach. We will see more examples of this in future chapters.

This chapter does not by any means describe the full range of available content processing techniques or all the ways in which algorithms can recognize and manipulate structures in content. The point of introducing you to structured writing algorithms and the basics

¹The rule-based and query-based approaches are often called “push” and “pull” methods respectively, but I sometimes find it hard to remember which is which. I find rule-based and query-based more descriptive.

of how content processing works is to enable you to think about structures with an eye to how they can be processed. Often when you look at structures in this way you can see how a simple structure could be used and how that simpler structure can be processed to achieve the same result as a more complex structured. Don't be afraid to write out your algorithms in pseudocode to make sure you have a clear idea of the processing you intend for your structures, and don't feel confined by the algorithms shown in this book, or the annotation I have used for pseudocode examples. You own your pseudocode. Choose a format that is clear to you and that communicates clearly to whoever will be implementing the actual code. Working together with that person can help you improve your appreciation of algorithms and your ability to write useful pseudocode. Can learning to write real code help with this to? Absolutely it can. Is it necessary? No. What is essential is to think in terms of distributing certain parts of your content system to algorithms and the ability to describe those algorithms well enough that you have a clear idea of what you are asking them to do, and a reasonable certainty that you are passing them the structures and information they need to do it.

Chapter 11. Single Sourcing

One of the sources of complexity in content creation and delivery is that you sometimes want to publish the same document in more than one media. These days that usually means on paper and on the Web. Publishing the same document to more than one media is generally called single sourcing.¹

Single sourcing adds complexity to the notion of separating content from formatting because we are now separating the content from not one but two or more sets of formatting. If you do not get the partitioning of content and formatting correct, it will show up when you try to apply two different sets of formatting to the same content.

This is an area in which many organizations have unhandled complexity which is falling through to their readers. Their authoring techniques do not allow them to sufficiently separate the content from all of the formatting that is appropriate in different media, resulting in their delivering one media's formatting to a different media. This means the content does not work as well in the media it was not designed for. By improving how we separate content from formatting, though, we can significantly reduce the amount of complexity that goes unhandled, resulting in a better experience for our readers.

Basic single sourcing

The basic single sourcing algorithm is straightforward and we have covered most of it already in Chapter 10, *Processing Structured Text*.

Basic single sourcing involves taking a piece of content in the document domain and processing its document domain structures into different media domain structures for each of the target media.

Suppose we have a recipe recorded in the document domain:

```
page:  
    title: Hard Boiled Eggs  
  
    A hard boiled egg is simple and nutritious.  
    Prep time, 15 minutes. Serves 6.  
  
    section: Ingredients  
        ul:
```

¹The term “single sourcing” gets used to mean different things, all of which involve a single source in one way or another, but which use different approaches and achieve different ends. I will look at a number of variation on this theme in subsequent chapters.

```
        li: 12 eggs
        li: 2qt water

section: Preparation
ol:
    li: Place eggs in pan and cover with water.
    li: Bring water to a boil.
    li: Remove from heat and cover for 12 minutes.
    li: Place eggs in cold water to stop cooking.
    li: Peel and serve.
```

We can output this recipe to two different media by applying two different formatting algorithms. First we output to the Web by creating HTML:

```
match page
    create html
        stylesheet www.example.com/style.css
    continue

match title
    create h1
    continue

match p
    copy
    continue

match section
    continue

match section/title
    create h2
    continue

match ul
    copy
    continue

match ol
    copy
    continue

match li
    copy
```

continue

In the code above, paragraph and list structures have the same names in the source format as they do in the output format (HTML) so we just copy the structures rather than recreating them. This is a common pattern in structured writing algorithms.

The above algorithm transforms our source into HTML that looks like the following:

```
<html>
  <head>
    <link rel="stylesheet" type="text/css"
          href="//www.apache.org/css/code.css">
  </head>

  <h1>Hard Boiled Eggs</h1>

  <p>A hard boiled egg is simple and nutritious.  
Prep time, 15 minutes. Serves 6.</p>

  <h2>Ingredients</h2>

  <ul>
    <li>12 eggs</li>
    <li>2qt water</li>
  </ul>

  <h2>Preparation</h2>

  <ol>
    <li>Place eggs in pan and cover with water.</li>
    <li>Bring water to a boil.</li>
    <li>Remove from heat and cover for 12 minutes.</li>
    <li>Place eggs in cold water to stop cooking.</li>
    <li>Peel and serve.</li>
  </ol>
</html>
```

Outputting to paper (or to PDF, which is a kind of virtual paper) is more complex. On the Web, you output to a screen which is of flexible width and infinite length. The browser generally takes care of wrapping lines of text to the screen size (unless formatting commands tell it to do otherwise) and there is no issue with breaking text from one page to another. For paper, though, you have to format for a fixed size page. This means that formatting for paper involves fitting the content into a set of fixed size pages.

This leads to a number of formatting problems, such as:

- Where do break each line of text so the margins look neat and the text is not cramped or spread out too much.
- How to avoid a heading appearing at the bottom of a page or the last line of a paragraph appearing as the first line of a page?
- How do you handle a list or a table when you run out of space on the page for it?
- How do you create a cross reference to the page on which a chapter starts or a figure appears when you don't know the page number yet?

Because of issues like this, you don't write a formatting algorithm for paper directly, the way you would write an algorithm to output HTML. Rather, you use an intermediate typesetting system which already knows how to handle things like inserting page number references and handling line and page breaks. Rather than handling these things yourself, you tell the typesetting system how you would like it to handle them and then let it do its job.

One such typesetting system is XSL-FO (Extensible Stylesheet Language - Formatting Objects). Other typesetting systems you can use for print output include TeX and later versions of CSS. XSL-FO is a typesetting language written in XML. To format your content using XSL-FO, you transform your source content into XSL-FO markup, just the way you transform it into HTML for the Web. But then you run the XSL-FO markup through an XSL-FO processor to produce your final output, such as PDF.

Here is a small example of XSL-FO markup:

```
<fo:block space-after="4pt">
    <fo:wrapper font-size="14pt" font-weight="bold">
        Hard Boiled Eggs
    </fo:wrapper>
</fo:block>
```

As you can see, the XSL-FO code contains a lot of specific media domain instructions for spacing and font choices. The division between HTML for basic structures and CSS for specific formatting does not exist here. Also note that as a pure media-domain language, XSL-FO does not have document domain structures like paragraphs and titles. From its point of view, a document consists simply of a set of blocks with specific formatting properties attached to them.

Because of all this detail, I am going to show the literal XSL-FO markup in the pseudocode of the algorithm, and I am not going to show the algorithm for the entire recipe. (The point is not for you to learn XSL-FO here, but to understand how the single-sourcing algorithm works.)

```
match title
    output '<fo:block space-after="4pt">'
        output '<fo:wrapper font-size="14pt" font-weight="bold">'
            continue
        output '</fo:wrapper>'
    output '</fo:block>'
```

Here, as you can see, the rule simply wraps the XSL-FO formatting structures around the text of the title. It unpacks the text from a document domain structure and inserts it into a subject domain structure.

Differential single sourcing

Basic single sourcing outputs the same document presentation to different media. But each media is different, and what works well in one media does not always work as well in another. For example, online media generally support hypertext links, while paper does not. Let's suppose that we have a piece of content that includes a link.

In Rio Bravo, {the Duke}(link "http://JohnWayne.com") plays an ex-Union colonel out for revenge.

In SAM syntax, the piece of markup represented by "http://JohnWayne.com" specifies the address to link to. In the algorithm examples below, this markup is referred to as the “specifically” attribute using the notation @specifically.

In HTML we want this output as a link using the HTML a element, so we write the algorithm like this:

```
match p
    copy
        continue

match link
    create a
        attribute href = @specifically
        continue
```

The result of this algorithm is:

```
<p>In Rio Bravo, <a href="http://JohnWayne.com">The Duke</a>
plays and ex-Union colonel out for revenge.
```

</p>

But suppose we want to output this same content to paper. If we output it to PDF, we could still create a link just like we do in HTML, but if that PDF is printed, all that will be left of the link will be a slight color change in the text and maybe an underline. It will not be possible for the reader to follow the link or see where it leads.

Paper can't have active links but it can print the value of a URL so that reader can type it into a browser if they want to. An algorithm could do this by printing the link inline or as a footnote. Here is the algorithm for doing it inline. (We'll dispense with the complexity of XSL-FO syntax this time.)

```
match p
    create fo:block
        continue

match link
    continue
    output " (see: "
    output @specifically
    output ") "
```

This will produce:

```
<fo:block>
In Rio Bravo, the Duke (see: http://JohnWayne.com)
plays an ex-Union colonel out for revenge.
</fo:block>
```

This works, but we should note that the effect is not exactly the same in each media. Online, the link to JohnWayne.com serves to disambiguate the phrase “The Duke” for those readers who do not recognize it. A simple click on the link will explain who “the Duke” is. But in the paper example, such disambiguation exists only incidentally, because the words “JohnWayne” happen to appear in the URL. This is not how we would disambiguate “The Duke” if we were writing for paper. We would be more likely to do something like this:

The Duke (John Wayne) plays an ex-Union colonel.

This provides the reader with less information, in the sense that it does not give them access to all the information on JohnWayne.com, but it does the disambiguation better and in a more paper-like way. The loss of the reference to JohnWayne.com is probably not an issue here. Following that link by typing it into a browser is a lot more work than simply clicking on it on a Web page. If someone reading on paper wants more information

on John Wayne they are far more likely to type “John Wayne” into Google than type “JohnWayne.com” into the address bar of their browser.

With the source content written as it is, though, there is no easy way to produce this preferred form for paper. The content is in the document domain and the document domain specifies the presentation of the content. It is therefore not a neutral format between different media that require different forms of presentation. The choice to specify a link gives it a strong bias towards the Web and online media rather than paper. A document domain approach that favored paper would similarly lead to a poorer online presentation that omitted the link.

What we need to address the problem is a differential approach to single sourcing, one that allows us to differ not only the formatting but the presentation of the content for different media.

One way to accomplish this differential single sourcing is to record the content in the subject domain, thus removing the prejudice of the document domain presentation for one class of media or another. Here is how this might look:

{The Duke} (actor "John Wayne") plays an ex-Union colonel.

In this example, the phrase “The Duke” is annotated with a subject domain annotation that clarifies exactly what the text refers to. That annotation says that “the Duke” is the name of an actor, specifically “John Wayne”.

Our document domain examples attempted to clarify “the Duke” for readers, but did so in media-dependent ways. This subject domain example clarifies the meaning of “The Duke” in a presentation-neutral way by explicitly marking up what the phrase refers to in a formal and unambiguous way. This disambiguation is not written for readers, however. It is written for algorithms. This allows an algorithm to generate a media-appropriate clarification for the reader in the target media.

For paper:

```
match actor
    continue
    output " ( "
    output @specifically
    output ") "
```

For the Web:

```
match actor
  create link
    $href = get link for actor named @specifically
    attribute href = $href
  continue
```

This supposes the existence of a system that can respond to the `get link` instruction and look up pages to link to based on the type and a name of a subject. We will look at how a system like that works in Chapter 15, *Linking*, which will revisit this example in more detail.

Differential organization and presentation

Differences in presentation between media can be broader than this. Paper documents sometimes use complex tables and elaborate page layouts that often don't translate well to online media. Effective table layout depends on knowing the width of the page you have available, and online you don't know that. A table that looks great on paper may be unreadable on a mobile device, for instance.

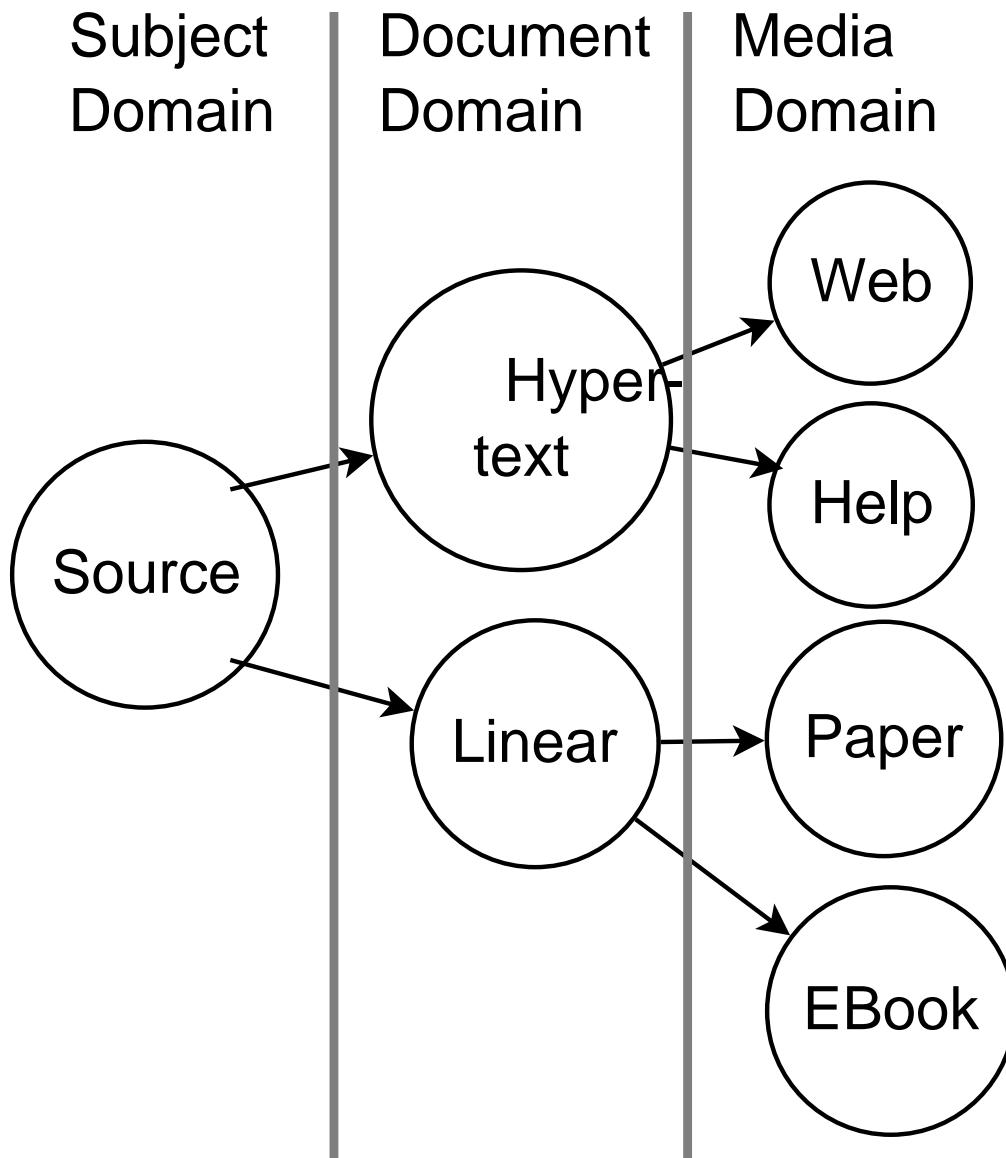
And this is more than a layout issue. Sometimes the things that paper does in a static way should be done in a dynamic way in online media. For example, airline or train schedules have traditionally been printed as timetables on paper, but you will virtually never see them presented that way online. Rather, there will be an interactive travel planner that lets you choose your starting point, destination, and desired travel times and then presents you with the best schedule, including when and where to make connections.

Single sourcing your timetable to print and PDF will not produce the kind of online presentation of your schedule that people expect, and that can have a direct impact on your business.

To single source schedule information to paper and online, you can't maintain that content in a document domain table structure. You need to maintain it in a timetable database structure (which is subject domain, but really looks like a database, not a document at all).

An algorithm can then read the database to generate a document domain table for print publication. For the Web, however, you will create a web application that queries the database dynamically to calculate routes and schedules for individual travelers.

In this scenario, you are moving content to the subject domain to partition the presentation from the information so that you can produce a different information design in different media.



We will examine these scenarios in more depth in Chapter 33, *Wide Structures*.

Conditional differential design

You can also do differential single sourcing by using conditional (management domain) structures in the document domain.

For instance, if you are writing a manual that you intend to single source to a help system, you might want to add context setting information to the start of a section when it appears in the help system. The manual may be designed to be read sequentially, meaning that the context of individual sections is established by what came before. But help systems are always accessed randomly, meaning that the context of a particular help topic may

not be clear if it was single sourced from a manual. To accommodate this, you could include a context setting paragraph that is conditionalized to appear only in help output:

```
section: Wrangling left-handed widgets
```

```
~~~(?help-only)
```

```
    Left-handed widgets are used when wrangling  
    counter-clockwise.
```

To wrangle a left handed widget:

1. Loosen the doohickey using a medium thingamabob.
2. Spin three times around under a full moon.
3. Touch the sky.

In the SAM markup above, the `~~~` creates a “fragment” structure to which conditional tokens can be applied. Content indented under the fragment marker is part of the fragment.

To output a manual, we suppress the help-only content:

```
match fragment where conditions = help-only  
    ignore
```

To output help, we include it:

```
match fragment where conditions = help-only  
    continue
```

Primary and secondary media

While there is a lot you can do in the way of differential single sourcing to successfully output documents that work well in multiple media, there are limits to how far this approach can take you.

In the end, linear and hypertext approaches fundamentally different ways of writing which invite fundamentally different ways of navigating and using information. Even moving content to the subject domain as much as possible will not entirely factor out these fundamental differences of approach.

When single sourcing content to both linear paper-like media and hypertext web-like media, you will generally have to choose a primary media to write for. Single sourcing

that content to the other media will be on a best-effort basis. It may be good enough for a particular purpose, but it will never be quite as good as it could have been had you designed for that media.

Many of the tools used for single sourcing have a built in bias towards one media or another. Desktop-publishing tools like FrameMaker, for instance, were designed for linear media. Online collaborative tools like wikis were designed for hypertext media. It is usually a good idea to pick a tool that was designed for the media you choose as your primary. Tools partition the complexity of content creation in a way that suits the primary media they are designed for.

In many cases, the choice of primary media is made implicitly based on the tools a group has traditionally been using. This usually means that the primary media is paper, and it often continues to be so even after the organization had stopped producing paper and their readers are primarily using online formats.

Some organizations seem to feel that they should only switch to tools that are designed primarily for online content when they have entirely abandoned the production of paper and paper-like formats such as PDF. This certainly does not need to be the case. It is perfectly possible to switch to an online-primary tools and still produce linear media as a secondary output format.

Changing your information design practices from linear paper based designs to hypertext designs is non-trivial, but such designs better suit the way many people use and access content today. I discuss these design differences in depth in my book, *Every Page is Page One: Topic-based Writing for Technical Communication and the Web*. Don't expect single sourcing to successfully turn document-oriented design into effective hypertext by themselves. To best serve modern readers it will usually be much more effective to adopt an Every Page is Page One approach to information design and use structured writing techniques to do a best-effort single sourcing to linear media for those of your readers who still need paper or paper-like formats – or for those documents where linear organization still makes sense.

Responsive Design

Responsive design, an approach to web design in which a page automatically adjusts to the viewport it is displayed on, is a form of differential single sourcing. Many attempts at responsive design are not as responsive as we would like them to be because the content is not stored in a source format that allows a sufficient degree of differential presentation. If you want to achieve effective and consistent responsive design in your output, you will find that the use of subject domain structures in your content will be enormously helpful.

It is very difficult to expect writers to encode the presentation structures of a responsive design into their files as they write. Not only that, the very idea of responsive design should take into account the possibility of having to respond to new media in the future.

Every sound responsive design decision is based on the appropriate organization of subject matter in a particular view port. Therefore, the rules of effective responsive design are going to be expressed in subject matter terms. The best way to create content that will be presented using responsive design, therefore, is in the subject domain where the subject matter divisions on which responsive design rules are expressed are made fully explicit in the content. We will look at this in more depth in Chapter 28, *Active content*.

Chapter 12. Reuse

Another source of complexity in content creation occurs when you want the same information to occur in more than one publication. If there is no coordination between authors, each author who want to include the piece of information will research and write it for themselves. Individual writers who want to use the same information in more than one publication they are writing, will copy the information from one publication to another. However it happens, you now have two or more instances of the same information that have to be maintained and edited whenever the subject matter changes. If some instances do not get updated, or some of them get updated incorrectly, that is content maintenance complexity that is not being handled, and, as always, it falls through to the reader in the form of inconsistent or incorrect information.

Content reuse is an attempt to handle the complexity associated with using the same information in more than one place. Reuse has become one of the main drivers of structured writing, particularly in the form of widespread adoption of DITA. Unfortunately, most reuse techniques also introduce a lot of management domain complexity. A single-minded focus on reuse has sometimes led to the implementation of systems that dump large amounts of complexity on authors (and thus, eventually, on readers). It is an area in which, if you are not careful, you can easily end up with more unhandled complexity in the system than when you began.

There are a number of different scenarios in which you might want to duplicate information in more than one place. For example:

- You are writing about different products in a product line that share common technology. You want to duplicate information on the common features in a document on each individual product.
- You are writing about several different releases of a product. You want information on features that have not changed to be duplicated in all documents.
- You are writing documents for different audiences (marketing material, technical documentation for various roles, training material) and you want to duplicate product descriptions in all these documents.
- There is general information about your organization that you want to duplicate in many different publications (such as your copyright and trademark ownership statements).

“Reusing” can suggest that this activity is somewhat akin to rummaging through that jar of old nuts and bolts you have in the garage looking for one that is the right size to fix your lawnmower. While you can do it that way, that approach is neither efficient nor reliable. The efficient and reliable approach involves deliberately creating content for use in multiple locations. This means that you need to place constraints on the content to

be reused and the content that reuses it, and that means you are in the realm of structured writing.

Fitting pieces of content together

If you are going to create one piece of content that can be used in many outputs, you have to make sure it fits in each of those outputs. In other words, partition it appropriately for reuse.

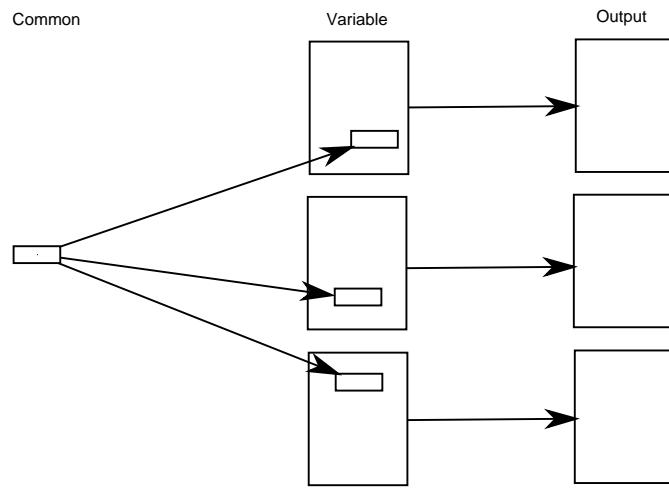
If you cut and paste, this is not a concern. You can cut any text you like, paste it in anywhere, and edit it to fit if you need to. But if the content you want to use is used in other places, you can't edit it to fit because that might cause it to no longer fit in the other places. For reuse to work, the content must be written to fit in multiple places. In other words, it has to meet a set of constraints that will allow it to fit in multiple places. We will look at this in more detail in Chapter 13, *Composition*. In this chapter we will focus on the algorithms for fitting the pieces together.

There are seven basic models for fitting pieces of content together:

- Common into variable
- Variable into common
- Variable into variable
- Common with conditions
- Factor out the common
- Factor out the variable
- Assemble from pieces

Common into variable

In the common into variable case, you have a common piece of content that occurs in many places. This could mean it occurs in many documents or in many places in the same document, or both.



Each individual procedure is the variable part and the standard warning is the common part. We looked at an example of this in Chapter 6, *The Management Domain: an Intrusion*:

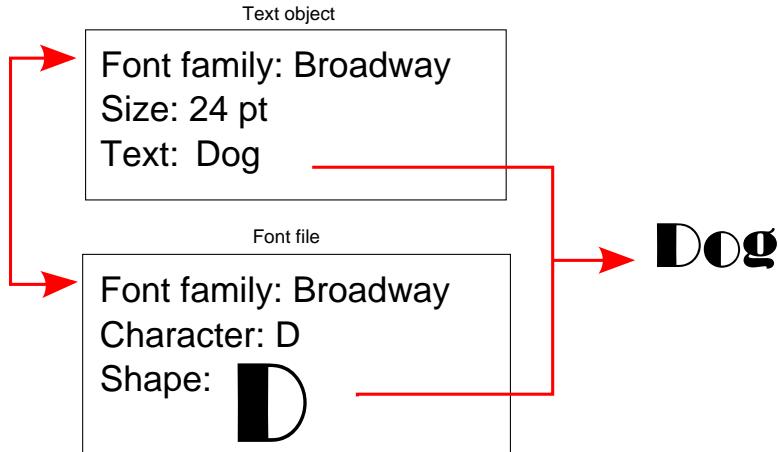
```
procedure: Blow stuff up
    >>>(files/shared/admonitions/danger)
    step: Plant dynamite.
    step: Insert detonator.
    step: Run away.
    step: Press the big red button.
```

To ensure that the included content will always fit, you need to make sure that there is a clear partitioning of responsibilities between the common content and each of the procedures it will be inserted into. The inserted content should give the safety warning, the whole safety warning, and nothing but the safety warning. The procedure structure should describe the steps and only the steps and should insert the reusable warning at the right place.

Of course, you can also use the subject-domain approach to common into variable that we looked at in Chapter 6, *The Management Domain: an Intrusion*. I will go into more detail on that approach later in this chapter.

Variable into common

In the variable into common case, you have a single document that will be output in many different ways by inserting variable content at certain locations.



For instance, if you are writing a manual to cover a number of car models you can factor out the number of seats each model has.

The vehicle seats >(\$seats) people.

This is the fixed content that will occur in all manuals, with the number of seats pulled in from an external source. Let's say we have a collection of vehicle data that is stored in a structure like this:

```
vehicles:  
    vehicle: compact  
        seats: four  
        colors: red, green, blue, white, black  
        transmissions: manual, CVT  
        doors: four  
        horsepower: 120  
        torque: 110 @ 3500 RPM  
    vehicle: midsize  
        seats: five  
        colors: red, green, blue, white, black  
        transmissions: CVT  
        doors: four  
        horsepower: 180  
        torque: 160 @ 3500 RPM
```

Then we write the algorithm to process the insert so that it queries this structure.

```
match insert where variable = $seats
    $number_of_seats = vehicles/vehicle[$model]/seats
    output $number_of_seats
```

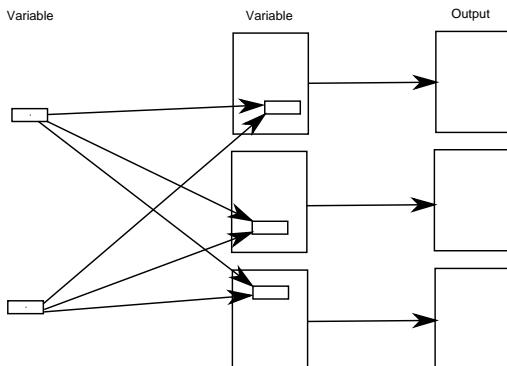
All these insert and query mechanisms are pseudocode, of course. Exactly how things work and exactly how you delineate, identify, and insert content vary from system to system.

With the variable into common technique, you are creating a common source by factoring out all the parts of the different outputs that are not common. This is, in some ways, the inverse of the usual pattern of factoring out invariants: we are actually factoring out the variants. But really, it amounts to the same thing. We are factoring variants from invariants. The only real difference between this and the common into variable is whether the common parts are embedded in the variable parts or vice versa. Either way, we still end up with two artifacts: the variable piece or pieces and the common piece or pieces.

Variable into variable

Variable into variable is a variation on common into variable in which you can make a wholesale change of the common elements that you are pulling into a set of variable documents.

For example, suppose you decide to market your product line to a new market. The new market has different safety regulation which means you need to insert a different standard warning into all your manuals. In this case, you want to swap out the common elements used in your home market and substitute the common elements for the foreign market.



Here we need to talk about how we identify the content to be inserted. In the common into variable example, we inserted the content of a file that contains a standard warning.

But for variable into variable this approach simply does not work. Variable into variable requires loading a different file, which is difficult when the content specifies a particular file name to import.

As always in structured writing, we look for a way to factor out the problematic content. So here we look for a way to factor out the file name and replace it with something else.

Using IDs

The most basic way to factor out the file name is to give the content of the file an ID. An ID is a management domain structure used to identify a piece of content in a location independent way. No matter where the content is stored, it keeps the same ID. Here is the warning file with the ID `#warn_danger` added:

```
warning: (#warn_danger)
    title: Danger
```

Be very very careful. This could kill you.

We can then insert the warning into our procedure by referring to that ID.

```
procedure: Blow stuff up
    >>>(#warn_danger)
    step: Plant dynamite.
    step: Insert detonator.
    step: Run away.
    step: Press the big red button.
```

The responsibility for locating the warning has now been shifted from the content to the algorithm.

```
match insert with ID
    $insert_content = find ID in $content_set
    output $insert_content
```

This is a constant pattern in structured writing. When it comes to locating resources, you want to move that responsibility from the content to the algorithm. This makes it easier to update the locations, but it also gives you far more options for storing and managing your content, since algorithms can interact with a variety of systems in sophisticated ways, rather than just storing a static address. It also means you can make wholesale changes in how your content is stored without having to edit the content itself. This is a major win in terms of partitioning complexity so it can be distributed and handled more efficiently.

This means that the synthesis algorithm (which I will talk about in Chapter 27, *Publishing*) needs some way to resolve the ID and find the content to include. In many cases, a content management system is used to resolve the ID. In other cases it is as simple as the algorithm searching through a set of files to find the ID or building a catalog that points to the files that contain IDs.

To do variable into variable reuse in a system that uses IDs, you simply point the algorithm at a different set of files that contain the same IDs, but attached to different content. So if your foreign market requires a different warning, you can create a file like this:

```
warning: (#warn_danger)
title: Look out!
```

Pay close attention. You could really hurt yourself.

By telling the build of the foreign market docs to search this file for IDs rather than the file with the domestic market warning, you automatically get the the foreign warning rather than the domestic one.

Using keys

Another way to do this is with another management domain structure called a key. A key is like an ID that is not directly tied to a resource. Instead, it is an abstract identifier for an abstract resource that might be instantiated by many different resources in different circumstances. Since a key does not represent any one concrete resource, we don't assign the key to a resource. Instead we use an intermediate lookup table to resolve keys to particular resources for a particular purpose.

So in this case we have the warning in a file called files/shared/admonitions/domestic/danger with the following content (no ID):

```
warning:
title: Danger
```

Be very very careful. This could kill you.

And we have the procedure which includes the warning via a key:

```
procedure: Blow stuff up
>>>(%warn_danger)
step: Plant dynamite.
step: Insert detonator.
```

```
step: Run away.  
step: Press the big red button.
```

(I am using # to denote IDs and % to denote keys. This is the notation that SAM uses for IDs and keys, but it is purely arbitrary and has nothing to do with how they work. Different systems will denote IDs and keys in different ways.)

To connect the key to the warning file, we then create a key lookup table:

```
keys:  
key:  
    name: warn_danger  
    resource: files/shared/admonitions/domestic/danger
```

When the synthesis algorithm processes the procedure, it sees the key reference %warn_danger and looks it up in the key lookup table. The key lookup table tells the algorithm that the key resolves to the resource files/shared/admonitions/domestic/danger. The algorithm then loads that file and inserts the contents into the output.

```
match insert with key  
    $resource = find key in lookup-table  
    output $resource
```

To output your content for the foreign market, you simply prepare a new key lookup table:

```
keys:  
key:  
    name: warn_danger  
    resource: files/shared/admonitions/foreign/danger
```

You then tell the synthesis algorithm to use this lookup table instead.

Using keys is not necessarily better than using IDs. What it comes down to is that you need some kind of bridge between the citation of an identifier in the source file and the location of a resource with that identifier in the content store. This bridge can be created by a key lookup table, by remapping file URLs, or by modifying a query to a content repository.

One feature of the key approach is that, because it does not attach the key directly to the content, it can be used to identify resources that do not have IDs, which may include resources that you do not control.

One downside of keys is that by themselves they can only point to a whole resource. This could force you to keep all your reusable units in separate files. To avoid this, you can combine keys with IDs. The following example combines the foreign and domestic danger warnings into one file and gives each an ID:

```
warnings:
```

```
  warning: (#warn_danger_domestic)
    title: Danger
```

Be very very careful. This could kill you.

```
  warning: (#warn_danger_foreign)
    title: Look out!
```

Pay close attention. You could really hurt yourself.

Now we can rewrite our key lookup tables to use the IDs to pull the right warning out of this common file. For the domestic build we would use a key lookup table like this:

```
keys:
```

```
  key:
```

```
    name: warn_danger
```

```
    resource: files/shared/warnings#warn_danger_domestic
```

And for the foreign build, one that looks like this:

```
keys:
```

```
  key:
```

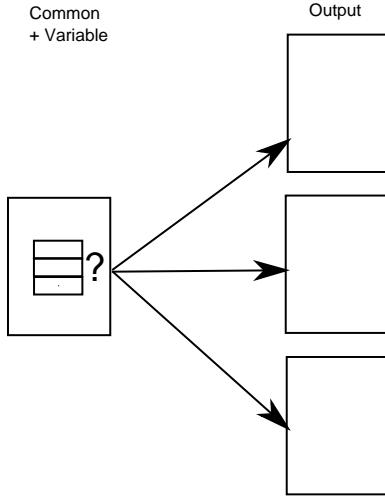
```
    name: warn_danger
```

```
    resource: files/shared/warnings#warn_danger_foreign
```

Here we have partitioned the warnings into a separate file and also partitioned the location of those files from the author. The use of keys as a bridge between two partitions can be a convenient way to manage content relationships without having to update source files every time the relationships change. The downside is that keys introduce an abstract element into the author's world, and abstractions are a form of complexity that can be difficult to deal with.

Common with conditions

In some cases of variable into common, the variant pieces is not actually factored out into a separate file. Rather, each of the possible alternatives is included in the file conditionally.



For instance, in content for a car manual you might have conditional text for the number of people the car seats.

The vehicle seats {four} (?compact) {five} (?midsize) {seven} (?van).

Here the main text is the fixed piece and the variable pieces are the words “four”, “five”, and “seven”. Which of these will be included in the output depends on which condition is applied during the build. If the condition `midsize` is applied, then the output text will be “five” and the other alternatives will be suppressed.

```
match phrase with condition
    if condition in $build_conditions
        continue
    else
        ignore
```

The upside of the conditional approach is that it keeps all the variants in one file, so your algorithm does not have to know where to go to find the external content.

But there are a number of downsides to this approach:

- It gets very cumbersome to read the source if there are many different conditions applied.
- When the subject matter changes, you have to find all the places the conditions occur and update them.
- If the same data point (the number of seats) is mentioned in many different documents, that information is still being duplicated all over the content, which makes it hard to

maintain and verify, and hard to change if, for example, the compact seats five in the next model year.

Common with conditions is not limited to cases where there are alternate values, however. In some cases, content may simply be inserted or omitted for certain outputs.

The main features of the car are:

```
ol:  
    li: Wheels  
    li: Steering wheel  
    li:(?deluxe) Leather seats  
    li: Mud flaps
```

In this case, the list item “Leather seats” would only be published if the condition `deluxe` was specified in the build. It would be omitted for all other builds. In these kinds of cases, it is harder to get away from the use of conditionals as a reuse mechanism.

This approach to reuse is often called filtering or profiling. Some systems have elaborate ways of specifying filtering or profiling of the content. The net effect is the same as the simple condition tokens shown here, but they may allow for more sophisticated or elaborate conditions than shown here.

Because common with conditions is essentially a form of variable into common where the variable content is contained inside the common source, it can technically be replaced by a variable into common approach in all cases. In practice, the use of conditions tends to occur when:

- The number of variations is small and thought to be fixed or to change infrequently.
- The variable pieces are eccentric or contextually dependent.
- The writer or organization wishes to avoid managing multiple files.
- The current tools don’t support variable into common.

How successful a common with conditions approach will be also depends on what you choose for your conditional expressions. Generally, subject domain conditions will be much more stable and manageable than document domain conditions. For instance, conditions that relate to different vehicles (subject domain) are based in the real world and are therefore objectively true as long as the subject matter remains the same. Conditions that relate to different publications or different media, on the other hand, are not objectively true and can’t be verified independently. The only way to verify them is to build the different documents or media and see if you got the content you expected.

This makes maintaining such conditions cumbersome and error prone – an indication that complexity is not being distributed in an optimal way.

Factor out the common

In Chapter 6, *The Management Domain: an Intrusion*, we noted that the subject domain alternative to using an insertion instruction for the warning text was to specify which procedures were dangerous, thus factoring out the constraint that the warning must appear.

```
procedure: Blow stuff up
    is-it-dangerous: yes
    step: Plant dynamite.
    step: Insert detonator.
    step: Run away.
    step: Press the big red button.
```

In this case, the author does not have to identify the material to be included, either directly by file name or indirectly through an ID or a key. Instead, it is up to the algorithm to include it:

```
match procedure/is-it-dangerous
    if is-it-dangerous = 'yes'
        output files/shared/warnings#warn_danger_domestic
```

To produce the foreign market version of the documentation, you simply edit the rule:

```
match procedure/is-it-dangerous
    if is-it-dangerous = 'yes'
        output files/shared/warnings#warn_danger_foreign
```

Or, to further the partitioning of complexity in the code, you can use keys:

```
match procedure/is-it-dangerous
    if is-it-dangerous = 'yes'
        $resource = find key '%warn_danger' in lookup-table
        output $resource
```

The beauty of this approach is that the content is entirely neutral as to what kind of reuse may be going on or how dangerous procedures may be treated. Because the content

contains only objective information about the procedure itself, you can implement any algorithm you like to publish or reuse the content in any way you like at any time based on this information. By making the content not specific to any form of reuse or any reuse mechanism, we effectively make it much more reusable and have partitioned the complexity of reuse much more neatly and reliably.

We are also making the content much easier to write, since this approach does not require the writer to know how the reuse mechanism works, how to identify reusable content, or even that reuse is occurring at all. All they have to do is answer a simple question about the content – is the procedure dangerous or not -- to which they should already know the answer. In other words, this approach partitions the entire reuse mechanism from the author. This is a really big win because the biggest problem with most reuse techniques is the amount of complexity they add to the author's task, directly compromising their finite and valuable attention.

This is very important from both a complexity point of view and a cost point of view. Where a writer is asked to consciously reuse content they have to go looking for that content every time the potential for reuse occurs. This cost is incurred whether or not they find reusable content, whereas any savings from reuse are realized only when reusable content is found. But with this approach, the writer is relieved of all responsibility for the reusable content. Locating that content is the job of an algorithm. If it does not exist, the algorithm will report that it is missing and it will be somebody's job to create it, after which the algorithm will locate it automatically every time it is needed. This is far more efficient than authors looking for reusable content over and over and over.

The downside of this approach is that it is not as general. The *is-it-dangerous* metadata applies only to dangerous procedures. It does not address the inclusion of reusable content in other places. You would need to factor out other interesting reuse cases in a similar way to create a complete subject-domain solution. Again we see that complexity always has to go somewhere. But as we have also seen, if authors cannot fully handle the complexity thrust on them, that complexity ultimately gets dumped on the reader.

Factor out the variable

You can also factor out the variable content. For example, in the case of the different models of a car, rather than conditionalizing the list of features in the document, like this:

The main features of the car are:

```
o1:  
  li: Wheels  
  li: Steering wheel
```

```
li:(?deluxe) Leather seats  
li: Mud flaps
```

You can factor out the list entirely:

The main features of the car are:

```
>>>(%main_features)
```

You can then maintain the features list in a database. The organization probably already has a database of features for each vehicle, so we don't need to create anything new. We simply query the existing database. (After all, this is about reusing what already exists rather than recreating it!)

So now our algorithm looks something like this:

```
match insert with key  
  $resource = lookup key in lookup-table  
  output $resource
```

We then have a key lookup table where the resource is identified by a query on the database

```
keys:  
key:  
  name: %main-features  
  resource: from vehicles select features  
    where model = $model
```

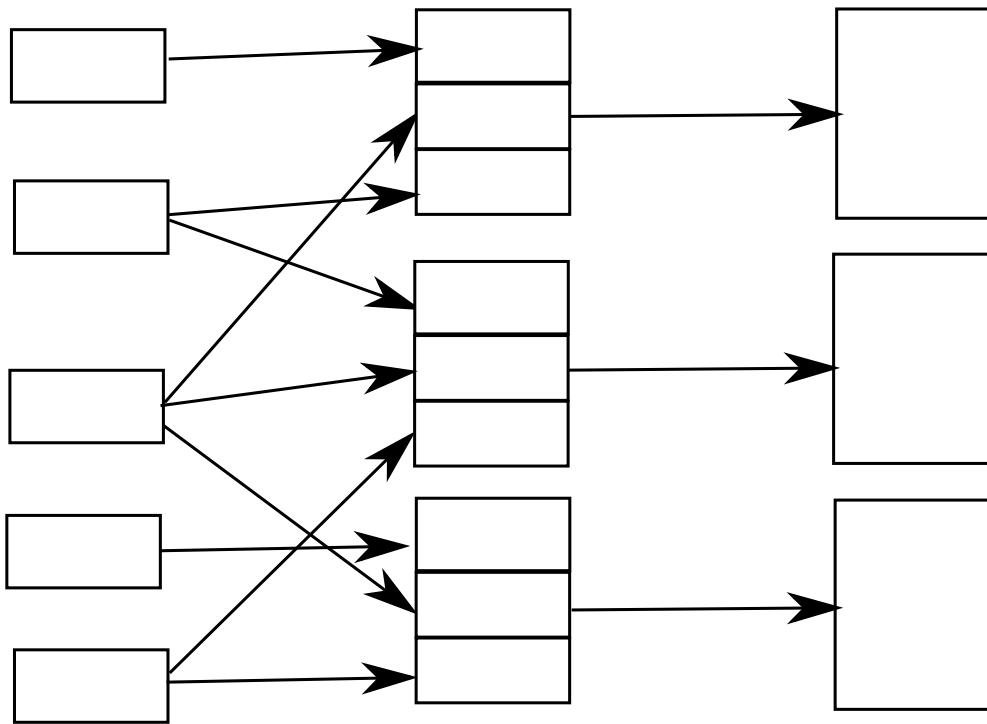
This retrieves a different set of features from the database depending on how the variable \$model is defined for the build. Launch the build with \$model = 'compact' and you get the feature set for the compact model. Launch the build with \$model = 'van' and you get the feature set for the van model.

Naturally, this is leaving out a whole lot of detail about how this query gets executed and how the results get structured into a document domain list structure. But those are implementation details.

Assemble from pieces

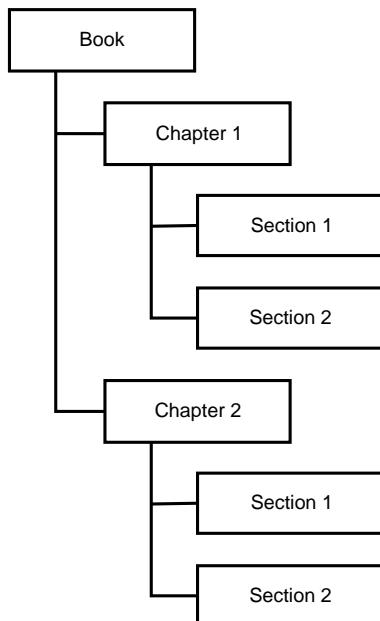
In the assemble from pieces approach, there is no common vs. variable distinction and no single source document into which reused content is inserted or to which conditions

are applied. Instead, there is a set of content units that are assembled to form a finished document.



For example, if you have a range of products with common features, you might assemble the documentation for those products using a common introduction with a piece representing each feature of each model.

This could be a flat list, or it could be a tree structure. For instance, you might assemble a chapter of a manual with a introductory piece and then several sections below it in the tree.



The assembly approach requires a structure to describe how the units are assembled. This structure is often called a map. (It is called a map in DITA, for instance.) Some applications may also refer to it as a table of contents.

```
map: Widget Wrangler Deluxe User Manual
    unit: units/ww/deluxe/intro
        unit: units/ww/shared/basic_features
        unit: units/ww/deluxe/deluxe_features
    unit: units/ww/shared/install/intro
        unit: units/ww/shared/requirements
        unit: units/ww/deluxe/requirements
        unit: units/ww/shared/install
        unit: units/ww/deluxe/install_options
```

Rather than using a map, you can allow the units themselves to pull in other units, which may in turn pull in other units. So the Widget Wrangler Deluxe install introduction unit might look like this:

```
unit: Installing the Widget Wrangler Deluxe
```

You should be very careful when installing the Widget Wrangler Deluxe. Follow these steps carefully:

```
>>>(unit units/ww/shared/requirements)
>>>(unit units/ww/deluxe/requirements)
>>>(unit units/ww/shared/install)
>>>(unit units/ww/deluxe/install_options)
```

This avoids the need for a map, but the downside is that it can make the units less reusable. In the above example, for instance, you would need a separate introduction unit for the regular Widget Wrangler since the introduction file imports all the requirements and procedural units. By assembling units with the map, you can use a shared install intro, which increases the amount of reuse you can do.

Combining multiple techniques

There is one problem with the idea of using a common install intro for both the regular and deluxe widget wrangler. The intro mentions the name of the product. To solve this problem without requiring two different units, we can use the variable into common or common with conditions reuse techniques within the intro unit. Here is an example using variable into common:

unit: Installing the >(\$product_name)

You should be very careful when installing the >(\$product_name). Follow these steps carefully:

```
>>>(unit unit/ww/shared/requirements)
>>>(unit unit/ww/deluxe/requirements)
>>>(unit unit/ww/shared/install)
>>>(unit unit/ww/deluxe/install_options)
```

There are a number of ways in which you can mix and match the basic reuse patterns to achieve an overall reuse strategies. Most systems designed to support reuse will allow you to do all these patterns and to combine them however you wish. However, you should not lose site of the amount of complexity that you are introducing into your content and into your writer's work when you rely on a complex set of reuse techniques like this.

Content reuse is not a panacea

Content reuse can seem like an easy win, and in some cases it can return substantial benefits, but there are pitfalls to be aware of. You will need to plan carefully to make sure that you avoid the traps that await the unwary.

Many reuse techniques introduce a lot of complexity into the author's job and into the content itself. These techniques may address major sources of content complexity in your organization but they come with a high cost in terms of new complexity introduced. Even if the reuse tools look easy to use, they can introduce large amounts of complexity into your process.

Quality traps

There are three main quality traps with content reuse.

- Making content too generic
- Losing the narrative flow
- Failure to address the audience appropriately

Many works on content reuse casually recommend making content more generic or more abstract as a means to making it more reusable, without saying anything about the potential downside. This is very dangerous and can do serious harm to the quality of your content. Statements that are specific and concrete are easier to understand and communicate better than statements that are generic and abstract. Replacing specific and concrete statements with generic or abstract statements will reduce the effectiveness of your content significantly. This is a classic case of dumping complexity onto the reader.

Unfortunately, human beings suffer from the curse of knowledge. The curse of knowledge is a cognitive bias that makes it very hard for people who understand an idea to appreciate the difficulties that idea presents to people who do not understand it. The curse of knowledge makes the generic or abstract statement of an idea appear equally communicative, and perhaps more succinct and precise, than the concrete and specific statement of it. This is a problem for writers at all times, pulling them away from the kind of specific and concrete statement that make ideas easier to comprehend. The desire to make content reusable reinforces this temptation.

Replacing the specific and concrete with the generic and abstract always reduces content quality and effectiveness. You may decide that the economic benefits of content reuse outweigh the economic costs of less effective content, but you should at least be aware that there are real economic consequences to this choice.

Another potential quality problem comes with the loss of narrative flow. Not all content has or needs a lengthy narrative flow, but if you start breaking your content into reusable units and putting them back together in different ways, the narrative flow can easily be lost. In some cases you can avoid this problem by making the topics you present to your audience more self contained using an Every Page is Page One information design. But don't assume that you have an effective Every Page is Page One design just because you have broken your content into reusable units. If that content was written in a way that assumed a narrative flow, it is not going to work when reused in a way that breaks that flow.

Finally, reuse can encourage us to come up with one way of telling our story that we present to all audiences. But not all audiences are alike, and the way we tell our story to one audience may not work for another audience. Good content tells a good story to a particular audience. Two different tellings of the same story do not constitute redundant content if they address different audiences.

Cost traps

It is easy to see content reuse as a big cost saving. Reusing content means you do not have to write the same content over and over again. It is easy to add up the cost of all that redundant writing and regard that number as pure cost savings from a content reuse strategy.

But all of the reuse techniques create multiple artifacts that need to be managed. This includes both content and processing code. You need a mechanism to make sure that your content obeys the constraints required to make the pieces of content fit together reliably. You need a mechanism to make sure that way you have done reuse actually produces the documents you want. The cost of such management can be non-trivial and the consequences of the management breaking down can be significant.

While reuse is supposed to reduce the cost of modifying content when the subject matter changes, there are hidden cost traps here as well. It is often not until the subject matter

changes that you find out if the content we have treated as common is really common once the subject matter changes. If not, you may have a complex management task to sort out what is really common and what isn't. This can involve complex edits that then have to be tested and verified. If you get everything right, you can realize major savings when it comes time to modify your content, but if you get it wrong, it can multiply costs. Not only that, it also means that the money you spent factoring duplicate content out in anticipation a change that did not happen is wasted as well. Reacting to changes that actually happen is sometimes cheaper than preparing for multiple changes that never happen.

If your content collection and its web of reuse relationships is not audited and validated regularly, it can become chaotic over time and lose cohesion. This can make adding new content or changing existing content increasingly difficult and expensive.

Depending of the techniques you use, content reuse strategies can complicate the lives of authors, which may reduce the pool of authors you can use or reduce their productivity.

With many reuse techniques, the writer has to find the content to be reused each time. Not only does this take time and impose a piece of complexity on the writer, it is a cost that has to be paid every time the writer looks for content to reuse, whether they find any or not. Indeed, if a writer is expected to look for reusable content before they write anything at all, much of their time may be taken up with unsuccessful content queries. Reducing the expense of content queries, therefore, has to be a major component of any general reuse strategy. We will look more at this in Chapter 14, *Normalization*.

Localized and constrained forms of reuse can avoid this cost trap. For instance, reuse that is focused on producing different versions of a manual for different versions of a product, is doing reuse in constrained places for constrained reasons does not require frequent broad-based content queries to find reusable content. It merely requires that the writers on the project be aware of the specific reuse strategies being used in that project (unless the reuse has been factored out altogether).

Some content reuse techniques are easy to use in non-structured ways and early in a project it may seem like a non-structured approach to reuse speeds things up by allowing writers to reuse content wherever they find it. Over time, however, this approach can lead to a rat's nest of dependencies and relationships between bits of content that makes it hard to update or edit the content with any confidence.

Once the cohesion and discipline of a content set starts to break down, the decline tends to accelerate. As it becomes harder to find content to reuse, more duplication occurs, which further complicates the search for reusable content, creating a vicious circle. As links and other content relationships break down, people tend to form ad hoc links and relationships to get a job finished, further tangling the existing rat's nest. Under the gun, it is almost always easier to get the next document out by ignoring the structure and discipline of the content set structure, but the effects of this are corrosive. Without consistent discipline, even in the face of deadlines, a reuse system can fail over time.

All of these issues can be managed successfully with the right techniques and the right tools, but they all introduce costs as well, both up-front costs and ongoing costs. Those costs have to be reckoned up and subtracted from the projected cost savings before you can determine if a content reuse strategy is really going to save you money.

One final trap: there may not be as much reuse potential reuse as your thought. Some organizations have plunged into reuse strategies in hopes of a big payoff only to find that they had far less reusable content than they first thought. The cost of systems and the complexity added to the process by complex reuse techniques require a high level of reuse to pay off. If that level is not there, your systems will end up costing you more than it saves.

Alternatives to reuse

There are three alternative to content reuse as a way of dealing with information that appears in more than one place:

Duplicate but label

Allow the information to be expressed in more than one place but make sure that it is clearly and unambiguously labeled so that it is easy to find when updates are required. If this sounds like an inferior solution, reflect that in fact there is all kinds of repeated information in the content of any organization and reuse techniques cannot possibly eliminate all of it without adding way more complexity into the system than they eliminate. No matter how much reuse you do, you are still going to have to search your content set for instances of content that needs changing whenever a major change in subject matter occurs.

Reduce duplication

Content reuse is always content duplication. Reuse is predicated on the idea that you actually want to duplicate information in multiple publications. This is actually a somewhat out of date idea. In the paper world, it was important to give each customer a book with all the information they needed in it, even if the same information occurred in many different books. But in an online environment, you don't need to duplicate information to make it available to every user who needs it. Maintaining the information once and linking to it whenever it is needed is a much less complex solution

for a number of cases. To do this efficiently may require some of the techniques described in Chapter 15, *Linking*.

Combine and differentiate

Another approach to reducing duplication in your content is to combine information on multiple products into a single publication and differentiate them in the text. There are limits to this, obviously, and some organization got into content reuse specifically so that they could remove the complexity of multiple versions in a single manual. Still, there are times when it is an appropriate solution.

Chapter 13. Composition

A major source of complexity in content reuse is the need to write pieces of content that will work when combined with other pieces to form a larger work. This is the composition problem. But the composition problem does not only occur in reuse scenarios. It can also occur in any situation in which you want multiple writers to contribute to a single work. In these scenarios, you have partitioned the writing of certain pieces of content so that they occur only once, and so that, presumably, they are written by the person best qualified to write them.

But as we have noted before, successful partitioning requires that sufficient information be passed between the partitions so that each can do its own work completely without any complexity being dropped. When a writing task is divided among many authors, we need to make sure that their individual pieces will come together to form a coherent whole. The definition of what constitutes a coherent whole may vary. A collection of Every Page is Page One topics organized as a hypertext has a different definition of what constitutes a coherent whole than does a printed product manual designed to be read in linear order. But whatever the criteria, we have to make sure they are met in order to distribute authoring or practice reuse without allowing elements of complexity to fall through to the reader.

The composition algorithm deals with the problem of composing a larger work out of smaller pieces. Many traditional writing tools produce files that are meant to encompass a whole work. If you take two Word files, for example, or two FrameMaker files and simply join them together, the result will not be a document that is the combination of the two files, it will be a corrupt file that will not open.

Both Word and FrameMaker have features that allow you to compose Word and FrameMaker documents out of smaller pieces. To a limited extent, they may even allow you to compose documents out of files other than their own. Some systems are specifically designed to allow you to compose document from many different source files. In many cases, though, some cleanup is required before the imported files can be used, particularly if they contain media domain formatting information. This cleanup requirement means that this is usually a one time import. You can't keep editing the original files and have the changes immediately reflected in the importing system.

Organizations that require a high degree of composability, therefore, have often turned to structured writing for solutions to the composability problem.

There are several parts to the composability problem, each of which structured writing helps address in different ways.

Fundamental composability

The first requirement of composability is that you must actually be able to combine the pieces. Most structured writing formats consist of a hierarchy of structures. Those structures tend to be self similar in form. For instance, all structures in an XML document are composed of XML elements. This means that you can take an XML document apart at any point in the structural hierarchy and insert, remove, or rearrange the structures at that level. To compose a larger structure out of smaller structures, you simply wrap new elements around them. Thus XML, and many other structured writing formats, provide a fundamental composability often lacking in other formats.

Structural composability

The second requirement of composability is that the result of combining markup structures must be a valid document (must conform to the appropriate constraints for that document). The simplest way to assure this is to plan all of your pieces to fit the constraints of the documents they will be inserted into. The most obvious way to do this is to make sure that all of your pieces come from the same markup language. Thus DITA has good support for structural composability, but only for sources that are themselves DITA.

But belonging to the same markup language is not enough. Markup language constrain where certain structures can occur and you must also make sure that the pieces go into a place where they are structurally allowed in that language. Just because all the pieces come from the same languages does not mean that every possible combination results in a valid document. Thus you cannot insert a DITA `steps` structure into a DITA concept or reference topic because `steps` are not permitted in those topic types. This requires planning and careful management to make sure the combinations you create are valid.

However, it is not essential to composability that all the pieces you want to combine come from the same language as the document you are composing. You can also take content from different sources and with different structures, as long as you can transform their structures on input to match the structures of the destination document. This can be a very powerful technique. For instance, you can use it to compose documents from content in a database. (Indeed, all database reporting systems are exactly this: systems that compose documents in one format from tabular data in another format.) Structural composability depends on the semantic equivalence of structures, not common syntax.

For this approach to work, however, it is important that all of the sources you draw from have a high level of conformance to their own constraints. If you don't know, or cannot rely on, the structure of the pieces you are drawing in, you cannot reliably combine them with an algorithm. Thus it is often better to focus on strategies for getting the most reliable sources rather than forcing everyone into a common format that they might not use reliably. For more on this, see Chapter 16, *Conformance*.

Stylistic composability

While structural composability is vital, it is not always sufficient. You could have pieces in a media domain language that are structurally composable but formatted differently. The resulting document would be valid and would publish successfully, but it would be a mess of competing styles and fonts.

For practical purposes, then, you should not try to create composable content in the media domain. You should at least use the document domain. The document domain separates content from formatting so you can compose a document in the document domain and then apply consistent formatting to the result.

This is true even if the pieces are in different document domain languages. All document domain languages essentially describe the same set of abstract document structures – documents are documents after all, they all have the same basic features which all document domain languages seek to represent. As long as you can recognize the same basic features in each of the source languages, you can compose a document from pieces in different document domain languages by converting to a common output language. (Embedded management-domain markup may spoil the party, however, since there is not the same level of semantic equivalence between management system. For this reason you generally want to do composition after the resolution of management-domain structures, unless those structures are actually doing composition themselves.)

Narrative composability

Even if you can assemble pieces from different document domain sources and format them all with a single consistent look, that does not mean that the result will be a complete, correct, coherent narrative. To create a coherent narrative, you need to ensure narrative composability.

This is not necessarily a matter of making the document sound like it came from a single person. Many business documents are the result of several different writers, sometimes working together, sometimes inheriting and maintaining a document over time. Truly making such a document sound like it was all written by one person is a tall order and usually not necessary to achieve its business purpose.

What does matter is that the document be cohesive and coherent. The terminology it uses should be consistent from beginning to end. The end should flow logically from the middle and the middle from the beginning. There should not be obvious duplications of content nor omissions (obvious or not). This clearly requires a number of constraints on the content affecting both composition and style.

There are a couple of approaches to narrative composability. One is the information typing approach that you find in systems like DITA or Information Mapping. In this

approach, content is broken down into certain broad types such as procedure, process, principle, concept, structure, and fact (Information Mapping) or task, concept, and reference (DITA – though DITA allows you to define others through specialization.) The idea here is that if you keep reference information in a separate chunk from a concept, for instance, the chunks will compose more reliably, since there will not be duplicate concept information in a reference chunk. (For more on these mechanisms, see Chapter 40, *Heavyweight markup languages*.)

The difficulty with this approach is that these abstract categories don't always make a lot of sense to writers when they are writing about concrete subjects, and different writers may interpret the chunk types or their boundaries differently, resulting in material that does not compose as well as you might hope.

Also, this approach, while it has been shown to improve the quality of writing in some cases, can also impose an artificial clunkiness and lack of flow on the content, leaving it feeling choppy or disjointed.

And, if one of your aims is to impose a specific rhetorical style or structure for particular kinds of content, any composition that you do needs to be subject to the same rhetorical constraints as if the piece had been written as a whole. This is often difficult to achieve, not least because it is often difficult to authors to write material that conforms to a rhetorical structure when they can't see the whole structure.

The other approach to narrative composability is to move content to the subject domain. A subject domain structure for a particular subject does not have to be structured as a collection of abstract chunk types. The structure is highly specific to the subject matter at hand and is therefore much more concrete and less susceptible to varying interpretation by writers. Also, you can use the subject domain to factor out many of the style issue that might otherwise compromise composability. (This is just like factoring out formatting issues by moving from the media domain to the document domain.) A narrative can then be composed algorithmically by arranging well-identified pieces of information in a predetermined order.)

Of course, not all material fits into obvious strongly typed subject domain structures. Content that is more conceptual or theoretical in nature does not have a strong subject domain structure because it does not approach its subject matter in such a systematic or regular way. Then again, the ability to compose such content out of existing pieces is limited anyway. By its very nature such content requires a continuous flow of exposition that is very hard to assemble from pre-written chunks.

Chapter 14. Normalization

A significant source of complexity in any content organization is simply determining if a piece of content already exists or not. This problem obviously affects any attempt at content reuse, since every time you consciously reuse content you have to determine if reusable content exists. The longer it takes to determine if a piece of content already exists, the longer, and therefore more expensive, each instance of content reuse becomes. (And bear in mind that the cost of determining if reusable content exists is incurred every time the writer looks for it, even if they don't find it, but any saving associated with reuse are realized only when reusable content is found.)

Even if reuse is not a major part of your process, however, determining if content already exists is still important, as it helps you avoid recreating content you already have, and all of the complexity that goes with maintaining two copies of the same thing.

Creating a formal system for ensuring that content only exists once is called normalization. Normalization is not just about eliminating duplicates from an information set – that is merely housekeeping. It is about creating a set of constraints by which duplication can be defined, detected, and eliminated. Normalization is not something that happens after the fact, after your data types are defined. It is something that is integral to the design process itself. It is about designing each data type in your system so that you can tell at once if one piece of data is a duplicate of another. Essentially it says that if item X matches item Y in aspects A, B, and C, then it is either a complete duplicate or it contains duplicated overlapping information. And this means that aspects A, B, and C have to be formalized as part of the model of X and Y if we are going to implement any algorithmic support for normalization, and even if we are going to have clear criteria for human decision making on what is or is not a duplicate.

Consider two movie reviews written in markdown:

Disappointing outing for the Duke

=====

After a memorable outing in Rio Grande
and Sands of Iwo Jima, John Wayne
turns in an pedestrian performance
in Rio Bravo.

and\:

Wayne's best yet

=====

After tiresome performances in Rio Grande and Sands of Iwo Jima, the Duke is brilliant in Rio Bravo.

Do we have two reviews of *Rio Bravo* or not? A human reading the text can tell easily enough, but an algorithms would have no way to tell. But suppose these same reviews were written in a subject domain movie review language:

```
movie-review: Disappointing outing for the Duke
movie: Rio Bravo
review-text:
    After a memorable outing in {Rio Grande}(movie)
    and {Sands of Iwo Jima}(movie),
    {John Wayne}(actor) turns in
    an pedestrian performance
    in {Rio Bravo}(movie).
```

and\:

```
movie-review: Wayne's best yet
movie-title: Rio Bravo
review-text:
    After tiresome performances in {Rio Grande}(movie)
    and {Sands of Iwo Jima}(movie),
    {the Duke}(actor, "John Wayne") is brilliant
    in {Rio Bravo}(movie).
```

Now it is very easy to tell that these two pieces of content are both movie reviews and that they both review the same movie. However, since they each take a very different view of the movie, it is fair to ask if the fact that they are both movie reviews about the same film is enough to consider them duplicates. That is a business decision that the content owner has to make.

You can do content reuse without the assurance that your content set is free from duplication, and organizations frequently do. The task of finding some content that meets your needs and can be reused is very different from the task of assuring that there is only one piece of content in existence that can meet a specific need. Finding content to reuse is a task for the writer doing the reusing. Ensuring that there is only one source for that content is a task for the writer of every piece of original content that is created. They have to make sure when they create and store some piece of content that there really is no other piece of content serving the same purpose.

There is a clear partitioning of responsibility here. Normalization partitions the problem of ensuring that only one copy of something exists from the problem of finding content

that can be reused. Unfortunately, normalizing your content set is a much more difficult task than simply doing ad hoc reuse. Many attempts at creating a reuse system founder because people focus on establishing ad hoc reuse capability without giving any thought to normalization. That neglected complexity has to do somewhere, and it tends to accumulate in the system until it brings it to a halt.

But the problem with establishing and maintaining a normalized content set is defining exactly when one piece of content is the duplicate of another. For some types of content, this question is easy to answer. What is the customer's birthday? A person can only have one birthday, so there is no difficulty creating a clear policy that says that a customer's birthday may only be recorded once across the organization and should be accessed from that single source whenever it is needed.¹

Normalization rules are constraints, and the set of constraints that defines a piece of information as unique are not universal. The definition of what constitutes unique for different pieces of information is complex and specific to the subject matter at hand. Take a recipe for guacamole, for instance. Is guacamole a single dish for which there can only be one recipe? Then normalization is easy enough. If the type of the item is "recipe" and the value of the dish field is "guacamole", then the content is duplicate.

But there are many different ways in which you can prepare guacamole, some differing only slightly from one another and some presenting welcome variations that different people might like to try. Clearly a recipe site would not want eight essentially identical guacamole recipes, but nor would they want to pick one variation to the exclusion of all others. So then the question becomes, how do you decide when a recipe is an effective duplicate of another text and when the subject is a welcome variation? If you decide the variation is welcome, how do you differentiate it from other guacamole recipes in your collection?

If we decided that we were willing to have multiple reviews of the same film in our collection as long as they gave different opinions, we might add a grading system to our review structure, and allow more than one review of the same movie to be created as long as each gave the film a different rating:

```
movie-review: Disappointing outing for the Duke
  movie: Rio Bravo
  5star-rating: 2
  review-text:
    After a memorable outing in {Rio Grande}(movie)
    and {Sands of Iwo Jima}(movie),
    {John Wayne}(actor) turns in
```

¹Stating the policy is straightforward; implementing and enforcing it may be more difficult, since it means every system or document that wants to include the customer's birthday has to be capable of retrieving it dynamically from the central data store.

```
an pedestrian performance  
in {Rio Bravo}(movie).
```

and\:

```
movie-review: Wayne's best yet  
movie-title: Rio Bravo  
5star-rating: 5  
review-text:  
After tiresome performances in {Rio Grande}(movie)  
and {Sands of Iwo Jima}(movie),  
{the Duke}(actor, "John Wayne") is brilliant  
in {Rio Bravo}(movie).
```

Now if our normalization rule is that a piece is a duplicate if it is a movie-review and the movie-title is the same and the 5star-rating is the same, then these two reviews are not duplicates because they have different 5star-ratings. (Note that you don't have to include every field in your type definition in your normalization rule, just those fields that determine if a piece of content is a duplicate or not according to your business rules.)

Clearly the answers to these questions are not universal. The way you decide these questions for recipes are not the same way you decide them for API reference topics, used car reviews, movie reviews, or conceptual discussions of ideas. Normalization happens in the subject domain and is specific to a particular type of content about a particular type of subject. To do any kind of content normalization you either need content in the subject domain or in a content management system that applies subject domain metadata to the content. (More on this in Chapter 23, *Content Management*). Whatever constraint you decide upon, the business processes and systems that ensure that these constraints are followed are not universal, but specific to each function and organization.

The limits of content normalization

Content, by its nature, deals with those subjects that do not fit neatly into rows and columns, and thus cannot be formally normalized according to database rules. Where databases describe and normalize relationships formally by use of records and keys, content describes relationships informally in prose. In particular, content deals with complex, unique, and potentially ambiguous relationships that could not be reduced to rows and tables at any reasonable cost.

Content is not as granular as typical database data. Content is essentially narrative, and the same fact may be mentioned in many different narratives for many different purposes. It may be elaborated on in one place, explained briefly in another, and merely mentioned in a third. For examples, in most Wikipedia articles on countries, there is a section on the economy of that country, and at the beginning of that section there is a link to an entire

article describing the economy of that country, followed by a summary coverage of that country's economy presumably much briefer and less detailed than that provided by the main article. There may also be a brief mention of the highlights of the country's economy in the four or five context-setting paragraphs that lead most Wikipedia articles. These different levels of detail serve different user needs, and so each is a valuable contribution to the content set. But how do you normalize the statement of any of the facts that appear in more than one of these places? They each present the same fact in a different way for a different purpose and audience.

Content is always designed for a particular audience, both to serve a particular need and to suit a particular background and level of knowledge. Everything we know about effective content tells us that we need to address different audiences and different tasks differently. Taking a piece of content designed for one audience and using it for all other audiences, or attempting to write generic content that takes no account of any audience's needs or tasks is certain to produce content that is significantly less effective. This is a classic case of dumping complexity on the user, though in this case it is more a case of doing it by deliberate action rather than failure to properly manage the natural complexity of content creation.

Examples of content normalization

The way we achieve the normalization of a fact is by abstracting it out of all of the expressions of that fact. Abstracting variants from invariants is, of course, what structured writing is about. A simple case of abstracting out a fact is the replacement of certain pieces of text by variables. In technical communication, it is a frequent practice to replace things like company names and product names by variables:

Thank you for buying >(\$company-name)'s >(\$product-name).
We hope you enjoy it for many years to come.

There are a couple of reasons for doing this. One is that company names and product names often have a precise formal variant that the marketing department wants everyone to use, and one or more informal variants that people actually use. There is constraint here: use the formal name for company and product not the common name. But because authors are more used to using the common name (like everyone else) they are likely to slip it in without thinking. And if they do remember that they are supposed to use the formal name, they may not remember it correctly and so use the wrong form. Using the variable instead tries to ensure that the correct version of the name is used.

The second reason is that products and companies are sometimes re-branded. The names change. If the names are written into the source in a hundred places, you will need to find those hundred instances and change them all. If a variable is used, you just need to change the definition of the variable and you don't have to touch the source content at all. For more on this approach and its pitfalls, see Chapter 22, *Change management*.

This is clearly a very tactical approach to normalization. We have a single source of truth for the name of the company and the name of the product. We don't have a single source of truth for every name of every subject we mention. We only do it for certain names we consider critical or subject to change.

Other examples of normalization concern the normalization of entire content units. Obvious and natural examples come from reference material. For example, an API reference entry is normalized based on the name of the library it belongs to and the name of the function it is describing. If you have an entry for the `hello()` function of the `greetings` library, then it should be easy to detect if someone tries to create another entry for that function. With some repository systems, it might be impossible to create the duplicate at all.

A great deal of your ability to normalize your content comes down to your ability to clearly name the things you are writing about. API library and functions are clearly and unambiguously named. Is the set of your product's feature names as clear and unambiguous? Sometimes features correspond to buttons you can press on the interface, but sometimes they are more marketing terms that refer to business problems solved. Can you agree across the company on what such feature names are? Is that even desirable, or would one list of feature names market the product better to one consumer group and a different list to a different group?

The ability to name things consistently is a compounding factor in any attempt to normalize content, or indeed to detect or limit duplication by any means. Duplication can be hidden by the same fact being referred to by different names. The elimination of duplication can be made more difficult when the same word refers to more than one truth. But merely standardizing terms is not a solution, because different readers frequently use different terms to refer to the same ideas, or use the same term to refer to different ideas. Structured writing techniques can help here by putting such ambiguous terms into contexts where their meaning may be easier to delineate. I will look at this in more detail in Chapter 36, *Terminology*.

Normalization and discovery

Since normalization rules define when a piece of content is unique, they make it easy to determine if a piece of content exists. If you know which fields of a proposed content item are used to normalize it in the content set, you can query for a topic that has the same values in those fields. If you find one, you are confident the content already exists; if you don't, you can be confident that it does not exist and needs to be written.

At least, in theory you can. The problem is that not all content can be normalized to the same degree. If you have a set of tightly-constrained subject-domain content types, you can be reasonably sure that if you don't find an existing item in the set of items of its type, you also won't find it in any of the other tightly-constrained types, because such content would not fit those type definitions. But what about your less constrained

content? All content sets, even those that have been as tightly constrained as possible, will have some unconstrained or loosely constrained content. Some subjects just don't lend themselves to tightly constrained content types, while others might only occur once or twice in your content set, making it pointless, or, at least, expensive, to tightly constrain that information.

For example, while we know with a high degree of certainty that there is only one API reference entry on the `hello()` function of the `greetings` library, it is much harder to detect if a writer decides to insert a full description of the `hello()` function into a topic in the programmer's guide. Programmer's guides typically deal with the relationships between different APIs and other parts of the system, and on how to accomplish certain real world tasks with the system as a whole. This focus may lend itself to a reasonably strict content type for programming topics, but it does not lend itself to strict normalization through a few unambiguous fields like library name and function name. Detecting that the author of a programming topic has duplicated information provided by the API reference may therefore be difficult.

And it is also possible that while the programming guide author has indeed duplicated information from the API guide, it may also be the case that they needed to do so. If, for instance, they are explaining the reasons one might choose to use a function from the `salutations` library rather than the `greetings` library, then explaining the differences between the `hello()` function in each library is necessary, and necessarily involves repeating some of the information in each libraries API reference. Simply referring the reader to each API reference to compare and contrast for themselves would eliminate the duplication, but at the expense of dumping the complexity of detecting and understanding the differences onto the reader.

Proximity detection

A less rigorous approach to detecting potentially duplicate content is what we might call “proximity detection” – pieces of content with several points of similarity which might indicate that they are covering the same subject, or at least that their coverage overlaps.

For instance, you might look at the list of terms indexed in each topic, or the list of subject annotations they contain (pretty much the same thing, in the document and subject domains respectively) and if there are matches above a certain threshold you might inspect them to see if there are duplications.

Proximity detection is much easier when all of your content items are as constrained to subject-domain content types as is reasonable for their subject matter. Strongly defined content types tend to be more cohesive – meaning they cover the same piece of ground and cover it more consistently for each instance of the subject matter. Without strong types, different authors may chunk up ideas and information differently, so that topics from two different authors may partially overlap each other. Not only is partial overlap harder to detect, since there will be fewer points of similarity between the items, it is

also harder to fix because each item contains different information that the user needs. Eliminating one of the duplicates means find a place for all of the extra information it contains, a process that could potentially affect several other content items and perhaps raise other normalization questions.

Normalization via aggregation

One way to normalize content is to aggregate information at source that will be output to different documents. For instance, if you have a product with multiple versions, you can aggregate the information about all of the versions into a single source file and then publish only the blocks that apply to a particular version. (This is an applications of the common with conditions reuse algorithm that we looked at in Chapter 12, *Reuse*.)

Doing this reliably is greatly aided by moving content to the subject domain where you can effectively create a small database of product facts in a subject-domain source file and have algorithms select and publish those that belong to each version of the product. For instance, including both the alcoholic and non-alcoholic beverage match in our hard boiled egg recipe allows us to publish it in both *Wine Weenie* and *The Teetotaler's Trumpet*.

Overall, then, a full normalization of a content set is not possible. The variations on how, when, and why you mentions particular facts in content, and the various purposes and audiences for which you describe different subjects make it impossible to have hard and fast rules about what is an is not a duplicate. Too zealous a pursuit of normalization can result in the elimination of valuable differences in information and presentation – dumping the complexity of managing all the instances of a description of a subject on the head of the reader in the form of excessively generic content that is harder for them to understand and may not give them all the information they need.

However, moving as much of your content as possible to the subject domain with give you a fair amount of useful normalization without any additional effort – the subject domain tends to enable additional algorithms from the same structures and without additional effort or expense.

Chapter 15. Linking

A large part of the complexity of content is that readers do not read in a straight line. Numerous studies¹ have shown that readers often read opportunistically, looking for one piece of information or the answer to a particular question. Their behavior can best be described as information foraging. They are sniffing for the scent of the information they want and will follow its trail as long as the scent gets stronger and the trail is easy to follow. They use search to start their quest for information and will follow links as long as they feel the information scent is increasing. No two readers take the same path though your content. They want to forge their own path based on their particular interest of the moment.

The path the reader takes is always based on relationships between the subject matter of one piece of content and the subject matter of another. Suppose a reader is reading a recipe and comes across the instruction “sweat the onions” but does not know how to do that. They are then going to start looking for information on how to sweat onions because they can’t usefully continue with the recipe without knowing how to do this. This is a deflection point in your content, a point at which the reader needs to deflect from the current content to some other source of information.

But the deflection point is not an arbitrary one. It is based on the relationship between the task of making the dish described in the recipe and a cooking technique used in that task. We can fully anticipate that such deflection points will occur in our content, and that some of our readers will wish to deflect at those points. The need to make such deflections is part of the complexity of information seeking. The point of information delivery is to make information seeking simpler. If we don’t make adequate provision for our readers to deflect when they need to, we are dumping complexity on the reader. If we force them to use search again or to go to a table of contents or an index when we could easily provide a direct link, we are dumping complexity on the reader.

But linking is not the only way to handle deflection points. The writer may also choose to use footnotes, cross references, sidebars, or parenthetical material to provide deflection choices. They may use tables or flowcharts to allow readers to choose different paths through content. They may even attempt to anticipate and forestall deflection by using information about the individual reader to dynamically reorder the content to suit the reader’s needs. A discussion of linking, therefore, needs to take account of other types of deflection which may be appropriate in particular circumstances.

If we reuse content in different media, we might want to have a different deflection strategies between paper and hypertext outputs. We may want to include the same chunk of content in multiple paper documents but link to a single copy of it when creating a

¹For a discussion of these studies and their implications for how we write, see my book *Every Page is Page One: Topic-based Writing for Technical Communication and the Web*.

hypertext. (Linking, in other words, is a kind of reuse: reuse by reference rather than copying.)

Thus we should not be thinking solely in terms of managing links in our content. We should be thinking about implementing the right deflection strategy in each of our outputs.

The problem is that implementing these strategies, particular a comprehensive linking strategy, is itself very complex. Thus many organizations do not link their content adequately and do not provide adequate alternative deflection strategies either. They just dump the complexity of the reader finding the next piece of information they need right back on the reader.

However, there are ways to partition the complexity of linking and other types of deflection support, making them easier and more economical to manage. Different approaches work in each of the structured writing domains, but some domains offer much more effective partitioning than others.

Deflection in the media domain

In the media domain, we simply record the various deflection devices as such: cross references, tables, links, etc. For example, in HTML a link simply specifies a page to load:

```
<p>In Rio Bravo,  
<a href="https://en.wikipedia.org/wiki/John_Wayne">the Duke</a>  
plays an ex-Union colonel out for revenge.</p>
```

The phrase “the Duke” is a deflection point. The reader may not know who “the Duke” is, or may want more information on him. The link supports the reader at the deflection point. The reader can either deflect by clicking the link or stay the course and read on.

But if the HTML page gets printed, the link is lost. The phrase “the Duke” is still a deflection point. The reader can still deflect, by doing a search for “the Duke”, perhaps, or asking a friend what it means. But the printed version lacks any support for that deflection.

If the content had been written for paper, the deflection point might be supported in a different way. For example, it might be supported by adding an explanation in parentheses:

In Rio Bravo, the Duke (John Wayne) plays an ex-Union colonel out for revenge.

Or it might be handled with a footnote:

In Rio Bravo, the Duke* plays
an ex-Union colonel out for revenge.

...

* "The Duke" is the nickname of the actor John Wayne.

Clearly this is a case in which we would like to do differential single sourcing and handle a deflection point differently in different media. To accomplish this, we need to move the content out of the media domain.

Deflection in the document domain

Moving to the document domain is about factoring out the formatting specific structures of the media domain. But a link is not really a piece of formatting, so conventional refactoring into abstract document structures is not going to apply. For this reason, people working in the document domain often enter hypertext links exactly the way they would in the media domain: by specifying a URL. Thus in DITA you might enter a link as:

```
<p>In Rio Bravo,  
<xref href="https://en.wikipedia.org/wiki/John_Wayne"  
format="html">The Duke</xref>  
plays an ex-Union colonel out for revenge.</p>
```

The difference from HTML is slight here. The link element is called `xref` rather than `a`. But the meaning of `xref` is bit more general. The HTML `a` element is saying, create a hypertext link to this address. The DITA `xref` element is saying, create some sort of reference to this resource. (As we will see in a moment, it is capable of linking to things other than HTML pages, which is why it requires the `format` attribute to specify that in this case the target is an HTML page.) This generality gives us a little more leeway in processing. We can legitimately create print output that looks like this:

In Rio Bravo, the Duke (see: https://en.wikipedia.org/wiki/John_Wayne)
plays an ex-Union colonel out for revenge.

This is not the way we would handle the deflection point if we were designing for paper, but it is a definite improvement from a differential single sourcing point of view. At least the link is now visible to the reader.²

²Technically we could do this from the HTML markup as well, but that would be cheating. The HTML markup is not really giving us permission to do this. It is telling us to create a hypertext link and nothing else. The problem with cheating is that you are basing your algorithm on constraints that are not being promised or enforced, and this can fail in ways you may not expect or catch. Some cheats are more reliable than others, but you probably don't want to get into the habit. Better to move your content creation to a format where cheating is not required to get the output you want.

Fundamentally, though, this is not a satisfactory differential single sourcing solution. Unless there were no alternative (such as when you are citing a specific source), you would not normally direct a reader of paper to the web for more information, nor vice versa. Linking to an already published file, such as an HTML page, means linking to the address where the published file resides. It commits us to a particular format for the link target. If instead of linking to the published address, we link to the address of the unpublished source file, or to an identifier for that file, we gain the freedom to link to any format of that content that we choose to publish.

In DITA, you can link to another DITA file (the default format, so we don't need the `format` attribute):

```
<p>In Rio Bravo, <xref href="John_Wayne.dita">The Duke</xref>  
plays an ex-Union colonel out for revenge.</p>
```

We don't yet know if that DITA file will be published to paper or the Web, what the address of the published topic will be, or if that topic will stand alone or be assembled into a larger page or document for publication. This means that the publishing system is taking on responsibility for both ends of the link. It has to make sure that the target page is published in a way the source page can link to, and that the source page links to the right address. But transferring this responsibility to an algorithm gives us the leeway to publish this link as we see fit.

If we publish as a book on paper and the target resource ends up as part of a chapter in the same book, we can render the `xref` as a cross reference to the page that resource appears on. We could format that cross reference inline or as a footnote. These are all legitimate interpretations of the `xref`'s instruction to create a reference to a resource.

If we publish to a help system and the target resource ends up as a topic in the same help system, we could render the `xref` as a hypertext link to that topic.

This is a big step forward, but it still does not let us do this:

In Rio Bravo, the Duke (John Wayne) plays an ex-Union colonel out for revenge.

In other words, we can render the `xref` as a cross reference or a link or a footnote, but we can only handle the deflection point as a reference to the specified resource. We can't decide to link to a different resource or handle it by parenthetical clarification instead. To give ourselves the ability to link to different resources, we can turn to the management domain.

Deflection in the management domain

Linking to a source file rather than to an address gives us more latitude about how the link or cross reference is published, but we are still always linking to the same resource.

If we are doing content reuse, this is a problem because you do not know if the same resource will be available everywhere we reuse our topic. We need to be able to link to different resources when our topic is used in different places.

To accommodate this, we can factor out the file name and replace it with an ID or a key. IDs and keys are management domain structures that we looked at in Chapter 12, *Reuse*. They allow us to refer to resources indirectly. Using IDs lets us use an abstract identifier rather than a file name to identify a resource. Using keys lets us remap the resources we point to. This makes keys the more efficient way to address this problem. So instead of referring to a specific resource on John Wayne, we refer to the key `John_Wayne` using a reference to a key. In DITA this would look like this:

```
<p>In Rio Bravo, <xref keyref="John_Wayne">The Duke</xref>  
plays an ex-Union colonel out for revenge.</p>
```

Somewhere in the DITA map for each publication, the key `John_Wayne` points to a topic. Publications link the `keyref` to the resource pointed to by that key in each of their DITA maps. This allows you to link to different resources in each publication.

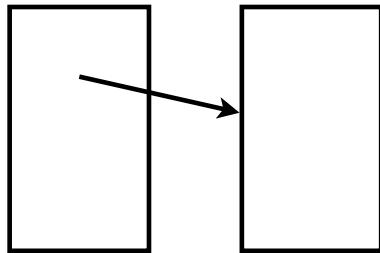
The problem with IDs and Keys

However, there is still a problem with linking based on IDs and keys. Keys will let you vary which resource a `keyref` resolves to, but what happens when there is no resource to which that key can reasonably be assigned?

The `xref` demands that a reference to a resource be created, but there is no resource to link to. You are going to have a broken link, and fixing it is not easy. You can't simply go in and remove the `xref` from the source for one publication, because it defeats the purpose of content reuse if you have to edit the content every time you reuse it. Removing the key reference would fix your broken link in one publication, but that would result in the link being removed from all the publications, even where the resource does exist and the link ought to be created.

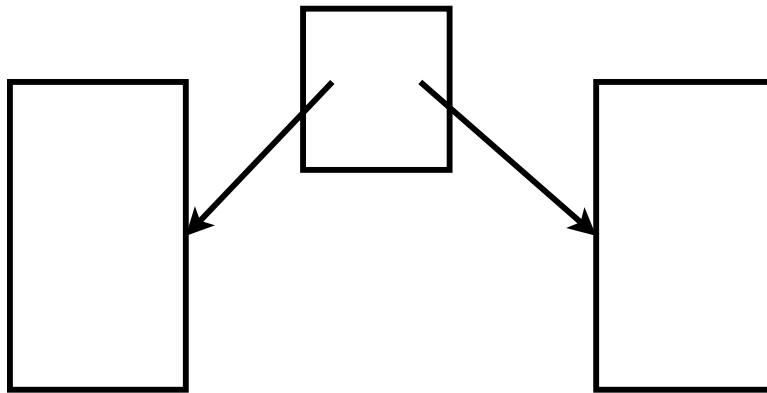
Relationship tables

One approach to the link-only-when-resource-available problem is to use a relationship table. In a conventional linking approach, the source page contains an embedded link structure pointing to the target page. The source knows it is pointing to the target, but the target does not know it is being pointed to.



The idea that the target resource does not know it is being pointed to is important because it means it does not have to do anything in order for other resources to point to it. The fact that only the source and not the target has to know about the link is fundamental to the rapid growth of the Web. If the target resource had to participate in the link process, every link would require negotiation between authors and the agreement of the author of the target resource to edit it to accept the link. It would be impossible for the Web to grow explosively and organically as it has under these conditions.

A relationship table takes this one step further. When you create a link using a relationship table, you factor the link out of the source document and place it in a separate table. The relationship table says resource A links to resource B, but neither resource A nor resource B knows anything about it. (Think of it like being introduced to a stranger by a third party because you share a common interest. I collect china ducklings. You make china ducklings. We don't know each other, but our mutual friend Dave introduces us. You and I are the source and destination resource; Dave is the relationship table.)



Once the links are factored out of a piece of content, you can reuse it anywhere you like. If there is a suitable resource available to link to, you enter it in a relationship table for that build and have the presentation algorithm create the link at build time. If no suitable resource is available for a different publication, no entry is made in the relationship table for that publication, and the presentation algorithm does not create a link.

The problem with relationship tables

The problem with relationship tables is that they separate the link from the deflection point it supports. The link that was created to serve the deflection point in the content,

the point at which the reader encountered something they might not understand, or might need more information on, or might find more appropriate to their needs than what they are reading now. But the relationship table does not record the deflection point, it says that there is some connection between topic A and topic B. It does not say where in topic A the deflection point that creates the relationship is located. The deflection point itself has not been marked up, so there is no way to put the link back where it belongs. Links generated by relationship tables end up in a block, usually at the end of the page.³

The fundamental problem here is that links exist as a way to service deflection points in content. If we lose sight of this and think only in terms of managing links, it makes sense to pull links out of the content and manage them separately. But this is to ignore the reason the links exist in the first place – which is to provide the reader with options at a deflection point. It is a classic case of a focus on one part of the partition problem without adequate regard for the complexity that is being redirected. By moving the link away from the deflection point, we simplify link management, but in doing so we lose a level of support for the reader's need to deflect, thus dumping the complexity of navigating from the deflection point to suitable content onto the reader, damaging content quality in the name of process efficiency.

The other problem with the relationship table approach is that it is time consuming. You have to rewrite the links for each content set, and because the deflection points are not recorded in the content source, you have to figure out the appropriate links each time. This goes against the spirit of recording something once and using it many times. A mechanism intended to help you reuse content ends up forcing you to redo the work of linking for each publication you create. In other words, this mechanism does not partition the complexity of link management very well.

Conditional linking

Before we leave the management domain, it is worth mentioning a management domain approach that we could use to address our differential single sourcing problem and get the appropriate deflection strategy for online and paper publishing. We could use conditional structures to define both options in the source file. With a little specialization to support media as a conditional attribute, you could do this in DITA:

```
<ph media="online"><xref keyref="John_Wayne">The  
Duke</xref></ph><ph media="paper">The  
Duke (John Wayne)</ph> plays an ex-Union  
colonel out for revenge.</p>
```

³It is not impossible to imagine a markup system in which you could markup the source of the deflection point in topic A, and then have the relationship table reference that deflection point by ID, thus allowing you to put the link at the deflection point. But this is clearly cumbersome for writers to implement and would complicate the management task. I don't know of any systems that work this way.

In DITA, the `ph` element is used to delineate an arbitrary phrase in the content that you want to apply management domain attributes to. Here we define two different versions of the phrase “the Duke”, each with different forms of deflection support (parenthetical expression for paper, link for online), and each with a corresponding media condition. The synthesis algorithm would then choose the appropriate version of the phrase for each publication based on the conditions set for the build.

There are some pretty obvious problems with this approach. It is twice the work for authors to create every link, and it doubles the maintenance cost of the content as well. It also flies in the face of the idea of creating formatting-independent content.

Unfortunately, in a general purpose document domain markup language with management domain support, it is pretty much impossible to prevent writers from doing things like this in order to achieve the effects they want. And in practice writers do end up using conditional markup like this for all kinds of differential single sourcing and reuse problems that are not easy to solve in the document and management domains. In some cases this can lead to tangles of conditions that are hard to maintain and debug.

The inescapable fact is that the management domain approach does not do a great job of partitioning the complexity of links, and of deflection point handling in general.

- Like all management domain structures, they are artificial. They don’t correspond to things in the author’s everyday world, which makes them harder to learn and use. They introduce complexity into the author’s world, which is sufficient to make most authors severely limit the links they create.
- You can’t link to a key or an ID that does not exist. This means that as you are developing a set of content, the first pages you write have very few other pages to link to. Authors cannot enter links to content that has not been written yet. This introduces the complexity of system dependencies into the author’s world.
- In reuse scenarios, the use of IDs and keys does not solve the whole problem because it cannot guarantee that the resource that an ID or key refers to will be present in the final publication. You can use relationship tables to address this problem, but they create additional complexity for authors and don’t correctly handle deflection points in the middle of the content.
- Unless you resort to ugly conditional structures, you can’t use media-appropriate deflection mechanism for differential single sourcing.

Deflection in the subject domain

As we have seen before (Chapter 12, *Reuse*), we can often remove the need for management domain structures by moving content to the subject domain. The same is true with deflection points. In the document domain we handled a deflection point by

specifying a resource to link to, specifying both that the deflection mechanism would be a link and that the link target would be a particular page. In the management domain we used keys to factor out the target resource but not the deflection mechanism (it was still an `xref`). In the subject domain, we can factor out the target resource as well. We do this by marking up the subject of the deflection point:

```
<p>In <movie>Rio Bravo</movie>,
<actor name="John Wayne">the Duke</actor>
plays an ex-Union colonel out for revenge.</p>
```

This markup clarifies that the phrase “the Duke” refers to the actor named John Wayne. These are respectively the type of the subject (actor) and its value (John Wayne).

The subject domain markup is not link markup. Unlike the document domain markup, it does not insist that a reference should be created nor does it specify any resource to link to. This markup is a subject annotation. It clarifies that the phrase “the Duke” refers to the actor named John Wayne (and not the Duke of Wellington or the Duke of Earl) and that the phrase “Rio Bravo” refers to the movie (and not to the city in Texas or the nature reserve in Belize⁴).

Given this markup, we can easily create the paper-style deflection mechanisms we have been looking for. We simply have the presentation algorithm take the value of the `name` attribute and output it between parentheses:

```
<p>In Rio Bravo, The Duke (John Wayne) plays an
ex-Union colonel out for revenge.</p>
```

It also allows us to create a link if we want to. We’ll look at how in a moment. But first we should look at the implications of subject annotation more deeply.

Subject annotation markup says, “this is an important subject that we care about in this context”. How is this an appropriate way to handle a deflection point? The intersection of subjects – where a topic on one subject makes a significant mention or another significant subject -- are likely deflection points. When we create a link in the media domain or the document domain it is because our text is mentioning a significant related subject that the reader may want to know more about. The difference is that in the document domain we handle the mention of an important subject by creating markup that specifies a resource on that subject and in the subject domain we create markup that specifies the subject itself. This leaves us with more options about how to handle the deflection point, and that is what we have been looking for.

Marking up a phrase as a significant subject does not oblige the publishing algorithm to create a link. If you decide to have the publishing algorithm create a link on the Web

⁴[https://en.wikipedia.org/wiki/R%C3%A9v%C3%A9sto_Bravo_\(disambiguation\)](https://en.wikipedia.org/wiki/R%C3%A9v%C3%A9sto_Bravo_(disambiguation))

and a cross reference on paper, nothing in the markup obliges you to use any particular formatting or target any particular resource. There is no question of cheating here if you decide to create one kind of deflection device or another, or not to create one at all.

In all our previous examples based on this text, mentions of “Rio Bravo” were not marked up, even though it is clearly an important subject and a potential deflection point. This reflects the author’s decision not to create a link to support this deflection point. But what if we want to make a different choice later? By marking up “Rio Bravo” as a significant subject, we keep our options open. Now we tell the presentation algorithm to create links on the names of movies if we want to, or not if we don’t want to.

But there are additional reasons to annotate Rio Bravo as a significant subject, because that annotation can be used for other purposes as well.

1. The subject annotation says that “Rio Bravo” is the title of a movie. In the media domain, the titles of movies are commonly printed in italics. We can use the subject domain `movie` tags to generate media domain italic styling.
2. We could use this subject annotation to generate document domain index markers so that we can automatically build an index of all mentions of movies in a work.

Subject annotation thus serves multiple purposes, and correspondingly reduces the amount of markup that is required to support all these different publishing functions. This is a common feature of subject domain markup. None of it is directly tied to specific document domain or media domain structures which will be required to publish the content. Each piece of subject-domain markup may be used to generate multiple document domain and media domain structures. For example, we could generate the following document domain markup from from the subject domain markup above (the example is in DocBook):

```
<para>
  In
  <indexterm>
    <primary>Rio Bravo</primary>
    <secondary>Movies</secondary>
  </indexterm>
  <citetitle pubwork="movie">Rio Bravo</citetitle>,
  <indexterm>
    <primary>John Wayne</primary>
    <secondary>Actors</secondary>
  </indexterm>
  <ulink url="https://en.wikipedia.org/wiki/John_Wayne">
    The Duke
  </ulink>
  plays an ex-Union colonel out for revenge.
```

</para>

This sample contains index markers, formatting of movie titles, and links on actor's names, all generated based on the subject annotations in the source text. It should be clear how much less work it is for an author to create the subject domain version of this content than the DocBook version. Yet all the same publishing ability is maintained in both version.

Generating links from subject annotations has a number of other advantages:

- In a reuse scenario, you never have to worry about broken links or creating relationship tables. The presentation algorithm generates whatever links are appropriate to whatever topics are available.
- In a differential single sourcing scenario, you are never tied to one deflection mechanism. You can generate any mechanism you like in whatever media you like.
- You don't have to worry about maintaining the links in your content because your source content does not contain any links. The subject annotations in your content are objective statements about your subject matter, so they don't change. All the links in the published content are generated by the linking algorithm, so no management is required.
- There is no issue with wanting to link to content that has not been written yet. The subject annotation refers to the subject matter, not a resource. Links to content that is written later will appear once that content becomes available to link to.
- It is much easier for authors to write because they do not have to find content to link to or manage complex link tables or keys. They just create subject annotations when the text mentions a significant subject. This requires no knowledge of the publishing or content management system. It does not even require knowledge of any other resources in the content set. It only requires knowledge of the subject matter, which the author already has.

Finding resources to link to

The subject domain approach represents a radically different partitioning of the complexity of linking and deflection point handling. Most notable, it partitions and redirects the responsibility for finding content to link to away from the author. All partitioning introduces some new form or complexity to replace the one that has been partitioned out. In this case, it is the complexity of doing the subject domain annotations of the text. But this is a particularly effective piece of partitioning both because that annotation only requires knowledge the writer already has, but also because that same annotation is useful for so many other algorithms as well. This is one of the best features of the subject domain – its structures tend to serve multiple algorithms rather than just one, which is, in itself, a highly effective piece of partitioning of complexity.

But since we have partitioned off the responsibility of finding resources to link to, we also have to look at how the people and processes in that partition go about locating appropriate resources. They do this by looking up resources based on the subject information (type and value) captured by the subject annotation. For this we need content that is indexed using those types and values (or their semantic equivalents). So naturally this means that we need to index our content. If you have a page on John Wayne, you can index it like this:

```
topic:  
    title: Biography of John Wayne  
    index:  
        type: actor  
        value: John Wayne  
body:  
  
    John Wayne was an American actor known for westerns.
```

Now the linking algorithm looks like this:

```
match actor  
    $target = find href of topic  
        where type = actor  
        and name = @name  
    create xref  
        attribute href = $target  
    continue
```

However, content stored in the subject domain may already be indexed effectively enough by its inherent subject domain structures. Suppose your content collection includes this subject-domain actor bio:

```
actor:  
    name: John Wayne  
    bio:  
        John Wayne was an American actor known for westerns.  
    filmography:  
        film: Rio Bravo  
        film: The Shootist
```

Here the topic type is actor, and the name field specifies the name of the actor in question. This is all the information we need to identify this topic as a source of information on the actor John Wayne.

Only very minor changes to the linking algorithm are required to use this:

```
match actor
    $target = find href of actor topic where name = @name
    create xref
        attribute href = $target
    continue
```

There is a lot more to how this mechanism works in practice, including what you do about imperfect matches and what happens when the query returns multiple resources. But that takes us into the specifics of individual systems and that is more detail than we need for present purposes.

Indexing of topics may also be done by a content management system, in which case the linking algorithm would query to CMS to find topics to link to.

A useful feature of this approach is that you can have the publishing algorithm fall back to creating a link to an external resource if an internal one is not available. If a search of the index of your own content fails, you can search indexes of external content. You can build such an index yourself, but some external sites may also provide indexes, APIs, or search facilities that you can use to locate appropriate pages to link to.

Of course, building these linking algorithms is a piece of additional complexity in your content system. The effect of the subject domain it to partition complexity away from writers and distribute it to information architects and content engineers.

Deferred Deflection

Readers don't always deflect the moment they reach a deflection point. In some cases, they choose to set the alternate material aside for later reading. This is particularly easy to do on the Web, where you can simply open pages in new browser tabs for reading later.

The idea of the deferred deflection can also occur in document design. A document design that gathers a set of links together at the end of a document, rather than including them inline, is recommending deferred deflection to the reader. It attempts to keep the reader following the writer's default course to the end of the document before they go off to other things. The relationship table approach to link management that we mentioned earlier can only produce deferred links.

The merits of deferred links are debatable. Some argue that inline links are a distraction, that they actually encourage deflection. But the lack of links does not stop the reader from deflecting if they want to, and if they do deflect, the lack of a link means they may leave your content set and land on competitor's content or content that is of poor quality or that contradicts what you have been saying. The fact that the debate exists, however, suggests that we may want to factor this design choice out of our source content so that we can choose between inline and deferred links later.

To leave open the option of deferring or not deferring links, we have to record links at the deflection points they belong to. We can choose to defer them at publishing time if we wish, but if we defer at writing time, we can't put the links back inline at publishing time because we don't know where they belong.

But for this strategy to work, we need to be able to tell the difference between links that can be deferred and those that cannot. A simple example of a link that cannot be deferred is one that says "For more information, click here." Obviously this link has to remain on the words "click here".

But there is a more subtle issue as well. For a link to be deferred on publishing, it must be possible to contextualize the link in the deferred location. In other words, when the deflection point occurs inline in a paragraph the reader should be able to infer where the link will lead from the paragraph and from the text the link is applied to. But lifting the same link text out of the paragraph and putting it somewhere else is not guaranteed to provide the same context.

For example, a link marked up like this is hard to defer algorithmically:

```
<p>In Rio Bravo,  
<xref href="https://en.wikipedia.org/wiki/John_Wayne">  
The Duke</xref>  
plays an ex-Union colonel out for revenge.</p>
```

We could generate a list of links and insert it later in the document. It might look like this:

```
<p>For more information, see:</p>  
  
<ul>  
    <li><a href="https://en.wikipedia.org/wiki/John_Wayne">  
        The Duke</a></li>  
    ...  
</ul>
```

But will it be clear out of the context of the original text what the words "the Duke" refer to? (The answer here is maybe, but it is not hard to imagine cases where it would be a definite no.)

On the other hand, if the deflection point is marked up in the subject domain like this:

```
<p>In <movie>Rio Bravo</movie>,  
<actor name="John Wayne">The Duke</actor> plays  
an ex-Union colonel out for revenge.</p>
```

Then, given that we know what the subject of the deflection point is, we could use it to create a list of links that are categorized by type and use the real names of actors even when the original text use a nickname:

```
<p>For more information, see:</p>

<ul>
  <li>Actors:
    <ul>
      <li>
        <a href="https://en.wikipedia.org/wiki/John_Wayne">
          John Wayne
        </a>
      </li>
      ...
    </ul>
  </li>
  <li>Movies:
    <ul>
      <li>
        <a href="https://en.wikipedia.org/wiki/Rio_Bravo_(film)">
          Rio Bravo
        </a>
      </li>
      ...
    </ul>
  </li>
</ul>
```

In short, algorithmically deferring document domain links is always tricky, but we can comfortably defer linking of subject annotations if we want to.

Different domain, different algorithm

What the linking algorithm illustrates perhaps better than any other is that the movement from one domain to another represents significantly different partitioning of content complexity, which means that it changes the structured writing algorithms in fundamental ways. While the algorithm has the same purpose in each domain, the way it achieves that purpose can be significantly different.

Structured writing algorithms always start with the content structures. How you design the content structures – the way the author records the content – determines everything you can do with the content. You create content structures to support algorithms. You create algorithms to improve content quality or streamline content management and

publishing. You then choose the content structures that support the algorithm you want to implement.

In the document domain, the data structures tend to have a one to one correspondence with their algorithms. As system designers determine they need a particular algorithm, they create structures to support that algorithm. Thus document domain languages that require support for linking, reuse, indexing, and single sourcing have separate data structures for linking, for reuse, for indexing, and for single sourcing. (Some of these may be management domain structures, of course.)

In the subject domain, though, the data structures reflect the subject matter. If you go looking for a one-to-one correspondence between a structure and the algorithm it supports, you won't find it. Thus you will not find link markup or reuse markup or index markup or single sourcing markup in the subject domain. You will find markup that clarifies and delineates the subject matter of the content it contains. Any algorithm we want to apply has to interpret that subject domain annotation and use it as the basis for creating whatever kind of document or media domain structure you want for publishing.

System designers do still have to think about what algorithms they want to apply, but that is to make sure that the aspects of the subject matter needed to drive the algorithms are captured. Since every subject structure can potentially drive many publishing algorithms, however, you will often find your subject domain content already supports any new algorithms you want to apply. This helps future proof your content.

Moving from the document domain to the subject domain is not a matter of asking what the subject domain equivalent of a document domain structure is, therefore, but a matter of asking what information in the subject domain drives the creation of document domain structures. Subject domain content can look very different from its document domain counterpart and will often be starkly simpler and easier to understand. This represents a better partitioning of complexity. But as noted before, all complexity has to go somewhere, and the use of the subject domain tends the transfer more of the complexity to the information architect or content engineer.

Chapter 16. Conformance

Structured writing uses constraints to partition and redirect complexity. It is constraints on composition that enable us to partition the complexity of content creation. Constraints on the interpretation of content allow us to redirect complexity to algorithms rather than people. The success of our effort to partition and redistribute complexity, therefore, depend on everyone's conformance to the applicable constraints. Every failure to conform is a piece of complexity going unhandled, and since complexity cannot be destroyed, it falls through to a downstream process and ultimately to the reader. Conformance, therefore, is a key component of any structured writing system.

But conformance is a complex problem in its own right. We can't expect to be successful if we just make up structures and systems and demand conformance to their constraints without any thought as to how conformance is to be specified and assessed. It is perfectly possible (and all too common) to invent systems that would operate flawlessly if everyone conformed, but which never work well because their constraints are impossible to conform to in the real world of work.

Constraints have always been part of the authoring process. Style guides and grammatical reference works express constraints that writers are expected to follow. Editorial guidelines tell writers what kind of content a publisher is looking for, at what length, and in what format. If a publisher says that manuscripts must be delivered in DocBook or Word format, that is a constraint. When the government says that you must submit your online tax return in a particular file format, that is a constraint.

Some of these constraints are merely statements of requirements. Authors are not given any assistance in following them nor is there any verification mechanism to tell them if they have followed them or not (other than perhaps an email from an irate editor). Others are highly mechanical. Good tax preparation software will guide you all the way in filling out your tax forms and will run all kinds of checks to make sure that you did it correctly. It will also factor out many of the complexities of the tax code and ask you for information in a way you can understand, thus making it easier for you to conform.

This higher level of conformance checking and support helps make the process easier and the results more reliable. Unless the data passed from one partition to another is reliable, partitioning breaks down and complexity falls through. The conformance algorithm is the linchpin of structured writing. Without it, none of the other algorithms will work reliably.

How many constraints you need to place on your content depends on your quality and process goals – how and where you want to partition and distribute complexity in your organization. The larger your content set becomes, and the more critical content quality is to your business, the more frequent and dynamic your outputs are, and the more of your processes rely on algorithms, the more constraints you need and the more pressing the issue of conformance becomes. Content reuse, for example, relies on conformance

to the constraints for writing content that fits when reused, and on conformance to the constraints on what can be reused where. If you want to do any kind of real-time publishing, meaning there is no time to do quality assurance on the output of the algorithm, then reliable content is key, and conformance is how you ensure that content is reliable.

One of the ways in which a structured writing project can get into trouble is by introducing new constraints to meet management or publishing automation goals without considering how conformance to those constraints will be achieved. In some cases, this results in a highly imperative approach to conformance, in which writers are trained to implement the constraints, but where the structures they are creating provide no guidance or validation of those constraints. The system constraints, in other words, are not reflected in the content structures. If you are creating complex structures and also creating complex constraints that are not reflected or implemented in those structures, you have dumped a huge amount of complexity on your writers and you are going to have a twofold conformance problem: conformance will be expensive, and it will be inconsistent.

The best way to ensure conformance with a constraint is to factor out the constraint. If the constraint is that the titles of works cited should always be formatted in a certain way, you can factor out this constraint by moving to the document domain and using something like DocBook's `citetitle` element to mark up the titles of works. Now it is the publishing algorithm that is responsible for conforming to the formatting constraint. The formatting constraint has been factored out of the content.

When we move content creation from the media domain to the document domain we are factoring out all of the formatting constraints of the document. When we move content from the document domain to the subject domain we are factoring out many of the document design or management constraints and enforcing a number of constraints on the information captured.

But while this factors out one set of constraints, it also creates a new set of constraint in the new domain. When we factored out the formatting constraint for the titles of works cited, we introduced a new constraint, which is to markup the title of works using `citetitle`. Factoring one constraint into another is useful if it makes the constraint easier to conform to or easier to validate or it captures additional data that enables other algorithms. A constraint may be easier to conform to if it is simpler, easier to remember, or does not require knowledge that is outside the writer's field. For instance, the `citetitle` tag is a single tag, not a set of formatting instructions, and the author knows when they are citing the title of a work. A constraint may be easier to validate if it has fewer components or can be limited to a narrower scope. For instance, the set of things that are titles of works is smaller than the set of things that are formatted in italic, so validating the `citetitle` constraint requires looking at a smaller and more homogeneous set.

If the constraint you are introducing is not easier to conform to or easier to validate, you probably should think twice before you introduce it. There are certainly cases where

moving content to a more formal document domain model introduces more constraints than it eliminates without making those constraints easier to comply with or validate. On the other hand, sometimes those additional constraints are required to reduce complexity somewhere else in the process, or to manage previously unmanaged complexity. In other words, you are transferring complexity to writers from somewhere else in the content system.

Inevitably, writers have to make some concessions to the needs of algorithms, but we don't want algorithms to impose such a burden on authors that we distract them from doing good research and quality writing. Any complexity that writers can't handle gets dumped on the reader. So if the structures you create for the sake of algorithms prove too complex for authors to conform to, or too difficult to validate, you should ask if you could refactor those constraints again, perhaps to the subject domain, so that you get both the precision and detail that the algorithm needs and the ease of use and validation that the writer needs. In other words, don't address one source of complexity in isolation. Keep moving complexity until every piece of it is handled by a person or process with skills, bandwidth, and resources to handle it.

The recipe examples we have been using shows how a subject-domain structure can transfer complexity away from writers while providing for a high level of conformance checking.

```
recipe: Hard Boiled Egg
introduction:
    A hard boiled egg is simple and nutritious.
ingredients:: ingredient, quantity
    eggs, 12
    water, 2qt
preparation:
    1. Place eggs in pan and cover with water.
    2. Bring water to a boil.
    3. Remove from heat and cover for 12 minutes.
    4. Place eggs in cold water to stop cooking.
    5. Peel and serve.
prep-time: 15 minutes
serves: 6
```

Pulling the prep-time and serving numbers into separate fields makes it easy to validate that the writer has conformed to the constraint to include this information while also making it easier for the writer to supply this information. Putting the ingredients into a record set rather than a list or table factors out any presentation constraint entirely, making it impossible not to comply. And again, it is easier for the writer to create the content in this form than it is to create a presentation-level table, for instance.

Completeness

Completeness is an obvious aspect of content quality. Unfortunately, lack of completeness is often hard for writers and reviewers to spot. The curse of knowledge means that omission of information is hard to see unless there is an obvious hole in a predefined and explicit document structure. Defining structures that encapsulate information requirements can significantly improve completeness. In Chapter 5, *Writing in the Subject Domain* we saw how calling out the preparation time and number of servings for a recipe helps ensure that the author always remembers to include that information, whether or not we decide to present it in fields or as part of a paragraph.

But this is not the only way that subject-domain structured writing helps ensure completeness. Every subject-domain annotation highlights a subject that is important to your business. You can use an algorithm to scan those annotation and build a list of subjects that are important to your business. You can use this list to make sure that all the subjects you need to cover are actually covered.

For example, structured writing allows you to annotate your text to call out the types of certain phrases such as function names, feature names, or stock symbols.

When installing widgets, use a {left-handed widget wrench} (tool) to tighten them to the recommended torque for your device.

This sample annotates the phrase “left-handed widget wrench” and records that these words describe a tool. If all mentions of tools are annotated this way, you can then compile a list of all the tools you mention in all your topics and make sure that you have suitable documentation for each of them.

Consistency

Similarly to completeness, consistency can make a big difference to readers, but lack of consistency is hard to spot if the structure of the content is not explicit in all the ways we want it to be consistent.

Being consistent simply means abiding by constraints. As we have seen, this can take the form of either enforcing the constraint through required structure or, preferably, factoring out the constraint so that it is handled by algorithms rather than people. We have looked at how you can factor out constraints in both the document domain and the subject domain.

If you annotate the important things in your content set, such as tools in the example above, you can use the annotations to check for consistency of names. Suppose a writer accidentally writes “spanner” rather than “wrench” in the name of the tool:

When installing widgets, use a {left-handed widget spanner}(tool) to tighten them to the recommended torque for your device.

Since you can now generate a list of tools mentioned in the text, you can check each mention against a list of approved tool names. This can reveal both incorrect names (consistency) and tools that may be missing from the official list (completeness).

The same would apply to values in fields, such as the wine match field in the recipe example. With the wine match in a separate field you can compile a list of wines mentioned or check each mention against an approved list. More on this in Chapter 36, *Terminology*.

Accuracy

Accuracy problems are often hard to spot. Typos, using old names for things, or giving deprecated examples are all hard for writers and reviewers to see. But there are structured writing techniques than can catch many of these kinds of problems.

For example, if you were documenting an API, you could annotate each mention of a function.

Always check the return value of {rotateWidget()}(function) to ensure the correct orientation was achieved.

API function names can be quite tricky to remember sometimes and small typos can be difficult to spot. But with this annotation, you can validate all mentions of functions against the API reference or the code base. This technique not only catches misspellings. I have seen it catch the use of deprecated functions in examples, for instance.

Semantic constraints

We can usefully divide constraints into two types: structural constraints and semantic constraints. Structural constraints deal with the the relationship of various text structures. Semantic constraints deal with the meaning of the content.

For instance, consider this structure:

```
<person>
    <name>John Smith</name>
    <age>middle</age>
    <date-of-birth>Christmas Day</date-of-birth>
</person>
```

Some people certainly describe themselves as middle aged, and Christmas Day is certainly a date of birth, if an incomplete one. The author has complied with the structure of the document. But the creator of this markup language was probably looking for more precise information, probably in a format that an algorithm could read. What they wanted was:

```
<person>
    <name>John Smith</name>
    <age>46</age>
    <date-of-birth>1970-12-25</date-of-birth>
</person>
```

Some schema languages (a concept I will explain in Chapter 42, *Constraint Languages*), such as XML Schema, let you specify the data type¹ of an element. You could specify that the type of the age field must be whole number between 0 and 150 and that the date-of-birth field must be a recognizable date format. Here's what the XML schema for these constraints might look like:

```
<xss:schema
  xmlns:xss="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

  <xss:element name="person">
    <xss:complexType>
      <xss:sequence>
        <xss:element name="name" type="xss:string"/>
        <xss:element name="age" type="age-range"/>
        <xss:element name="date-of-birth" type="xss:date"/>
      </xss:sequence>
    </xss:complexType>
  </xss:element>

  <xss:simpleType name="age-range">
    <xss:restriction base="xss:int">
      <xss:minInclusive value="0"/>
      <xss:maxInclusive value="150"/>
    </xss:restriction>
  </xss:simpleType>
```

¹The data types referred to in the example above are not data types as they are commonly understood in programming terms (which refers to how they are stored in memory). In XML, as in all major markup languages, the data is all strings. A data type in a schema is actually just a pattern. There is a language for describing patterns in text that is called regular expressions. Regular expressions are a bit cryptic and take some getting used to but they are incredibly powerful at describing patterns in text. XML schema lets you define types for elements using regular expressions, so there is a huge amount you can do to constrain the content of elements in your documents.

```
</xs:schema>
```

This schema uses the built in types `xs:string` and `xs:date` for the name and `date-of-birth` elements and defines a new type called `age-range` for the `age` element. Using this schema, the example above would now fail to validate with type errors reported for the `age` and `date-of-birth` fields.

Applying these kinds of semantic constraints to content is not going to work if most of your text is in free-form paragraphs. It is hard to define useful patterns for long passages of text. If you want to exercise fine-grained control over your content, therefore, you must first break information down into individual fields and then apply type constraints to those fields.

In some cases, we create text structures purely for the purpose of being able to apply semantic constraints to their content. We use structural constraints to isolate semantic constraints so that they are testable and enforceable.

This can be particularly effective when you are creating content in the subject domain since you don't have to specify information in sentences, even if you intend to publish it that way. You can break the content out into separate structures and define the data type of those structures to ensure you get complete and accurate information, and to ensure that you can operate on that information using algorithms.

The recipe text that we have used before is a good example of how content that could be expressed entirely in free-form paragraphs can be broken down in a fine-grained way that allows us to impose a variety of structural and semantic constraints.

```
recipe: Hard Boiled Egg
introduction:
    A hard boiled egg is simple and nutritious.
ingredients:: ingredient, quantity
    eggs, 12
    water, 2qt
preparation:
    1. Place eggs in pan and cover with water.
    2. Bring water to a boil.
    3. Remove from heat and cover for 12 minutes.
    4. Place eggs in cold water to stop cooking.
    5. Peel and serve.
prep-time: 15 minutes
serves: 6
wine-match: champagne and orange juice
beverage-match: orange juice
nutrition:
    serving: 1 large (50 g)
```

```
calories: 78
total-fat: 5 g
saturated-fat: 0.7 g
polyunsaturated-fat: 0.7 g
monounsaturated-fat: 2 g
cholesterol: 186.5 mg
sodium: 62 mg
potassium: 63 mg
total-carbohydrate: 0.6 g
dietary-fiber: 0 g
sugar: 0.6 g
protein: 6 g
```

This entire recipe could be presented free form. But when structured like this we can enforce detailed constraints like ensuring that there is always a wine match listed or calories are always given as a whole number. The publishing algorithms could actually stitch all this content into paragraphs again if that was how you wanted to publish it. But when it is created in this format the author is provided with a huge amount of guidance about the information you want, and you are able to manipulate and publish the content in many different ways. Readers will benefit because all the recipes will conform to what you know readers need and want in a recipe.

Entry validation constraints

The ability of algorithms to read the data in your structures can have another conformance benefit because it allows you to check one piece of information against another. For instance, if you have date of birth and age you can calculate current age from the date of birth and compare it against the value of the age field. If the values don't match, you know the author made an error and you can report it. Here is a Schematron assertion that tests this constraint (in a slightly imprecise fashion: date arithmetic is surprisingly hard!):

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
queryBinding="xslt2">
  <pattern>
    <title>Age constraint</title>
    <rule context="person">
      <assert test="age =
        xs:int(days-from-duration(current-date() -
        xs:date('1970-12-25')) div 365.25)">
        Age does not match the given date-of-birth.
      </assert>
    </rule>
  </pattern>
</schema>
```

Referential integrity constraints

In the management domain, there are a set of constraints that we could call referential integrity constraints. Referential integrity simply means that if you make a reference to something, that thing should exist. In the management domain, we often give IDs to structures and use those IDs to refer to those structures for purposes such as content reuse.

If you are going to reuse a piece of content by referring to its ID, there is an obvious constraint that a piece of content with that ID must exist. This constraint is important enough that the XML specification actually builds in direct support for it (as does SAM). XML itself requires that if an attribute is defined as having the type IDREF, then there must be an element with an attribute of type ID with the same value in the same document. This can be useful for checking things like a footnote reference in a document actually corresponds to a footnote somewhere in the document.

Many management domain algorithms, however, require referential integrity not only within a single document but between documents. The conformance of a document to these referential integrity constraints can sometimes only be judged by the publishing algorithm when it is published in a particular combination. In fact, it is possible for a document to have referential integrity when published in one collection and to lack it when published in another.

Since it is always better to validate a constraint as early as possible, a content management system that is aware of the referential integrity constraints of a system (such as a DITA CMS, for example) may validate the referential integrity of content in all its potential combinations without the need to actually publish it.

Nevertheless, referential integrity constraints of this complexity still present a management and authoring headache, even with content management system support. It is worth considering if there is a way to factor them out. One example of how this can be done is found in the various approaches to the linking algorithm (see Chapter 15, *Linking*).

Conformance to external sources

Referential integrity constraints can span multiple documents. So can semantic constraints. For example, you may want values in your document to match values in databases or in other documents. A technical writer documenting an API may produce an API reference, much of which may be extracted from the program source code (see Chapter 19, *Extract*), and also a programmer's guide, which they write from scratch. The programmer's guide will obviously mention the functions in the API many times. There is the possibility that the writer may misspell one of the names, or that the API may be changed after parts of the document are written, or that a function the writer has mentioned no longer exists.

It is clearly a semantic constraint on the programmer's guide that all the API calls it mentions should actually be present in the API. Since the API reference is generated from the source code, we can express this constraint as: functions mentioned in the programmers guide must be listed in the API reference.

This is an important constraint. When we implemented algorithmic support for this constraint on one project I worked on, it revealed a number of errors:

- Misspelled function names in the programmer's guide.
- The inclusion in the programmer's guide of material related to a private API that was never released to the public.
- The failure of the API guide to include an important section of the API due to incorrect markup in the source code.
- A section of the programmer's guide that discussed how to do things using a deprecated API and failed to discuss how to do them with the new API.

All these errors were present in the programmer's guide despite several thorough reviews by multiple people over multiple software release. These are all the kinds of errors that human beings have a hard time spotting in review. But they all have significant impact on users who are trying to actually use the API.

As part of the conformance algorithm for the programmer's guide, we added a check that looked up each reference to an API call, including those in code blocks, in the source files for the API reference and reported an error if they did not match. None of the errors listed above would have been detected without this check.

Of course, for this check to be possible, the algorithm that did the checking had to be able to identify every time the programmer's guide mentioned an API call, and it had to be able to find all the API call names in the API reference. For this to be possible, both documents had to be written in a structured format that made the function names accessible to the algorithm. Here is a simplified example. First, a code sample from a programmer's guide:

code-sample: Hello World

The Hello World sample uses the `{print}()` function to output the text "Hello World"

```
```(python)
print("Hello World.")
```

Next, a function reference listing from the API reference:

```
function:
 name: print
 return-value: none
parameters:
 parameter: string
 required: yes
 description:
 The string to print.
 parameter: end
 required: no
 default: '\n'
 description:
 The characters to output after the
 {string}(parameter).
```

Because the API reference labels “print” as a function name and the code-sample annotates “print” as the name of a function, we can look up “print” in the API reference to validate the annotated text in the programmer’s guide.

By adding these structures and annotations to the content, we isolated the semantics of the function call names so that we could apply semantic conformance checks to them.

## Conformance and change

Requiring conformance to outside sources means that a document’s conformance is neither static nor absolute. A document that was conforming may stop being conforming because of outside events. But this reflects reality. One of the most complex aspects of content management, in fact, is detecting when a document ceases to be conforming because of a change in the reality that it describes. Using structured writing techniques to validate the conformance of a document against an external source can go a long way to addressing this class of problem. For more on change management, see Chapter 22, *Change management*.

## Design for conformance

Mechanical constraints can do a lot to ensure and validate conformance. But conformance is fundamentally a human activity and we may need humans to conform to constraints that we cannot easily express or validate in purely mechanical terms.

Schemas and downstream algorithms can verify that a writer created the required mechanical structures, and, in some cases, that they created them with content in the required form. But in most cases there is little or nothing that they can do to verify that the content of those structures is actually what the structure says it is. Unless the author

understands what the structure is for, they are not likely to create content that actually obeys the constraints that the structure expresses.

If the author is not on-board with creating content according to the structure, there is nothing the mechanical conformance can do to ensure that they did not just write it the way they wanted to while wrapping the required structured tags around it to make it pass the validation tests.

If there are optional elements to a structure, there is nothing that mechanical validation can do to determine if the optional parts of the structure were included when they should have been or omitted when they should have been. Only the author can know this, and only if they understand the purpose of those structures and are on board with them as the appropriate structure for what they are writing.

If the structures are difficult for the writer to understand (if they are complex management domain structures that require complex abstract IDs, for instance) then the author has to think about issues outside of their domain of expertise. Complexity has been dumped on them that they are ill-equipped to handle. They have to stop thinking about writing and their subject matter and start thinking about the system. Since most writers don't really understand how complex structured writing systems work internally, they tend to do the simplest thing they can think of that makes the system stop throwing errors and accept their work. This often produces dumb compliance to mechanical system constraints rather than intelligent conformance to content constraints. And this dumb compliance essentially dumps the complexity that the writer could not handle on the reader or another part of the organization.

The real key to achieving conformance is to create structures that are easy to conform to. For most content, conformance is not about trying to catch evil doers. The authors are on side and trying to produce good content. Authors who understand structured content may impose constraints as an aid to their own work, just as a carpenter, for instance, might design a jig to guide their saw. Constraints are a tool for writers, not a defense against them. Constraints may force a lazy writer to pull up their socks and do some more research. They may force an inattentive writer to recast their first draft into a more consistent format. But they should never prevent a good and diligent writer from doing good work.

The real core of compliance in structured writing, therefore, is not enforcement, but creating structures that:

- Are clearly and specifically appropriate to the subject matter and the audience being addressed.
- Clearly and specifically supply the information that the reader needs to accomplish a clear and specific goal.
- Clearly communicate to authors what is expected of them in terms they understand.

- Either remind authors of what is required or (preferably) factor it out.

Writers may disagree, of course, about what goals should be addressed, what information readers need to achieve those goals, or how best to express information to that audience. Where we have many authors contributing to a common information set, it is important that these differences be addressed and resolved professionally and that all authors involved be on board with the plan going forward. It is at that point that well-thought-out content structures can capture the decisions made and make sure that everyone stays on track and is consistent.

For example, suppose we have a set of cooks contributing to a common recipe information set. Everyone gets on board with the principle that every recipe should include the preparation time and the number of servings. At that point, creating a recipe structure that explicitly calls out those fields helps all the cooks to remember, and therefore comply with, the agreement they have made. This will significantly improve compliance compared to merely stating the requirement in the style guide. (And, of course, it will make that information available to other algorithms as well, reducing the complexity of assembling various documents from a common set of recipes.)

Auditing and enforcement still have a role to play, not because authors are hostile to the system, but because they are human. But auditing and enforcement are secondary to the main aim of conformance-friendly design. And in that spirit, auditing and conformance should be seen as part of a feedback loop that is constantly seeking to improve the design. If you keep finding the same mistakes over and over again, that is not a training problem or a human resources problem, it is a design problem.

Finally, be aware of how your authoring interface affects conformance. How do you know if you are meeting constraints – in any activity? Feedback. With any activity, we need a way to know when we are done and when the work is correct. In the media domain, there is one form of feedback: how the document looks. With a true WYSIWYG display, if it looks right on the screen, it will render correctly on paper, or whatever media you are targeting as you write. That is the writer's signal that they are done, and done correctly. That's fine in the media domain. But if you are trying to create content in the document domain but giving your writers a media domain display (thanks to a structured editor that is trying to look as much as possible like Microsoft Word) then the feedback they receive will be media domain feedback, and the conformance you will get will be media domain conformance. There is a very real risk that content created this way will not work as intended when output to other media, or reused, or processed with any of the other structured writing domains. Don't expect to get real conformance to any structure that is hidden from the author. No one can conform to a structure they can't see.

---

# Chapter 17. Auditing

If the conformance algorithm is about making sure that an individual item meets its constraints, the audit algorithm is about making sure that the content set as a whole meets its constraints.<sup>1</sup> It is not a given that if every item in the set meets its individual constraints that the whole collection meets its constraints. Issues of completeness and referential integrity arise between the items in a collection and are a considerable source of complexity that affects every structured writing algorithm that deals with the relationship between different pieces of content.

As such, auditing is fundamentally a content management function. It is about making sure that:

- The definition of the content set is correct (we know what types of content it should contain, and which instances of each type)
- The content set is complete (it expresses all the items of each type that it should)
- The content set is uncontaminated (it does not contain any items or types it should not)
- The content set is integrated (it contains all of the relationships between items that it should)
- Each item in the content set conforms to its constraints

Auditing a large content set is difficult and many CMS solutions are deficient in audit capabilities. The main reason for this is that with the way most content is recorded and stored (media domain or generic document domain formats), it is very difficult to mechanically assess what content you have and what state it is in. It is hard to know if you have all the pieces you should have if you can't tell exactly what each piece is.

One of the biggest, and least appreciated, benefits of structured writing is that it makes content more auditable. When content management systems fail or become unmanageable, the root cause is often either an incorrect distribution of complexity from day one or a failure to audit: either lack of attention to regular audits, or the lack of ability to audit effectively. Without the ability to audit effectively, content sets often end up incomplete, corrupt, and poorly integrated, which reduces quality and increases costs at every stage of the process. It creates a body of unmanaged complexity that every downstream process and person has to deal with, including, of course, the reader. And a vicious cycle can develop in which writers, frustrated with the difficulties of the system, create workarounds that further corrupt the information set. Unmanaged

---

<sup>1</sup>Content strategists often use the term “content audit” to mean a current state analysis performed at the beginning of a website redevelopment project. A content strategy content audit is about cataloging, and possibly categorizing, the content you already have. I am using the word audit to refer to an ongoing and or recurring activity in which a you ensure that a content set is meeting or continuing to meet its goals.

complexity breeds more complexity. Whatever expenses you may incur to implement a more structured writing approach could well be offset by the savings associated with more effective auditing of the content set.

## Correctness of the definition of the content set

Content strategists will spend a great deal of time and effort developing a content plan (usually this is for a website, but the same principle applies to any content set). How they do this is beyond the scope of this book, but the result should be a definition of the content set: which types of information it is supposed to contain and what instances of those types. (This definition is based, of course on the goals it is designed to achieve, which it is the business of the content strategist to define.)

The definition of a content set is not necessarily static. It is not necessarily a fixed list of topic types or of specific topics to be developed. For one thing, the subject matter may change during the course of content development, which would change the content pieces needed, and perhaps require new content types or modifications to existing types. Second, the exact set of pieces or types may not be knowable at the outset. Content development explores a complex set of relationships between subject matter and the needs and background of the reader that cannot be fully known without traveling the ground in detail.

It is hard to be disciplined and deliberate in evolving the big picture model of the content set if you are not disciplined and deliberate in how you create the pieces. If you ask a writer to write a piece and then ask them after they are finished to describe the job it does by assigning CMS metadata to it, they will tag it using the terms that seem like the closest fit to the content they have already written, but they will probably not revise the content to fit the labels they are applying to it. Their view will not be that the content is wrong, but that the labels don't fit the content. And indeed, since the labels won't fit the content, you won't really know what type of content you have in your collection.

If you don't really know what type of content you have, you can't really tell if the definitions for your content are correct. Some content may perform poorly because it does not fit the type definition properly. But you can't tell whether it failed because it didn't fit the type definition or because the type definition is wrong. Thus you don't know what to fix.

Of course, a writer may come up with something that is better than the current type definition. This is a good thing. If it really is better, you want to change the type definition to match it so that all future content will be better as well. But if the author simply tags it according to the current CMS tags, you will never know that it is a different model, never have the chance to test the new model to see if it is better, and never have the chance to update the definition so that new content follows the successful new model. Unless your

content types are codified and auditable, you won't detect improvements in the types and they won't carry over to other content. See Chapter 26, *Repeatability* for more on this.

Having strong well defined content types makes it easier to audit your types to make sure they are doing the job they were designed to do. Similarly, having strong well defined content types means that you can have greater assurance that each item is doing the job it is supposed to do, which helps you make sure you have covered all the subjects you should have.

## Bottom-up content planning

But structured writing can do more than this to help you audit the definition of the content set. If you create content in the subject domain, including annotating the subjects that you mention in the text, you can use algorithms to extract a list of the types and subjects that your content is actually talking about. In your initial top-down plan, you may not have thought about the need for content on a certain subject or to support a certain activity, but if that subject or that activity start showing up in the body of your content, that is a strong indication that those subjects and activities are related to the purpose of your content set and should probably be included in the definition of the content set.

Subject domain structured writing is how you know what your content is actually talking about, what every author is discovering or thinks needs saying. Content needs are ultimately driven by subject matter and it is your writers working with the subject matter every day who are on top of what the subject matter is and how it is changing. Bottom-up content planning distributes the responsibility for discovery outward and for coordination inward to keep you in touch with evolving content needs. Without this information flow it is very difficult to establish that the content set is meeting its coverage goals. (We will see the same pattern of information flow again in Chapter 36, *Terminology*.)

This actually attacks two audit problems. If writers are writing about things outside your current coverage definition, either your coverage definition needs updating, or writers are polluting the content set with irrelevant material.

## Ensuring the content set remains uncontaminated

Subject domain content structures and annotations can help you prevent contamination of the content set by irrelevant material. But more important than catching writers in the act is catching the flaws in content types that allow for contamination to creep in.

A major form of contamination in any content set is redundant content. As we noted in Chapter 14, *Normalization*, we have to be careful in how we define redundancy, because it is not simply a matter of only addressing a subject once. It is a matter of addressing an audience need only once, and that may require several topics on the same subject

addressed to different readers. But it is all too easy for duplicate content to sneak into a content set. Some of it comes in because the same functionality is repeated in many products or in content delivered to different media. Some comes in through authors simply not knowing that suitable content already exists.

Content reuse is a major motivator for structured writing for exactly this reason. But the content reuse algorithm only addresses the problem of how to reuse content. It provides a method to reuse content you are aware of. It does not prevent you from duplicating content because you did not look for or did not find existing reusable content. You need to audit your content regularly to make sure that content is findable by writers who may want to reuse it, and to make sure that duplication is not creeping in.

There are natural language processing algorithms that will attempt to identify redundant content in a content set, but such algorithms focus on similar texts. This is not enough. The same or similar sequences of words may occur in different places without being redundant. They may mean different things, or perform different roles, in context. On the other hand, redundant pieces of information may be expressed in very different words. It is redundant information, not redundant phrases, the we care about.

Even when redundancies are found, they may be very difficult to consolidate if they don't have similar boundaries within their respective documents (the composability problem – see Chapter 13, *Composition*). Strongly typed subject domain content, meaning content that conforms to a model that breaks down and enforces the various pieces of information required to correctly cover a topic, makes it possible to detect duplication in a much more formal and precise way. Duplication of subject matter is much easier to detect when content is captured in the subject domain.

A person who consults a repository to see if there is a piece of content they can use relies on the ability to query the repository in a sensible way for the type of content they are looking for. They also rely on their ability to recognize the content when they see it, and on it actually being strongly conformant so that they can use it with confidence. Subject domain topic typing helps with all of these things. Subject domain labeling of document and media domain content can help as well, but only if it is conformant, a problem discussed in Chapter 16, *Conformance*. The easier it is to correctly identify reusable content and use it, the less corruption of the repository will occur.

## Ensuring that the content set is well integrated

A content set is never a collection of wholly independent pieces. The items in the set have relationships to each other that matter to the reader. (We will look at this in more depth in Chapter 21, *Information Architecture*.) Whether you express those relationships through links or cross references, or whether you rely entirely on tables of contents and indexes, it is still important to understand and manage the relationships between items.

Relationships between items may matter for management reasons as well. If you have documentation for multiple releases of a product, the relationship between the documentation for feature X in version 3 and that for feature X in version 2 matters to you. It may matter because the feature has not changed and you can reuse the item. It may matter because an error was found in version 2 and you want to fix it in version 3 as well. And if you put this content online, the relationship may matter for the reader as well, if they search for feature X and get the result for version 2 when they are using version 3.

You can describe the relationship between items externally. Items are related whenever they have the same value in any one of their metadata fields. Which field it is tells you what the relationship is. Finding the relevant metadata field to look at allows you to manage the relationship. But the same problem exists here as it always does with all external metadata (see Chapter 23, *Content Management*) – the content may not conform to the metadata, and without structured writing in the content itself, it is hard to audit the conformance of the content to its metadata.

But the bigger problem is that external metadata does not map the important relationship that can exist between a part of one item and the whole of another item. Are function names mentioned in the programming topics all listed in the API reference? Are the utensils mentioned in a recipe all covered in the appendix of kitchen tools? These are important content relationship questions, but their importance cannot be mapped with external metadata. You need subject domain markup inside the piece that identifies function calls and the names of kitchen tools.

Structured writing, particularly in the subject domain, helps you discover and manage these relationships by making clear the subjects on which these relationships are based.

## Making content auditable

I have talked all through this chapter about how using strong content types makes content easier to audit. What is a strong content type? Fundamentally, a strong content type is one that makes explicit what the content is supposed to say and how it is supposed to say it. Or, to put it another way, a strong content type is one that captures, enforces, or factors out the major constraints of the content, including its major rhetorical constraints. A strong content type constrains the interpretation of content as well as its composition, and the more reliably content can be interpreted, the more reliably it can be audited. Strong content types are almost always in the subject domain.

It is possible for content to conform to all of its rhetorical constraints without the use of structured writing techniques. But strong content types provide explicit guidance to the author and facilitate the use of conformance algorithms. They are created to meet your conformance goals. Similarly with auditing, you specify the content structures you need in order to meet your auditing goals.

Auditing is sometimes not as straightforward as conformance, even with structured writing techniques in place. Auditing often requires human review, not only to make sure

that all subjects have been covered, but to discover new issues or subjects that need to be addressed. Human review of a large content set is difficult, though, due to the sheer amount of content. An audit algorithm can simplify this work by creating different views of the content set that humans can review more easily.

Suppose, for instance, that an organization is using subject-domain annotations to drive linking as described in Chapter 15, *Linking*. Every topic in the collection is supposed to be indexed to state the type and names of the subjects it covers. Every mention of a significant subject is supposed be annotated with its type. The linking algorithm uses these annotations and index entries to link the content without any need for authors to create or manage links in the source text. But that does not guarantee that all the right links get made. There could be errors in indexing or annotation that are impossible to detect when conformance testing individual topics.

We can use those same index entries and annotations to create audit reports for several purposes. These are some of the things we can do:

- We can create a sorted list of all the phrases that have been annotated and see if they are being annotated consistently. This will tell us a lot about the types we are using, how well they are understood, and what instances of each type we should be covering.
- We can create a list of all the phrases that have been indexed and check it against our content plan (perhaps against a taxonomy, if we have one). This will tell us a lot about whether our coverage is complete, whether our writers are getting off track, or whether our content plan or our taxonomy is off base with reality.
- We can create a sorted list of all the index terms and check it against the list of annotated phrases to find phrases are that being indexed but not annotated or annotated but not indexed. This can tell us is there are subject we are not covering, if writers are discussing subjects they should not be, or if some topics are not being indexed or annotated properly.
- During the content development phase, the list of things that are annotated but not indexed will inevitable grow, as subjects are being referred to before the content that describes them is written. The trend line of the growth of new subjects being annotated vs subjects being indexed will allow you to track how close a content set is to completion, even in cases were defining the boundaries in advance is difficult.

## Performance auditing

Auditing marketing content can be easier than auditing other types of content because you can measure content against behavioral goals. Are readers taking the actions you want, and does a content change produce more of the desired action? All content aims at changing reader behavior in one way or another, but not all behavioral changes are easy to measure. Unless the behavior in question is an interaction with the web page that contains the content, behavioral changes are both hard to observe and hard to attribute.

Whether auditing the performance of your content is easy or difficult, however, when you do have evidence that a particular piece of content is performing well, you want to be able to reproduce that success. I will talk more about repeatability in Chapter 26, *Repeatability*.

If a piece of content follows a well defined set of constraints, it is much easier to figure out why it was successful. Rather than trying A/B testing of individual pieces, for instance, you can try A/B testing of content that follows different structures. If one structure consistently outperforms the other, you can adopt that structure for all your content.

Of course, to ensure continues success (or to ensure that the same structure is continuing to perform well) you need to be sure that the new content you are producing does actually conform to the proven successful structure. This means that you need to apply the conformance algorithms to your new content creation, as well as auditing it for overall compliance and continued success.

---

# Chapter 18. Relevance

Establishing the relevance of content is essential for readers, and for many of the structured writing algorithms.

- Every reader who looks for content has to decide if a given document is relevant to their needs.
- A search engine needs to determine if a document is relevant to a reader's search.
- The reuse algorithm needs to determine if content relevant to a certain point exists or not.
- The composition algorithm needs to find all the pieces of content relevant to a certain purpose.
- The linking algorithm needs to find any documents relevant to a subject mentioned in the current document so it can create links to them.
- The conformance algorithm needs to determine if there is information in one document that should conform to information in another document.
- The quality algorithm needs to ensure that a document contains all the information relevant to the reader's needs.
- The information architecture algorithm needs to discover all the connections between pieces of content as well as their types and the constraints they obey in order to build a navigable information set.

The management of relevance therefore has a twofold role: the first is to alleviate as much as possible the complexity the reader faces in establishing relevance, and the second is to make sure that every algorithm that needs to establish the relevance of content are able to do so correctly and unambiguously.

There are two fundamental parts to the relevance algorithm: being relevant, and demonstrating your relevance. These correspond to obeying your constraints and recording the fact that you have done so – constraining the authoring of content and constraining the interpretation of content.

## Being relevant

All too often, making sure that content is relevant is neglected. CMS systems will attach metadata to unstructured documents asserting their relevance to certain topic or requirements without verifying that they actually meet those requirements. In many cases, the CMS has a cataloguing system for content that requires certain metadata fields

to be completed, often from a predetermined list of values. Developing a good set of relevant criteria and values is complex and the development effort is often conducted in a way that does not yield sound relevance criteria for users or readers. Taxonomy definition by committee often imposes an abstract logic that does not match the concerns or language of actual readers and users.

If developed correctly, this metadata may have the potential to define the relevance of content, but these requirements are not always communicated to the author before they write. Instead, writers are left to compose content without any guidance as to what would make it relevant, and then asked to tag it with the CMS metadata after the fact. In many cases, this tagging is done hastily and lazily, simply picking whatever terms seem appropriate at first glance without really thinking about whether the piece fulfills the promise the metadata is making.

The result is that the content returned by the CMS is often not relevant to the reader's needs because it simply does not possess the relevance that the CMS metadata claims it does – another case of dumping complexity on the reader. In many cases, bypassing the CMS's navigation mechanisms and doing a simple site search will produce better results, since the search engine looks at the content itself to determine what it is relevant to. But while this can rescue the reader's quest for relevant content where such content exists, it does nothing to ensure that relevant content actually gets created. Without this assurance, the CMS may be telling lies to its owners as well as to its readers.

The plain fact is that if you are going to implement strict standards of relevance, and a strict vocabulary for proclaiming relevance, this needs to be communicated to the writer long before they have finished writing and are trying to submit their content to the CMS. One way to do that is through structured writing.

## Showing relevance

Being relevant is important, it is equally important to show that you are relevant. Readers and algorithms can't find you if you don't show your relevance.

Actually, that is not entirely true. Search engines can determine the relevance of content with an amazing degree of accuracy simply by reading the text itself and looking at how the content is used and linked to. This attests to the first importance of being relevant.

But if the search engine decides a topic may be relevant and adds that topic to search results, but the reader then looks at it and can't see the relevance, you still don't reach the reader. (And the search engine may take note as well and downgrade that topic's relevance ranking.)

Showing your relevance to the reader is all about establishing your subject matter and context and doing it clearly and unambiguously in the few seconds that the reader is likely to look before dismissing you and moving on to the next item in their search results.

When developing a markup language (or other structured content container) for your content, one of the most important things to consider is how the reader will recognize what your documents are relevant to. This is about the rhetorical structure of each piece of content and how well it works to support the reader's recognition of the relevance of content . This is something I deal with at length in my book, *Every Page is Page One: Topic-based Writing for Technical Communication and the Web*

Relevance is not established the same way for all readers or for all subjects. For a recipe, for instance, a picture may do a lot to establish relevance. For an API reference, a version number and the description of a return value may be key relevance indicators.

But relevance is not just about the subject matter. It is also about the reader's purpose. For a page about a business, the inclusion of a stock price chart may tell you that the page is of interest to an investor rather than a potential customer. Placing that chart at the top of the page helps establish the relevance quickly. For an article about a place, pictures of beaches or nightlife show that the page is relevant to potential tourists, not to residents trying to decide what schools to send their kids to.

Creating content in the subject domain allows you to make sure that writers produce all the pieces of information that make a page relevant to the intended audience. Because the subject domain also factors out the order of presentation of content, it allows you to make sure that every page is organized in a way that best shows its relevance.

Better still, because subject domain content is organized by algorithms, you can experiment to see if one organization of content works better than another and adapt your presentation algorithm accordingly without having to edit any of the content.

Also, because storing content in the subject domain allows us to use the extract and merge algorithm to pull in content from other sources, we don't have to include the beach pictures or the stock chart with our subject domain content. Instead, we can have the synthesis algorithm query other content sets or feeds to find the best current tourist shots or to generate the stock chart in real time.

But that, of course, depends on the mechanical part of the relevance algorithm, because in order for the synthesis algorithm to do this it has to be able to find relevant pictures or a relevant stock chart. It is not enough to show relevance to readers, you also need to show relevance to algorithms.

## Showing relevance to algorithms

Showing relevance to algorithms comes down to breaking information up into clearly labeled fields containing clearly unambiguous values. For example, to query a web service that generates stock charts, you would need to provide it with a clearly unambiguous identifier for the company whose chart you wanted. The best way to do that would not be the company name, since there can be duplicate or similar names in different

industries (Apple Computer vs Apple Music, for example), but the company's stock symbol, which is guaranteed to be unique by the exchange on which it is listed (though you do have to provide the exchange code as well to be globally unique: NASDAQ:AAPL, not just AAPL).

This means that the web service needs to index its information on stock prices according to stock symbols (which is exactly what it does, of course). Actually, it is more than likely that the stock chart drawing service does not hold price information at all, but requests it as needed from yet another web service. The stock ticker symbol is the unambiguous key that identifies the company in each of these transactions.

While humans and algorithms assess relevance in different ways, the foundations of relevance are the same for both: clear identification of the type and subject matter of the information. This means that you need to maintain metadata that clearly identifies the type and subject, and also present the content in a way that makes its type and subject evident at a glance.

```
recipe: Hard Boiled Egg
introduction:
 A hard boiled egg is simple and nutritious.
picture:
 >>>(image egg.jpg)
ingredients:: ingredient, quantity
 eggs, 12
 water, 2qt
preparation:
 1. Place eggs in pan and cover with water.
 2. Bring water to a boil.
 3. Remove from heat and cover for 12 minutes.
 4. Place eggs in cold water to stop cooking.
 5. Peel and serve.
prep-time: 15 minutes
serves: 6
nutrition:
 serving: 1 large (50 g)
 calories: 78
```

This recipe example contains a number of structures that both humans and algorithms can use to assess relevance. The simple fact that this is specifically a document of type `recipe` establishes basic relevance. The inclusion of a picture helps humans visually identify the recipe they are looking for. Information like the number of calories could be used by an algorithm to select recipes for a low-calorie cookbook.

It is possible to maintain metadata on type and subject of content in a content management system that hosts content written in the document or subject domains. In this scenario,

ensuring that the organization of the content makes the type and subject evident at a glance is a separate problem from correctly labeling the content in the CMS. While it is certainly possible, with the right authoring discipline, to make sure that the content demonstrates what the metadata claims about it, it is also possible for the content not to match what the metadata claims, or not to demonstrate its type and subject effectively.

Moving content to the subject domain allows you to show relevance to both humans and machines with the same structures, assuring that the two do not get out of sync.

---

# **Chapter 19. Extract**

A great deal of the content we produce, particularly technical and business content, is essentially a report in human language on the specific features of a product, process, or data set. Much of the data that defines those things is contained in some kind of formal data set, such as a database or software source code.

In a traditional approach to publishing, a large amount of complexity and work is involved in researching and recreating the information in those systems in a content format, and then in trying to keep the resulting content in sync with the original source.

The source data for those systems is, in effect, subject domain content for those products, processes, and data sets. Rather than researching and recreating that content, we can use structured writing techniques to extract information from those sources to create and/or validate content. This partitions and redirects the complexity of dealing with these sources away from authors towards information architects and content engineers who create the algorithms to extract that data and transform it into content.

## **Tapping external sources of content**

I have talked throughout this book about moving content from the media domain to the document domain and from the document domain to the subject domain. We have seen the advantages that come from creating content in the subject domain, and we have looked at the processing algorithms that can use subject domain content to produce various kinds of publications in different media.

Subject domain content is simply content that is created and annotated in structures that are based on the subject matter rather than on the structure of documents or media. Subject domain structures tell you what the content is about, rather than how it should be published. You can therefore write algorithms that processes it based on what it is about rather than how it is presented.

Any data source that is contained and annotated in subject domain structures is therefore a source of subject domain content, whether it was intended to produce content from or not. This includes virtually all databases and quite a bit of software code. It also includes all authored content anywhere available (under an appropriate license) that contains any usable subject domain structures or annotations. All of this is potential material for generating content as part of your overall publishing process. As such, the extract algorithm can work effectively with many other structured writing algorithms.

Perhaps most obviously, extract works with the composition algorithm. Extract provides new sources of content for the composition algorithm to work with.

As a source of subject domain content, the extract algorithm also naturally separates content from formatting and contributes to the differential single sourcing algorithm.

By tapping existing information to build content, the extract algorithm also works hand in hand with the normalization and content reuse algorithms. In fact, it is really the highest expression of reuse, since it not only reuses content in the content system, but information from the organization at large, or even beyond the organization – a process that further reduces duplication within the organization.

Because the extract algorithm taps directly into external sources of information, it is also a great source of information for the conformance algorithm. At one level, it provides a canonical source of information to validate existing content against. At another level, it factors out part of the conformance problem from the authoring function. It transfers the entire responsibility for maintaining the information to the creators of the source you are extracting content from, which is a responsibility they already have. This is yet another example of the power of the subject domain to reuse its structures for many purposes, which illustrates just how much complexity is being partitioned and transferred to the algorithms created by the content engineer or information architect.

## Information created for other purposes

There is nothing new, of course, about generating content from database records. Database reporting is a highly important and sophisticated field in its own right and it would be entirely correct to characterize it as a type of structured writing. What sets it apart, largely, from other structured writing practices, is that the databases it reports on serve other business purposes besides being sources of content. An insurance company policy database, for instance, may be used to publish custom benefit booklets for plan participants but it is also used for processing claims. The design of the structures and data entry interfaces of these systems has tended to fall outside the realm (and the notice) of writers and authoring system designers.

This is a pity, because it has often resulted in organizations developing separate processes, tools, and repositories for content creation in which the information already contained in databases is researched, validated, recorded, and managed entirely independently on the content side of the house. Rather than treating code and databases as sources of content, writers treat them as research sources. They look information up in these sources and then go away and write content (in a separate repository) to describe the information from those sources.

The essence of the problem is that many content organizations choose to work in the media domain or the document domain and have neither the tools nor the expertise to bridge the gap to all this material already available in the subject domain. But even when content organizations do extend their efforts into the subject domain, they are often blind to the fact that the subject domain content they are proposing to create already exists in the systems of another department.

Another drawback is that content produced from these systems, for instance by a database reporting process, exists in isolation from the rest of the content produced by the

organization. Such content can often be quite sophisticated and beautifully formatted and published. But it is the product of an entire structured publishing chain that has to be separately developed and maintained.

In the field of software documentation we see the same pattern in regard to programming language API documentation. Much of the material of an API reference guide is a description of each function, what information is required as input (its parameters or arguments), the information it produces as output, and the errors or exceptions that it can generate. All of this information already exists in the code that implements the function.

API documentation tools such as JavaDoc or Sphinx extract this information from code and comments and turn it into publishable content. This is an application of subject-domain structured writing and the API documentation tools that do this implement an entire structured publishing system, producing final output, often in multiple formats.

And here we see all the same problems again:

- An entire publishing chain is maintained separately from the main content publishing chain.
- The content produced from this publishing chain is isolated from all the other content produced by the organization.
- Much of the same information is often created and maintained separately in a different repository and tool chain in the form programming guides and/or knowledge base articles.

There are other cases of entirely separate publishing chains producing information that is isolated from the rest of an organization's content. Technical support organizations create knowledge bases to answer commonly asked questions. The material in these knowledge bases is technical communication, plain and simple, yet it usually exists in isolation from the product documentation set, even to the extent that users may not be able to search both the documentation and the knowledge base from the same search box. Most users, however, have no way of guessing whether the answer they are looking for is going to be in the docs or the knowledge base (or in the users forum, often yet another independent publishing system).

There are a couple of ways to address these redundancies and the isolation that goes with them. One is to attempt to unify all content authoring and production in a single enterprise-wide system, often with a single set of content structures intended for use across all enterprise departments. However, this is a highly expensive and disruptive approach and tends to create interfaces and structures that are less usable and less specific to various business functions than the ones they replace. It ignores the local complexity of individual groups and subject matter, meaning that complexity gets shoved downstream, eventually to the reader. It also ignores the fact that many of the systems from which we wish to extract content exist for other purposes besides the content that is generated

from them. Their subject-domain structures are specific and necessary to the database functions or software code generation they were built for.

Another approach is to leave the subject domain systems in place (and perhaps create more of them) and feed their output into a common document-domain publishing tool chain. It is a normal part of the publishing algorithm for subject-domain content to pass through the document domain on its way to media-domain publication. Subject domain content, by its nature, is not strongly tied to a particular document domain structure, so integrating many sources of subject domain content into a single publishing chain is not particularly onerous. (Specific management-domain features of certain tool chains make things more complicated, but since the subject domain tends to factor out a lot of the management domain, this is not an insurmountable problem.)

Most enterprise-wide content systems are based on document-domain languages. (There is, after all, no way to create a single enterprise-wide subject-domain system, since an enterprise creates content on many subjects.) In principle, a document-domain system should be capable of integrating content from domain-specific subject-domain systems. Unfortunately, it is not common for either the subject domain systems or the enterprise content systems to be designed with this kind of integration in mind.

Because of this, we sometimes have to find ways to extract content from these sources and feed them into a unified publishing chain. This creates the need for extraction algorithms.

A common example of the extraction algorithm is found in API documentation tools such as JavaDoc. These tools parse application source code to pull out things like the names of functions, parameters, and return values, which it then uses to create the outline, at least, of reference documentation. Essentially, it generates a human language translation of what the computer language code is saying.

How the extraction algorithm works depends entirely on how the source data is structured, but it should usually create output in the subject domain that clearly labels the pieces of information it has culled from the source. For instance, a Java function definition is a piece of structured content in which the role and meaning of each element is known from the pattern and syntax of the Java language itself (its grammar):

```
boolean isValidMove(int theFromFile,
 int theFromRank,
 int theToFile,
 int theToRank) {
 // ...body
}
```

This same information can be extracted by an algorithm that knows the grammar of Java to produce something that looks more like subject domain content:

```
java-function:
 name: isValidMove
 return-type: boolean
 parameters:: type, name
 int, theFromFile
 int, theFromRank
 int, theToFile
 int, theToRank
```

This is the same information, but in a different structure. In this structure, however, it is easily accessible by content processes and can then be processed through the rest of the publishing tool chain just like any other content.

Any structured data source that expresses the semantics of its data in a consistent way is a source of subject domain content. We just need to find a way to get at it.

## The diversity of sources

When it comes to drawing content from diverse sources, the term single sourcing can mislead us. Single source can lead us to think that it means all source content is kept in a single place. Some vendors of content management systems would like to encourage this interpretation. But a better definition is that each piece of information comes from a single source. That is to say, each piece of information comes from only one source, but there may be many sources, each maintaining different information.

Ensuring that content is only stored once is actually about making sure that the information, and the repository in which it is stored, meets an appropriate set of constraints. The constraints that establish the uniqueness of a piece of content are different for different types of content and different subject matter, so such constraints are better expressed and enforced by systems that are specific to their particular subject matter.

Of course, there can often be trivial differences between the ways in which different bodies within an organization store and manage information that could and should be rationalized. There are all kinds of isolated and ad hoc information stores in most organizations that could potentially be much more efficient and much more accessible, with a degree of rationalization and centralization. But it does not follow at all that absolute centralization into a single system or a single data model is appropriate, useful, or even possible.

The best way to ensure that information is stored once is to have it stored in a system with the right constraints and the right processes for the people who create and manage that information. This may mean that an integrated publishing system will draw from diverse sources of information and content. The ability to extract content from these sources and to merge it with other content for publication is therefore central to an effective strategy.

Still, while this model sounds grand, it also introduces a lot of complexity in terms of its integration and maintenance requirements. The point of the exercise, we should always remember, is to minimize the amount of unmanaged complexity in the system. Integrated system can manage complexity in sophisticated ways, but they introduce their own kinds of complexity that has to be factored into the calculation. Sometimes the optimal solution is less than total integration.

---

# Chapter 20. Merge

If the extract algorithm gives us access to new sources of subject domain content, it does not always give us everything we need for a complete document. Sometimes we need to combine the content we have extracted with content we have written in order to present a complete set of information. This is the job for the merge algorithm.

The information that we can extract about an API function from the code that implements it is useful content, there is not enough detail to build a complete API reference. A good reference entry also requires some explanation of the purpose of the function, a little more detail on its parameters, and possibly a code sample illustrating its use.

To address this, we can merge authored content covering these topics with the content we have extracted from the source.

In the case of API documentation tools, the authored content for merging is often written in the source code files. It is contained in code comments and is often written in small subject-domain markup languages that are specific to that tool. (Though as with all subject-domain structures, any other tool can read them if it wants to.)

Here is an example of authored content combined with source code in JavaDoc<sup>1</sup>:

```
 /**
 * Validates a chess move.
 *
 * Use {@link #doMove(int theFromFile,
 * int theFromRank,
 * int theToFile,
 * int theToRank)} to move a piece.
 *
 * @param theFromFile file from which a piece is being moved
 * @param theFromRank rank from which a piece is being moved
 * @param theToFile file to which a piece is being moved
 * @param theToRank rank to which a piece is being moved
 * @return true if the move is valid, otherwise false
 */
boolean isValidMove(int theFromFile,
 int theFromRank,
 int theToFile,
 int theToRank) {
 // ...body
}
```

---

<sup>1</sup><https://en.wikipedia.org/wiki/Javadoc#Example>

In this example, everything between the opening `/*` and the closing `*/` is a comment (as far as Java itself is concerned), and the rest is a function definition in Java. However, JavaDoc sees the comment block as a block of structured text using a style of markup specific to JavaDoc.

The JavaDoc processor will extract information from the function definition itself (the extract algorithm) and then merge it with information from the authored structured content (the merge algorithm). In doing so, it has the chance to validate the authored content (the conformance algorithm), for instance by making sure that the names of parameters in the authored content match those in the function definition itself. This ability to validate authored content against extracted data is an important part of the conformance algorithm.

However, the merge algorithm does not require that the authored content be part of the same file as the data you will be extracting other information from. You can just as easily place the authored content in a separate file. In many cases, this will be the only available approach, since the source format will not provide any place to include documentation. All you need to be able to merge the two is an unambiguous term or combination of terms that you can find in the source data. You then enter that term or terms as a field or fields in the authored content where it can be used to match the authored content to the relevant extracted data.

For example, on one project I worked on, we needed to create a reference for a large body of operating system components. The components were defined in a data file that allowed for a one line description of the component which was intended for display in a GUI configuration editor. We needed to supply much more extensive documentation for each component, so we created an extract algorithm to pull information from the data file that defined the components and dumped it to XML. We then developed a subject-domain XML format for capturing all the additional pieces of information we needed to document about each component. We then merged the two sources using the name of the component as the key. This produced an integrated subject-domain document, which we then processed into a common document-domain format used for the whole documentation system and processed that document to produce the the formatted reference. Because the operating system components we were documenting supplied additional APIs to the operating system, we also merged in information from the API reference and created links from the component reference to the API reference, all with algorithms.

One of the downsides of API documentation tools like JavaDoc is that they tend to be tightly coupled systems that produce media domain output such as formatted HTML directly, often providing little or no control over presentation or formatting. This is a problem because it means that your API reference content does not look like the rest of your content. And worse, it is not integrated with or linked to the rest of your content. This has obvious consequences, like mentions of API routines in you programmer's guide not being linked to the documentation of that routine in the API reference. It would be much better to generate subject domain content from the API documentation tool and

then process it with the rest of your content. For many API documentation tools this is actually possible because many of them offer an XML output which may be either subject domain or document domain. Even if it is document domain, it may be regular enough that you can extract the subject domain structures reasonably easily. You can then skip the API doc tool's built in publishing chain and run its output through your regular publishing chain.

---

# Chapter 21. Information Architecture

Information architecture is the arrangement of content so that it can be found and navigated. In other words, it is an attempt to transfer some of the complexity of information finding from the reader to the organization. But information architecture can be tricky. It is all too easy to think of it simply in terms of organizing things, and that can be a trap. An institution with a lot of content has its internal information finding problems to deal with, and often information architecture becomes an exercise in organizing the content for the convenience of the institution rather than the reader.

It is easy to think that organization is an absolute. Content is either organized or not. But this is not so. Organization is an orientation of content according to our knowledge and expectations. Things are organized for us if their location, or the means of retrieving them, match what we expect and know. But what the institution (and its staff) knows and expects is very different from what the reader knows and expects. What is organization for the institution can be chaos for the reader.

More than this, unless content follows an organizational scheme that the reader is already familiar with, or at least makes intuitive sense in terms of what they expect and know, few readers will be likely to spend much mental energy on trying to figure out a complex organizational scheme, even one that was designed with them in mind. Instead, they will forage for information, which, in the Web world, means that they will use search and will follow links as long as they believe that the scent of the information they are looking for is growing stronger. (We discussed information foraging in Chapter 15, *Linking*. For a more in-depth treatment of information foraging and its implications for information architecture, see *Every Page is Page One: Topic-based writing for Technical Communication and the Web*.)

It is a mistake, therefore, to think of information architecture simply in terms of organization. A building supply store organizes building materials. An architect takes those materials and uses them to construct an navigable edifice full of useful spaces with efficient passages between them. The organization and standardization of materials provided by the building supply store is highly important to the architect in their work, but organization and standardization are just starting points for the unique and useful edifice that the architect will design and build.

We have always had information architecture. The term is new, but not the need or the concept. In the past, though, information architecture was divided into two pieces. The basic unit of information was the book and the “architecture” of the book was an integral part of the responsibility of the author and editor. Larger sets of information were created by collecting and organizing books and that was the responsibility of the librarian or bookseller. (Libraries and book stores have different information architectures to serve different purposes.)

Those larger collections were, indeed, no more than forms of organization. If there was an architecture at that scale it was in the expertise of the librarian or book seller to make inferences from the clients needs and create useful connections.

With the advent of online media, first in the form of large capacity electronic media such as CD-ROMs and then the Internet and the Web, this division of architectural responsibilities was overthrown. The basic unit of information in electronic media is not the book but the page. Thanks to hypertext linking, the relationships between pages in electronic media are much more complex than on paper. This makes it possible to practice real information architecture, not mere organization, at a much larger scale than before.

Also, the architecture of online media has to account for the ability to add, modify, and delete individual bits of content at any time. It is possible to think of book or library architectures in largely static terms. It is a serious mistake to think of Web architectures as static.

This leads to the development of architectures of much smaller units with much more complex relationships to a much larger, more diverse, and more rapidly changing set of resources. These architectures include not only text and static graphics but active media: videos, animation, and dynamic feeds and information widgets. Given these factors, the old separation of roles between writer and librarian no longer works. Authors have to be much more conscious of how their pages interact with other pages in the collection, including those created by others. The scale at which these small pieces of content relate with each other is much greater than the scale at which the pieces of a book related to each other. This constitutes a significant increase in complexity and calls for a whole new approach to information architecture, and for the appearance of a function and a role that had no equivalent in the paper world. Thus the term “information architecture” was born, not to name something entirely new, but something transformed by new technology.

Information architecture as a discipline in its own right, as opposed to being an aspect of authorship or librarianship, has arisen to combat the chaos that emerged in many websites as they began to grow, lacking an overall organizing principle or influence. But we should recognize that information architecture is as much part of the book world as it is part of the web world, even if it was not traditionally a job title in the book world. And if you are producing both web-like content and book-like content, your information architecture has to comprehend both. I have spoken at many points about structured writing, particularly in the subject domain, being used to transfer complexity from the writer to the information architect or the content engineer. It is the vast increase in the complexity of information architecture that makes this transfer necessary. But it also points out how much information needs to pass from the writer to the information architect for the information architect to do their job and not let any of the complexity slip through the cracks. Correct partitioning is essential to success in this area.

Because information architecture involves the organization of large bodies of content it can benefit greatly from structured writing techniques. Structured writing can give works of content a more definite character and identity which makes them easier to organize.

By providing guidance and validation to authors it allows information architects to better communicate and validate requirements. By making the content more accessible to algorithms, it allows the use of algorithms to do information architecture tasks, such as the automated organization and linking of content.

## Top-down vs. bottom-up information architecture

How can structured writing structures and algorithms support information architecture in structured writing?

I'm going to start with making a basic distinction between two types of information architecture: top-down and bottom-up. Top down information architecture deals with navigational aids and organizing systems that stand apart from the content and point to it. A table of contents or a website menu system is a piece of top-down information architecture. Bottom-up information architecture deals with navigation and organization that exists within the content itself. A web site with a consistent approach to hypertext links within its pages is an example of a bottom-up information architecture.

But bottom-up information architecture is not just about linking, it is about the way content is written. A topic in a bottom-up information architecture is designed to be entered via search or links from almost anywhere (as opposed to being designed to be entered exclusively from a previous chapter). But it is also designed to help readers with onward navigation, to be a hub of its local subject space, offering readers many onward vectors according to their needs and interests. I call this approach to information design Every Page is Page One, and it is described in my book, *Every Page is Page One: Topic-based Writing for Technical Communication and the Web*. One of the key principles of Every Page is Page One is that a topic should follow a well defined rhetorical pattern or type. Structured writing, particularly subject-domain structured writing, is very useful in developing Every Page is Page One content. For more on rhetorical patterns, see Chapter 30, *Rhetorical Structure*.

Bottom-up and top-down information architectures are not incompatible with each other. In fact almost every information architecture has both top-down and bottom-up elements. (Books, for instance, which are principally top-down, based on a table of contents, may also have internal cross references, which are a bottom up mechanism.)

Structured writing can be used to drive both the top-down and bottom-up aspects of information architecture.

## Categorization

One of the key elements of top-down information architecture is categorization. An information architect develops categories of content and develops an organizational

schema (such as a table of contents) based on those categories. This may include levels of subcategories forming a hierarchical categorization scheme.

Not all categorization is hierarchical, though. In some cases content can be classified on several independent axes, allowing for the development of what is called faceted navigation. The easiest place to see faceted navigation in action is on a used-car site where you can narrow down your selection using any set of criteria that matter to you, such as selecting blue convertibles or all-wheel drive vehicles with manual transmissions.

Categorization may be implemented as part of the content management algorithm, with categories implemented as part of the external metadata that a CMS applies to a content object. This is common practice when dealing with content in the media domain or the document domain.

For content in the subject domain, however, the metadata required to assign a piece of content to a category may be inherent in its subject domain markup. It is the nature of the subject domain to describe the subject matter and any markup that describes the subject matter may already contain the fields that you need for categorization. This is one of the attractions of the subject domain: the markup can serve many purposes, which simplifies both markup design and content authoring and often means that you don't need to create additional structures to support a new algorithm.

Relying on the subject domain metadata already in the content, rather than creating a separate metadata record, can be a tremendous advantage, because it makes submission of content to a repository so much easier for authors. But in some cases it can also avoid the need for a costly CMS altogether, since it allows the publishing algorithm to categorize content at build time without the need of a separate metadata store or a separate system to manage categorization.

## Linking

We have covered the linking algorithm already (Chapter 15, *Linking*), but linking is at the heart of a bottom up information architecture. In a bottom-up architecture, a page is not simply a leaf on a tree: the prize you find at the end of the search. It is a junction point in the exploration of an information space and the quest to understand a subject. In reading a page, a reader may discover new subjects that they need to understand and new options that they need to consider (deflection points in the content). They may discover that what they thought they knew is wrong, or what they thought they wanted to do was not the right choice. They may find that their search or their traversal of the categorization system has led them to the wrong place, or they may discover whole new worlds they wish to explore. At a more mundane level, they may discover that they need additional information to complete their task, such as reference data.

These are all pointers to some next topic that the reader needs. Even the most prescient writer cannot have chosen all of them as the linear next topic in a linear narrative. To

serve the reader they need to pave all of these possible paths for them, and the way you do that is with hypertext links.

This means that linking is not something that happens at arbitrary points where the author feels like adding a link. It is something that is planned for as part of the information architecture. Whether you specify hard links in the media domain or the document domain, manage them with keys in the management domain, or generate them from subject annotations in the subject domain, they should be created in a disciplined and consistent manner according to a deliberate plan.

## Tables of Contents

Tables of contents can serve various purposes depending on the nature of the work. Some describe a linear reading order for a work, some provide a classification scheme for random access to the content, some are simply a list of chapters that does not necessarily imply an intended reading order.

A table of contents may seem like a document domain structure, but it is really more of a media domain structure, for two reasons. First, it contains specific links to specific resources at specific addresses, or specific page numbers in a paper or a virtual paper format such as PDF. Secondly, it is virtually always factored out in document domain markup languages. Tables of contents are not written, they are generated.

From an information architecture point of view, what matters is how they are generated. In DocBook, for instance, it is typical to write each chapter of a book in a separate chapter file and then pull them together into a book using a book file. The order of the table of contents is then determined by the order in which the chapters are listed in the book file. The TOC itself is generated by extracting chapter and section headings from the chapter files in the order they appear in the book file.

In DITA, the normal process is to assemble a book using a map file. A map file may assemble a book out of DITA topics or other maps, and this may include assembling the chapters from topics as well. In the end, though, the TOC is generated in the same way, by traversing the document assembled by the map.

In both these cases, the order of the TOC is specified by hand by the person who creates the book or map file. But there are other ways to determine the order of content in a TOC. For instance, a reference work such as an API reference may be organized by listing each library in order by name, and each function in alphabetical order by name within its library, creating a table of content with two levels. There is no need to write a map file to create this table of contents. There is an algorithm for creating this table of contents. In fact, it is the algorithm stated in the first sentence of this paragraph, “listing each library in order by name, and each function in alphabetical order by name within its library”. Here is that algorithm expressed in pseudo code:

```
create toc
 for each library sorted alphabetically
 create toc-entry library name
 for each function in library sorted alphabetically
 create toc-entry function name
```

Tables of contents serve different purposes. Some describe a curriculum, a designed reading order. Others are simply a means of navigation, a way to select one topic out of a collection of many. If your content is written in the subject domain, the chances are that it already contains the structures on which such a classification could be based, and again the TOC can be generated based on the metadata already in the content.

One advantage of this approach is that if a TOC is assembled based on metadata in the content, that means that when new content is added, it is automatically included in the TOC the next time output is generated. This simplifies the task of adding new content to a collection by avoiding the need to update multiple files or systems when a update occurs. This makes life easier for authors as they do not need to know how the TOC is constructed. They only have to create an individual piece of conforming content and submit it to the right location. This also constitutes yet another example of the single source of truth algorithm, since the basis for the content's inclusion at a particular point in the TOC is stored only in one place.

## Lists

A major feature of a bottom-up information architecture is the list. Like tables of contents, lists are a catalog of resources. But while a TOC is a list of resources defined by their container (contents = things in a container) a list may have any principle of organization or inclusion.

For instance, you might want to have a list of all the movies starring each actor in a collection of movie reviews. Such list are not only a useful piece of information, they are also an important aid for navigating around a site. Maintaining such list by hand would be laborious and error prone, especially with new movies being added to the collection all the time.

If you have your movie reviews in a structured format that lists the actors in the movie in a format accessible to algorithms, like this:

```
movie: Rio Bravo
 starring:: actor
 John Wayne
 Dean Martin
 Ricky Nelson
 Angie Dickinson
```

Walter Brennan

you can generate the filmographies for all your actors, like this:

```
create-filmographies
 for each unique actor in movie/starring/actor
 create filmography actor with link to actor
 for each movie where starring/actor = actor
 create entry movie with link to movie
```

Tables of contents are a top-down information architecture device. You expect to find them at the top of the information set. Lists are a bottom-up device. You expect to find them as independent pages or as elements within a page. Thus if our collection includes the biographies of actors, and we want each biography to include the filmography, we can omit the filmography from the subject domain version of the biography and add it to the output in the presentation algorithm.

```
match actor-bio
 create html
 create h1 "Biography: " + actor-name
 continue
 create h2 "Filmography"
 for each movie-review where starring/actor = actor-name
 create li
 create a with attribute href
 = address of movie-review
 output movie-name
```

It is worth noting that, besides being part of the information architecture algorithm, this kind of thing is also a sophisticated example of the reuse algorithm. It takes a set of movies, each with a list of actors, and uses it to generate a list of movies for each actor, reusing existing information to create new content. This happens without any explicit reuse markup.

## Personalized content

Another key feature of modern web architecture is personalized content, which means content that is generated in response either to what the site already knows about you (from your account information, or a transaction token such as a cookie, or the selections or entries that you make on the page). For example, when you log into Amazon, the first page you see is crafted for you based on everything Amazon knows about your browsing and purchasing history. As you make selections, such as adding an item to your shopping cart or wish list, that information is used to shape the next page you see.

If you browse a used car site like Autotrader.com, you can select those features of a car that you are interested in (red convertibles with manual transmission under \$20000, for instance) and the next page will be built based on that input.

The ability of a site to personalize pages depends on its ability to identify content that is relevant, based on everything they know about the reader, and to assemble those pieces to form a page. For this to work, the content has to be easy to identify unambiguously, and it needs to be highly composable.

As we have seen, these properties are maximized when content is stored in the subject domain, both because the subject domain makes the relevant metadata available, and because working in the subject domain helps authors produce more consistent content that works better with these algorithms.

The consistency of the content is most important in any personalized content application. There is no possibility for an author or editor to inspect the output of a personalized content publication before the reader sees it, since it is assembled in real time based on the unique things we know about each reader. This requires total confidence that:

- the content conforms to its constraints
- those constraints are completely and correctly expressed by its markup
- the algorithm correctly processes and delivers the content

All three of these requirements depend on the soundness and simplicity of the markup design. They require precise content structures with few alternatives, clear guidance for authors, and good audit capability. Without these properties, content and its markup will be inconsistent and reliable algorithms will be hard to write and test because of the wide variety of markup combination they may encounter.

Most personalized content applications model their content in relational database tables for these very reasons. However, with the correct markup design, almost certainly in the subject domain, there is no reason why you cannot use markup-based tools and solutions to achieve the same thing. (Personalized content is an example of active content, which we discussed in Chapter 28, *Active content*.)

---

# Chapter 22. Change management

Content changes all the time and those changes are a major source of complexity in the content creation and delivery process. Structured writing can do a lot to partition and redirect change and the consequences of change.

There are several reasons why content might change. For our purposes it is useful to consider the following sources of change:

Subject matter changes

Content has to change when the subject matter – the real world stuff that it describes – changes.

Stylistic changes

Content may also change because we decide to express ideas differently. For instance, we may discover a better way to present a certain class of content so that it works better for the reader, and then decide to change all existing instance of that class to fit the new structure.

Externally-driven changes

Content may change because of changes external to itself, its style, and its subject matter. For instance, if a item contains a link to another item and that item is removed and replaced with something new and different, you have to change the link in every item that links to it, though the subject matter and style of those items has not changed at all.

One important motivation for structured writing is what is often called “future proofing”. Future proofing means building a system or product with a view to making it able to survive future changes in environments or requirements. Future proofing is difficult because you cannot know with certainty what changes will occur, how likely they are, or what they will cost.

Building a future proof platform can increase up-front costs, delaying the time it takes to get to market and possibly missing a window of opportunity. Nor can you be sure that your investment will every pay off, since the future you prepared for may not be the future you get. But not building a future proof platform can result in you not being able to keep up with developments in a market and losing your early lead. It may require massive and expensive changes when future events render your current system obsolete. Instances of both problems abounded when traditional publication systems were confronted with the rapid rise of the Web.

The safest approach to future proofing is not to try to anticipate the particular way in which the future will develop, but to create features that will be of value no matter what happens in the future. Creating content in the subject domain is the best way to practice

this kind of future proofing, because writing in the subject domain creates metadata that contains only true statements about the subject matter itself. Those statements are going to remain true as long as the subject matter itself remains unchanged. That is as future proof as you can make your content.

For example, suppose you write your ingredient list in reStructuredText as a table:

```
===== =====
Item Quantity
===== =====
eggs 12
water 2qt
===== =====
```

Later you decide that you want to present ingredients as a list instead. To do this, you will have to go back to your content and change the markup. Doing this across a whole collection of recipes will be expensive.

Suppose instead that you use subject domain markup:

```
ingredients:: ingredient, quantity
eggs, 12
water, 2qt
```

Now you don't have to change the content to make the change in presentation. You just change the presentation algorithm to turn an ingredients record set into a list instead of a table. Thus the subject domain markup has future proofed your content against this change of presentation. The document-domain reStructuredText markup specified the use of a table, which is not a truth about the subject matter but a decision about presentation that can change independent of the subject matter. The subject domain markup simply specifies that "eggs" is an ingredient and "12" is a quantity. These are truths about the subject matter that will not change. Thus they are invulnerable to future changes outside of the subject matter itself.

Moving your content from the media domain to the document domain provides a degree of future proofing. By factoring out the formatting details, it protects your content against changes in formatting rules such as the font to use for headings. Moving your content from the document domain to the subject domain provides additional future proofing. By factoring out the content and organization of documents, it allows you to target different publications and to create different document designs for different media. For example, it lets you output different beverage suggestions to *Teetotaler's Trumpet* and *Wine Weenie* without making any changes to your source file.

Structured writing imposes specific structures on content for specific purposes. It does not make content magically immune to change nor does it guarantee you will not have to

rewrite the content or change the structure to accommodate future changes in your subject matter or your business requirements. You can, however, design your content structures to help you manage specific and foreseeable changes. If you are lucky, the structures you create may also allow you to adapt content for unforeseen circumstances, particularly if your content is stored in the subject domain. But this is a bonus. You cannot guarantee any content or structure will work for things you have not foreseen.

Changes in content happen all the time. Many of them are entirely predictable and you can use structured writing to support the management of those changes. For instance, companies re-brand from time to time, often requiring a change of formatting for all publications. If the content is in media domain structures, the effort to change to a new appearance could be significant. If the content is in the document domain, however, changing how it is formatted is simply a matter of changing the formatting algorithm to produce different-looking output.

It is worth noting, though, that while changing the formatting algorithm is less work than changing the formatting of a large body of content, it is also more complex work. It requires a skill set that is not as widely available as the skill of changing fonts in a word processor, for instance. It also cannot be done incrementally. Once the entire new algorithm is written, all the content can be converted to the new look almost instantly. But until it is finished, none of the content can be converted. A structured writing system is not the kind of thing you can set up once and walk away from. If designed properly, it transfers complexity from writers to information architects and content engineers and so you need to maintain the availability of those skill sets to your team. Hiring someone to come in and write a bunch of algorithms for you as if they will never need to change is to ignore where complexity is being directed in your new system.

Moving complexity to algorithms is the heart of the productivity and quality gains that you get from structured writing, but transferring complexity to an algorithm means transferring it to the person who writes and maintains your algorithms – your information architects and content engineers. In many cases, the change management strategy you are implementing depends precisely on this: that in the event of a certain kind of change it will be handled by changing a single algorithm rather than by changing thousands of pieces of content. Insisting on a fixed and invariant tool set essentially mean cutting yourself off from the possibility of partitioning complexity, particularly the complexity of change, in this way. This will greatly limit the extent to which you can bring much of your currently unhandled complexity under management, and thus the amount of complexity you can direct away from your readers.

A general move to the document domain (or the subject domain, or even a disciplined use of styles in a word processor) will allow you to handle font and layout changes. But what if the re-branding goes further? Suppose it involves changing the names of products or even the company. Should your structured writing approach explicitly support that change? Some organizations like to mandate that writers insert a variable rather than the actual name for the company name and all product names. That way, when a product name or the company names changes all you have to do is redefine the variables.

We here at >(\$company-name) do not recommend using our product to catch roadrunners.

I have always been skeptical of the value of this practice. It forces the writers to remember to use the variable every time. This interrupts their chain of thought, which slows their writing down and uses up some of their precious attention, thus impacting content quality. And it is virtually impossible to ensure compliance. Writers will sometimes simply forget and write the names out normally, which means you always have to search for these instances anyway when a change happens. Then there are issues with historical usage of the names, where you don't want the change to happen, and with inflections if the new or old names end in 's' (in English; other languages may have different inflection problems).

Company and product names are distinct strings that are easy to search for when you need to make a change. The overhead of creating and maintaining the variables is greater than the overhead of doing a search and replace through the content when a change occurs. The use of variables, in other words, is creating more complexity than it is partitioning and transferring. And doing a search and replace allows you to make intelligent choices about historical usage and inflections. If your content is held in text form (in a markup language) in a repository (file system or content management system) that allows you to do a search and replace across multiple files, this is probably easier and more reliable than using variables. (And it is what you are going to have to do anyway if there is a name change that you did not anticipate and therefore did not use variables for.)

You may well need some markup for company and product names, however. You may want to format them differently or link from them to more information about the product or company. Rather than use a variable I would rather use an annotation like this:

We here at {Acme Corporation}(company) do not recommend using our product to catch roadrunners.

This second approach identifies the words "Acme Corporation" as a company name. Creating this markup requires no extra thought from the writer. They do not have to remember what the appropriate variable name is. (They do have to remember company as an annotation type, but that is a type, not an individual name, and if your markup is well designed your annotation types should be few and memorable.) And this same markup can be used to format the company name appropriately and to generate links to information on the company.

This does not guarantee that the writer will always remember to add the annotation, or that they will always spell the company name correctly. (There is no way to guarantee that a free-floating annotation will always be remembered. The best you can do is make them easy to do.) But you can use the company annotation to find all the phrases marked as company name, sort them, and look for variants. This then allows you to go back and

fix incorrect spellings. But it also allows you to identify the ways in which writers are misspelling the company name and search the whole text for those misspellings. This improves your success rate catching both misspellings and the failure to annotate. This kind of content hygiene operation should be performed regularly on any content set, and subject domain annotation makes it easier to do while removing a distraction for writers.<sup>1</sup>.)

At another level, re-branding can involve the organization deciding to change its tone or voice. It may wish to go from professional and reserved to friendly and jocular. There is no way, of course, for any structured writing process to recast content from formal to funny. You can't make your content proof against every kind of change.

One form of change that is so common that it may be overlooked is simply the ongoing creation of new content and the editing of old content. In the age of the Web, this is a particular concern because we can now add a new piece of content whenever it is ready, edit an existing piece whenever it needs it, and delete an old piece whenever it becomes obsolete. We don't have to wait for a major publication release for all these changes to roll out. Each rolls out when it is ready. And each time one rolls out, it impacts the information architecture of the entire content set.

Adding, editing, or deleting one topic does not mean that all the other topics are unaffected:

- There may be topics that link to the deleted topic.
- There may be topics that should link to the newly added topic.
- There may be topics that should no longer link to a changed topic, and topics that should now start linking to it.
- Topics in a category may now have a new neighbor or may have lost one.
- Any top-down navigation tools need to be updated for the topic changes.
- Deleted topics may leave holes in the information set that need to be filled.
- New topics or edited topics may mention subjects that are not adequately covered by existing topics, revealing the need for yet more topics.
- Deleted or edited topics may leave other topics orphaned, needing to be removed or edited to serve a current purpose.
- Events in the world can change the status of a whole set of topics, for instance, those relating to an upcoming event which becomes a previous event the moment it has taken place.

---

<sup>1</sup>For more on this, see [#chapter.taxonomy]

When there is the potential for adding, editing, or deleting a topic to have ripple effects through the whole content set, and when such additions, deletions, and edits happen on a daily basis, it is vital to have algorithmic support for change management. Managing all the effects by hand is doomed to failure.

## Content management systems

Content management systems often have change management features that can be helpful. For instance, many of them will inform you if a change or deletion of an existing topic will break any existing links. They will also help you find topics on related subjects or manage the membership of categories and the navigation aids that are based on them. What they won't do is tell you things like which pieces of existing content should be linking to the new content you just added. The only way to discover that is if the existing content contains subject domain annotations that relate to the subject for the new topic. With these in place, a linking algorithm (as described in Chapter 15, *Linking*) can discover these relationships automatically.

As should be coming clear by this point, the change management algorithm is really an aspect of all the other algorithms. For any of the algorithms to keep working over time and as content changes, the structures that support those algorithms have to stay conformant. Part of designing those structures, therefore, should be thinking seriously about what it will take to maintain them in a conformant state when changes happen. Change management, therefore, relies heavily on the conformance algorithm. And this is a reciprocal relationship. To maintain conformance of your content, you have to be able to manage change successfully. But to manage change successfully, you need content to be conformant so that you can reliably identify the changes that are required and, as far as possible, execute those changes algorithmically.

For example, say you have a movie review site that contains movie reviews as well as biographies of actors and directors. You have a review of *Rio Bravo* marked up like this:

```
movie-review:
 movie-title: Rio Bravo
 review-text:
 In {Rio Bravo}(movie), {the Duke}(actor "John Wayne")
 plays an ex-Union colonel.
```

At first, you do not have a biography for John Wayne on your site, so none of the reviews that mention John Wayne have a link to a biography. Then you add a John Wayne biography to your collection:

```
topic:
 title: Biography of John Wayne
```

```
index:
 type: actor
 value: John Wayne
body:
```

John Wayne was an American actor known for westerns.

Suddenly every movie review that mentions John Wayne in the text or lists him as a star should be linking to this biography. That might be fifty pages of your site that now should be linking to the topic you just added. That is a lot of change to process just because of one added page. But if you have used the linking technique based on subject annotations, as described in Chapter 15, *Linking*, then all you have to do is rebuild your content set and those links will be created automatically. There is literally no separate change management step that needs to be done to get those links. It has been entirely factored out.

As we noted in Chapter 16, *Conformance*, the heart of conformance is designing structures that are easy to conform to. Change management is not something you can tack on as an afterthought, but nor is it something that necessarily requires a separate set of structures. Content that is highly conformant and highly auditable is easy to change consistently, which in turn helps maintain conformance. The heart of the problem lies in designing content for conformance.

---

# Chapter 23. Content Management

Structured writing is an instrument we use to help us partition and redirect the complexity of content creation and delivery. In other words, it is an instrument of content management. There are, of course, many tools available to help you manage content. Collectively, they are called content management systems. They are quite varied in their concept and operation because they partition and redirect different parts of the complexity of content creation and delivery, and/or they partition and redirect complexity in different ways. Some systems attempt to be all encompassing – to be the only tool anyone in the organization ever uses to create and deliver content. Others are designed to work in concert with other tools. Some are more frameworks than tools, platforms on which you can construct your own content management functionality. All of them implement some form of structured writing for at least some of their functions, some in the media domain, some in the document domain with more or less of the management domain mixed in. Some work with, or at least support certain management functions for, the subject domain.

If you use a content management system, it is important to understand how the partitioning and redistribution of complexity that the CMS is designed to do fits with the partitioning and redistribution that you are planning to do in your content system. If the two are not a good match, additional complexity will be created, and some of the complexity you are seeking to manage will be dropped and fall down to the reader.

While content management, which is the partitioning and redirection of complexity in the content system, is actually concerned with the whole of the content problem, when we speak of content management systems specifically we are mainly talking about two problems:

- Orchestrating content for the publishing process. That is, making sure all the right bits get into the right documents at the right time. In many cases, there is a workflow aspect to this, making sure that every piece of content is seen, acted on, and or approved by every interested party in the organization prior to release. It may also include rights management issues, if there are different people with a financial stake in the content who may be entitled to compensation based on its use. These functions are well covered by other works, and I don't intend to discuss them in any detail here.
- Orchestrating content for writers during the writing process. For instance, if you are practicing content reuse, writers need to find content to reuse, and a content management system can help. Change management is also a huge part of content management: when a change happens in subject matter or in business requirements, how to you make sure all the necessary content changes are made, and made efficiently?

# Metadata is the foundation of management

Content management system do their job largely through the collection and management of metadata. Partitioning and redirecting complexity requires a method to pass information between partitions in a reliable way, and that is what structure and metadata do. Metadata is the record of the identity and status of content. Management actions are actions on metadata: either creating and updating metadata or performing actions (running algorithms) based on metadata. (More on metadata in Chapter 35, *Metadata*.)

I defined structured writing as writing that not only obeys constraints, but also records the constraints that it follows. The record of those constraints is metadata. In fact, most of the metadata that a content management system manages is simply the record of the constraints that content obeys. This includes much of the metadata related to workflow, since the workflow requirements of a system are also constraints on the content.

The location of the metadata that records the identity and status of the content, that records the constraints it obeys, differs from one structured writing domain to another. The media domain captures virtually no metadata that is useful for content management, the document domain captures some, but not enough, and the subject domain often captures almost everything you may need, except perhaps for workflow information. (Though workflow information is management domain data and there is nothing to prevent you from adding those structures to your document domain or subject domain content if you want to.)

Where the content does not contain the metadata necessary for management, the CMS has to gather and store it separately. This naturally adds complexity to the CMS interface, a complexity that is a major source of pain for many users. This also illustrates that the kind of CMS you need for content in different domains can vary significantly. In the case of CMSs that are designed to be the only tool you use, this generally means that the structured writing format is baked in and can only be changed in limited ways, if at all. A transfer of metadata from the CMS to the structures of the content would involve a major reconfiguration of the entire CMS. If you buy this kind of system, therefore, you need to look at the total picture – how it partitions and redistributes all of the complexity of the content systems, and what complexity, if any, it neglects and lets fall through. There will likely be little you can do to change the partitioning or distribution after the fact, so you better make sure it is what you want going in.

Most off-the-shelf content management systems are designed for media domain or basic document domain content. This makes sense from a commercial point of view because it allows them to develop their own metadata scheme and associated management algorithms independent of the content that will be stored. This means they can be sold to a wider variety of clients and also that they can advertise that they have simple editors

or work with the editors that people already have. The problem with this model, in terms of developing a comprehensive solution for managing complexity across the content system, is that it draws a hard line between the management going on in the CMS and any type of management or constraint you may wish to impose on the composition process. This means that algorithms that depend on the consistency of content or its relationships with other content are largely unsupported by the CMS, and there is no integration between those algorithms and the algorithms preformed by the CMS. The result is often that a lot of complexity gets dropped due to this lack of integration and falls through to the reader.

Other CMSs are built for more complex document domain languages, which typically means that they are built to support the specific management domain features of those languages. By far the most common instances of such systems today are based on DITA. Typically, such systems are specific to the one language they support and are sold as such: a DITA CMS or an S1000D CMS.

You might expect that the principal type of metadata contained in a CMS would be management domain metadata. After all, we described the management domain as an intrusion into the structured writing world, since it does not actually describe the structure of content. The reason for the intrusion of the management domain into content is to allow for the management of the content below the level of whatever file or chunk size you store in the CMS.<sup>1</sup>

But while you will rarely find much in the way of media domain or document domain metadata stored at the CMS level, CMSs often contain a great deal of subject domain metadata. If you are managing a large volume of content, you will need some way to find content on a particular subject. If you are doing content reuse, for example, you will constantly be asking if content already exists on the subject you are preparing to write about. If your CMS is managing the delivery of content dynamically to the Web, it will need to respond to queries based on subject matter. And if you are optimizing your content for search you will need to provide the search engine with subject metadata in the form of keywords or microformats. All of this depends on subject domain metadata. Subject domain metadata is therefore central to CMS operations.

## The location of metadata

It is a very common pattern for a CMS to store document domain or media domain content and attach subject domain metadata to it as an external label. For instance, a CMS might store recipes written in MarkDown and attach separate metadata records to each recipe listing the key recipe metadata needed for retrieval and sorting of recipes. One

---

<sup>1</sup>In some CMSs, this distinction between the chunk stored in the CMS and the structures expressed inside that chunk is moot. A CMS based on a native XML database, for instance, makes no distinction between the chunk and the structure of the chunk, but treats the entire repository as a single XML resource that it can query and manage down to any level of granularity. Even with such a system, however, this distinction remains for the writer, who has to deal with the structure of whatever sized chunk of content they are being asked to write.

of the things that writers often complain about with CMS systems is that they are not allowed to submit content to the system without filling out complicated metadata records.

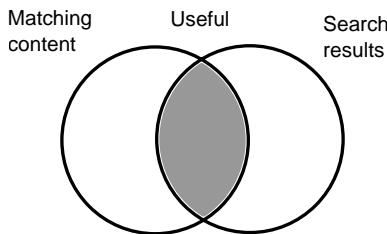
An alternative approach would be to write recipes in a subject domain format in which all the recipe metadata is included in the content from the beginning. The CMS then requires no external metadata label, though it does obviously require a way to access and query the metadata embedded in the content. (CMSs based on XML databases often have this capability as a natural consequence of the XML database architecture.)

Which approach is preferable? The conventional CMS approach arises because most CMS's are based on relational databases, which are good at storing metadata records and attaching them to blobs of text, but are not good at storing or querying the hierarchical structure of structured content. It has several disadvantages, all of which introduce complexity which is often not well handled.

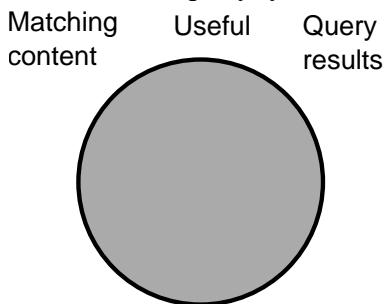
1. It can only record the characteristics of a chunk of content as a whole. It cannot look down into the content to find more fine grained metadata. One of the advantages of writing a recipe is the subject domain is that it allows you to do things like querying the collection of recipes for all those with a calorie count below 100. But unless the metadata record for the recipe includes that level of detail, the CMS cannot respond to that query. And if the CMS does store that level of detail, it is effectively asking the author to write the entire content twice, once in the document domain and once in the subject domain. Not only is this more work, it is quite likely that the two versions will fall out of sync with each other.
2. It gives no support for subject-domain validation of the content. It does nothing to help improve content quality. By requiring document-domain content as the storage format, it precludes the use of the subject domain for authoring and cuts you off from all the advantages it provides.
3. The system has no way of telling if the content meets its constraints. It records the content constraints in a separate record without ever validating that the content meets them.
4. It separates the metadata from the content it describes. This allows for drift between the content and the metadata.

But storing metadata in the content presents some challenges as well. Having each piece of content stored in the subject domain makes a lot of sense from a semantic point of view and makes it easy to submit content, since no additional metadata forms have to be filled out. The problem is how to retrieve it. A CMS is essentially a database, and the way you retrieve information from a database is to write a query. A query is different from a search. A search is fuzzy. A search engine takes a plain text question or search phrase and tries to figure out which documents are the best match. Search engines may be powerful and sophisticated, but their results are essentially a sophisticated mechanical guess, and sometimes they get it wrong. Ask a search engine for a list of recipes with less than 100

calories, and it will give you a bunch of guesses based mostly on the plain text of those documents. Chances are it will catch some, miss others, and give you some false hits.



A query, on the other hand, is a precise request for items whose metadata precisely matches specified criteria. If you write a query to return recipes for which the value of the field recipe/nutrition/calories is less than 100, it will return all the results, miss none, and give you no false hits. However, it will work only for content that is stored that way. To write that query, you will need to know exactly how recipes are stored in the system.



If you have many different content structures in your repository, you will need to know how each of them is structured in order to create the queries to return them. This is not the end of the world. Information architects and content engineers can save writers from having to remember how to do all of the queries by creating saved queries that they can run at any time. But it is still a complicating factor.

In the end there is no way around this. Accurate reliable queries depend on precise consistent metadata. Precise consistent metadata is specific to the object it belongs to. There is no such thing as a generic metadata record. They are always specific to the things they describe. Subject domain metadata is specific to its subject. If you want to be able to find all recipes with calorie counts less than 100, you need recipe specific metadata that specifically records the number of calories in the recipe. If you want to find a used car listing for a blue convertible, you need metadata that specifically records the car color and body style. There is no generic metadata format that supports both these queries. The inherent variability of content means no generic query system is possible. The trick is to find the right balance and to make your content as queryable as possible.

## Managing the process

When we create an individual piece of content for one-time publication in a single media, there is really not much of a role for management in the process. Content management becomes a concern when you want to manage the production and publishing of many

pieces of content, to manage the relationship between them, ensure consistency and quality, or to publish them many times in different ways.

Of course, many of the reasons we have looked at for moving content from the media domain to the document domain or the subject domain have to do with managing the production and publishing process. But managing a body of content and the processes and tools that create and process that content, requires a whole set of metadata of its own.

Structured writing is about imposing constraints on content. Content management is about imposing constraints on the content process. But it is also about managing the constraints we impose on content.

Doing structured writing requires recording content in document domain or subject domain structures, factoring out invariants into separate files, expressing constraints, and creating algorithms to translate the content to the media domain for publishing. All of this creates a lot of artifacts to keep track of, and requires a process both for keeping track of them and for running the structured authoring and publishing tool chain. Thus there is a need to manage both the artifacts and the process. But don't fall into the trap of assuming that this is a generic processes. All these artifacts that require managing are the result of partitioning your content system, and how they should be managed is determined by how the partitions communicate with each other. This is specific to the overall partitioning strategy of your particular content system. You need to find a tool that fits that partitioning strategy.

A common mistake is to focus on some management problem and simply push out the related complexity to some other function. For instance, the heavy use of management domain markup can solve a bunch of management problems, but it pushes a lot of management complexity onto the writer, which they may not be able to handle without compromising quality. Nowhere is it more important to take an holistic approach than in the selection of you approach to content management and the tools you choose to implement it.

# Conflicting constraints

The purpose of structured writing is to help partition and redirect the complexity of the content system. The ultimate purpose of this is to make sure that none of the complexity gets dropped in the process and falls through to the reader. But as content collections grow, and as online content becomes more integrated, we also have a growing need to manage content, and to use algorithms to help manage content.

For algorithms to manage content they need to know what constraints it meets, so structured writing is a natural place to turn when you want to implement content management. We can partition and redirect content management complexity using structured writing techniques. Establishing constraints on content is part of the process of partitioning complexity and making sure that all the needed information is transmitted between partitions.

But the constraints you use to partition the content management problem may not always be the same as those you would implement to improve content quality. You may find your management goals at odds with your rhetorical goals. It is easier to manage content (or anything else) if it is more uniform. The constraints that you will naturally wish to impose to make content more manageable are those that make it more uniform. Thus a system like DITA which, as a starting point, proposes that there are just three types of content (concept, task, and reference) has an obvious appeal from a management point of view. Remember though, that it is the accomplishment of your rhetorical goals that serves your readers. Your management goals should logically be subservient to your rhetorical goals.

The constraints that you impose to improve content quality, on the other hand, are those which make sure that a piece of content does just the job it is supposed to do. They are the kind of constraints that make sure that a recipe contains everything a recipe needs and is presented in the way a recipe should be presented. They are highly specific to the subject matter and to the audience. Three generic content types are not going to provide all the constraints we need to effectively manage content quality. Indeed, some of the constraints that are designed to facilitate content management may be positively damaging to content quality.

We have something of the same issue with the publishing algorithm. To manage publishing effectively is to good to have a single highly constrained presentation file format (a pure document domain description of how the content should be presented). Such a format obviously does nothing to constrain how subject matter is expressed, so it does very little to address content quality issues beyond those related to consistent presentation and formatting. Working in a generic document domain format does nothing to improve conformance to rhetorical best practices or your specific rhetorical constraints.

But as we saw when we looked at the publishing algorithm, you don't need to write in the presentation file format. You can write content in the subject domain and translate it to the presentation file format as part of the publishing process. And if you are doing differential single sourcing, you may be translating it into two different presentation file formats for presentation in different media.

This works fine for the publishing algorithm. The algorithm gets the consistent presentation format it needs to work reliably and efficiently, but authors still write in the subject domain and thus have the benefit of subject domain constraints to manage content quality. All the other algorithms that support content quality can be run on the subject-domain content before or as it is transformed to the document domain for publishing. The consistent document domain format is then just a temporary artifact created as part of the publishing process.

But for the content management algorithm, this is no solution. The content management algorithm needs to manage the original source files in which the content is created. It does it no good to manage temporary artifacts created by the publishing process. If it is

to manage the source files in a system where content is created in the subject domain, it is going to have to deal with many more source formats.

But while this may be the approach you would naturally choose if you merely set out to implement a content management system – if improved content quality were not one of your business goals – I would argue that managing subject domain content may actually lead to better content management in the long run.

In any system that relies on constraints, on data that is known to meet certain constraints, it is necessary to make sure that the constraints are actually being met. This is the role of the authoring algorithm and the conformance algorithm. And as we have seen it is often much easier to provide effective guidance and perform effective validation in the subject domain. Also, the subject domain allows you to factor out many constraints, which is the most effective way of making sure they are obeyed. The document domain provides far fewer opportunities for factoring out constraints and providing effective guidance and is much more difficult to audit correctly.

Thus while a simple document domain system of concept, task, and reference topics meets the content management algorithm's desire for uniformity, it provides little opportunity for ensuring that the full range of constraints necessary to make content management and reuse work are actually followed. The result can be deterioration of the quality of the content set over time, a process that tends to be self perpetuating, as disorder in current content makes it harder to impose order on new content. (Just as you cannot put things away neatly in a messy drawer.)

The variety of constraints and formats found in a subject-domain system may present a greater content management challenge initially, but it can go a very long way to ensuring that the necessary constraints are met. And, as we have seen, you can often use subject domain structures to factor out management domain concerns (Chapter 6, *The Management Domain: an Intrusion*), which can go along way to removing the conflict between quality structures and management structures in content. This not only leads to more effective management, but also to a simpler authoring experience.

## Creating manageable content

We saw with the conformance algorithm that the key to conformance was to create structures that are easy to conform to. The content management algorithm relies heavily on the conformance algorithm. Successful management of content depends on knowing exactly what assets you are managing. The more you know about each asset, and the more reliable what you know about that asset is, the more confidently you can manage it, and the less likely the management system is to slowly descend into chaos.

What this means is that the content management algorithm depends on content that is easy to manage. It may seem like simple generic units of content would be the easiest to manage, but the problem with such generic units is that you know very little about

them and what you do know is often unreliable. Generic units may be easy to create and easy to store, but they are not easy to manage. It may require more initial though and effort to plan for the management of highly specific well constrained content units, but such units will in fact prove to be the easiest to manage over time, especially as your content set grows.

---

# Chapter 24. Collaboration

Many organizations produce large information sets that cannot be produced and maintained by a single person. Thus collaboration is required in producing and maintaining all that content. Collaboration is a difficult and complex process, and the more people you have trying to collaborate, the more complex it becomes. As Tom Johnson writes:

So often we place the bar for contribution at whether someone can write. In reality, it's not just whether someone can construct clear, grammatically correct sentences. It's whether the person can integrate the information into a larger documentation set.

—<http://idratherbewriting.com/2016/12/14/higher-level-technical-writing/>

And, of course, the larger the documentation set becomes, the more complex the task of integration becomes, to the point where it can quickly come to exclude the participation of full time writing staff that have been specifically trained in how to do it. How does each collaborator know what others are doing? How do they know which parts of the wider work they are responsible for? How do they integrate their work with the work of others? How do you manage the overhead created when collaborators have to be aware of other people's work as well as their own? As always, any complexity that is not handled in the organization gets dumped on the reader in the form of quality and findability problems.

The fundamental constrain on collaboration is the amount of time that collaborators have to spend on collaborative activities – orienting themselves to the work of others – as opposed to creating new work. Unless you do something to ease this burden you quickly reach the point where adding more collaborators actually slows the project down because every new person you add to the project increases the collaboration overhead by more than the amount of work time they add to the project.

The essence of managing collaboration, therefore, is to minimize the amount of collaborative overhead in the system. In other words, you need to partition the work of collaborators sufficiently that the complexity of communication with other collaborators does not overwhelm the work to be done. But unless you do this in a way that allows all of the necessary coordination of work to get done, the unmanaged complexity gets dumped on the reader. Before the Web, organizations handled the overhead of collaboration largely by assigning different books, pamphlets, and other publications to different writers and issuing them with some basic style guidance for language, physical appearance, and layout. All the other aspects of collaborative complexity, like making sure that everyone was saying the same thing at the same time in the same way, or that people were not creating the same content over and over again, or that at least one person was saying everything that needed to be said, were ignored and the results were dumped on the customer in the form of inconsistent information that was incomplete, incorrect, contradictory, and hard to find.

With the advent of the Web, all the same content, produced by the same non-collaborative approach to collaboration, got dumped onto the company website. It wasn't necessarily any better or worse than it was before, but because it was now all searchable in one place, the quality problems became much more obvious. Early attempts to allow employees to post content to the Web arguably made things worse, since they combined a technically difficult process with a complete lack of constraints on coverage, consistency, or style. In response to this mess, the discipline of content strategy was born.

As always, we can address these problem by partitioning and redirecting the complexity to make sure that each part of the burden is placed on a person or process best able to handle it. Structured writing can help by enabling you to more effectively redistribute the complexity while reducing the amount that slips through the cracks.

Management-domain-oriented languages, such as DITA, move a lot of the complexity of content management, and therefore of collaboration, into the source file in the form of management domain structures. If the writer is working directly in DITA, therefore, that complexity is transferred to the writer. In a collaborative situation, this means that that part of the burden is being transferred onto the collaborators, which means that it takes more time, discipline, skill, and knowledge to be an effective collaborator.

A well-designed management domain system backed by well-designed tools my reduce the collaboration overhead compared to doing it add hoc, so this may be a step in the right direction, but it is still leaving a lot of the collaborative overhead on the writer's plate and therefore the same basic limit on the effective scale of collaboration still apply, even if they have been pushed out a little. Alternative approaches can reduce the overhead further at all scales.

Part of the complexity of collaboration is the difference in skills, knowledge, and background between contributors. A good interface can correctly partition complexity for a collaborator, but with diverse users we need diverse interfaces that partition complexity in different way in accordance with the needs and capacities of their users.

Some of your contributors may be full time professional writers while others are engineers or marketers, field personnel or support people for whom communication is an important, but not central, part of their jobs. It makes perfect sense in these situations to design a collaborative system that distributes authoring complexity from your occasional contributors to your full time writers, or, better still, to your information architects and content engineers. People can handle far more complexity in their core task than then can in any of their peripheral tasks.

This kind of approach can be very fruitful, but it is not well supported by most off-the-shelf tools which essentially present the same interface to everyone. A structured writing approach in which different contributors use different structured writing languages, each best suited to their contributions, can be very effective. Of course, any use of structured writing distributes some degree of complexity towards the writer, since they are now obliged to know and follow the structure. On the other hand, this can also distribute a lot of complexity away from the writer. While the blank page may seem like the simplest

possible interface, it actually give no task guidance at all. It is an interface without any affordances. It leaves it to every author not only to write, but do design the information they are creating, to decide what needs to be said and how to say it. Structured writing can provide this design information to the writer, thereby distributing the design complexity to the information architect who designs the structured writing language. What is vital here, though, is that the structured writing language not distribute any other complexity to the writer. Any language that requires them to master publishing or content management concepts, for instance, is not going to work well for this purpose. What works is a simple subject-domain language the addresses the writer in terms they already understand and ask for annotations using concepts and ideas that the writer already knows.

One of the more common reasons for introducing content management and/or structured writing to an organization is to improve collaboration. It is certainly not the only way to facilitate collaboration. In fact, the more common approach is to create simple and largely unstructured tools such as message boards and wikis. One of the most widely collaborative projects in the world – Wikipedia – runs on a wiki using a fairly simple document domain markup language which is often hidden behind a simple WYSIWYG editor.

This is powerful model for collaboration. Complexity does not require or even respond well to central top-down control (which essentially pushes more complexity to the center than anyone there has sufficient attention or knowledge to handle). But it is also a model that distributes almost all of the complexity to people rather than algorithms. There is simply not enough structure in this model to allow you to transfer very much complexity to an algorithm. Nor is there anyone making sure that all of the complexity is getting handled. The system relies as much for auditing and content management on the uncoordinated work of volunteers as it does for content creation.

Why then might one turn to a more complex structured writing system for collaboration. In a word: integration. The kind of collaboration that is supported by message boards and wikis is one in which collaboration chiefly means everybody being able to see what everyone else is doing. Any connections between the pieces created by different people are loose and non-critical, mostly taking the form of hypertext links. And such connections as there are are managed by large scale community efforts. Wikipedia is full of links between articles largely because anyone can go in an edit an article to add a link to an article on a related subject.

But not all collaboration can rely on such loose and uncoordinated activity. In many cases you need to bring the pieces created by collaborators together to form an integrated and cohesive whole. You can do that by hand, of course, but that can be cumbersome and time consuming. It may be hard for any one coordinator to keep up with all the content that is being created, especially if new content is continually being written, without any freeze period to allow the integration to take place.

Equally important, the collaborators on a message board or a larger wiki like Wikipedia are largely ignorant of each other and each other's activity. Duplication of effort and

even outright contradiction may be frequent. If you have huge numbers of volunteers constantly reading the site and finding duplication and contradictions, as Wikipedia does, you can live with this (though there will always be parts of the system that are in error at any give time). But an organization that is paying its writers and editors may not be able to afford this labor-intensive approach. It may need a more efficient way to coordinate the activity of its collaborators to avoid duplication and error. This means using structured writing to transfer a lot of that complexity and effort to algorithms.

## Bridging silos

There is a lot of talk in content management circles about breaking down content silos. The naive way to do this is to have everyone use a single system and a single markup language. But as we have seen, this means either adopting a simple document domain language that everyone can learn, like Markdown, which does not have enough structures to meet everyone's needs, or adopting a large complex document domain language like DITA or DocBook that meets a lot of needs but has poor functional lucidity, especially for part-time contributors. It is a classic case of focusing on one problem and paying no attention to where the complexity that is directed away from that problem gets dumped.

The most pernicious myth about collaboration is that it requires everyone to use the same tools and to understand each other's work. In fact, this is the worst way to collaborate, because it creates a huge amount of overhead which can swamp the ability to actually get anything done. Efficient collaboration is actually achieved by limiting the amount that collaborators have to know about each other's work and each other's tools. This allows each group or individual to work efficiently while still creating a product that can be integrated successfully with the work of others.

This approach to collaboration is seen throughout the worlds of engineering and computer programming. The secret ingredient that allows workers to collaborate with minimal knowledge of each other's work is the interface. A structured writing language is an interface to content creation. It works by partitioning and redirection the complexity of the content system.

---

# Chapter 25. Timeliness

Information tends to change quickly these days, and readers no longer have any patience with outdated content. But there are many difficulties in ensuring that content is always timely. How do you detect when content is out of date? How do you push updated content to the reader quickly? How do make sure updates in one place don't break other content? These are all sources of complexity in the content system. They are also part of the complexity of many of the algorithms we have already looked at. Poor partitioning and redirection of complexity in any of these algorithms, or any leaking of complexity within them or in the way they hand off responsibilities to other partitions, will impact your ability to deliver in a timely fashion.

Traditional paper publication could endure considerable inefficiency in these processes because publication was a rare event. Indeed, publication was treated very much like a wedding day. All the effort and coordination of the preceding months went into making that day work, and normal life for many of those involved ceased several day before the event itself. The complexity of getting everything ready for that day, and of executing all the events of the day, was enormous, and tolerable only because such wedding day publication events were rare.

For modern Web publishing, a wedding day model of publication is untenable. Publication has to be a simple and quick process because it happens every day. Every day is publication day. You cannot treat every day like a wedding day. There can be no elaborate process of getting ready. Instead, you need to maintain your systems and your content in a constant state of readiness to publish. To attempt to do this without the correct partitioning and distribution of complexity in your content system is to be certain that much of the complexity will get dumped on the reader in the form of outdated, inaccurate, or inconsistent content. Structured writing provides ways to address all of these issues.

There is no separate structured writing algorithm for timeliness. Timeliness rests on executing the entire writing process, from ideas in the head to dots on a page in the hands of a reader, in the shortest time possible, while maintaining all of your quality goals. Thus all of the structured writing algorithms can contribute to timeliness.

## Quality

The maintenance of quality is key to timely delivery. Any way in which you can use algorithms or structures to assist the writer to produce quality content helps you deliver in a timely manner. In particular, if your structured writing formats are specific about what pieces of information are required and factor out not only formatting but presentation, you can accept new content and publish it very quickly with minimal concerns about maintaining quality.

Authoring	Functional lucidity in structured writing design helps authors work faster and with more confidence. Factoring out any non-content concerns, particularly those related to content management and publishing, will help maintain quality while improving delivery speed.
Separating content from formatting	Formatting content by hand takes time and is subject to unintended variation through human error. If you want to publish quickly and with consistent formatting, you need to factor out the formatting from the content and hand the formatting task over to a formatting algorithm. This will allow you to shorten the time it takes to publish by making formatting and output virtually instantaneous once the writing is finished. It also helps speed up the writing process and improve content quality by avoiding the division of the writer's attention between writing and formatting. However, it is important to make sure that the structures you create to factor out formatting are not more complicated for authors to deal with than the formatting they replaced.
Single sourcing	If you are delivering content to multiple media, you need to avoid having to separately prepare the content for publication in different media. The less manipulation of the content you have to do to output to each target media, the better. If you want to maintain equal quality across different media, taking a differential single sourcing approach will ensure high quality in each media without slowing down the publishing process.
Content reuse	Content reuse can improve timeliness by avoiding the need to create content twice. If a new piece of content can be created by pulling in pieces of existing content, they may allow you to get it out faster. Be careful not to assume that this is an automatic win, however. The reuse process itself takes time. A lot of its payoff comes not from reducing end to end authoring time but from avoiding re-translation of content. Complex reuse systems also require the work to be done

by authors who are conversant with the tools and with the content set (so that they can find reused content efficiently). The availability of those authors may impact your ability to move quickly. Reuse can be a timeliness win under the right circumstances, but don't lose track of the fact that simplicity can often be a key virtue when you need to act quickly.

### Composition

If you are going to use content reuse as part of your timeliness strategy, ensuring the composability of your content becomes doubly important. If you want to deliver reused content quickly, you need the pieces to go together seamlessly and reliably. But don't forget that it is also important to maintain a cogent narrative flow across a piece of assembled content. Avoid composability tactics that may compromise the quality of the finished piece.

### Normalization

Where it can be achieved, maintaining a single source of truth can be a huge win for timeliness. If you can assure that there is only one source for a particular truth (however you define truth for this purpose) then you simply deliver it where required.

### Publishing

Automating publishing can allow you to release content much more quickly. This is particularly important if you are going to release content to multiple media or if the changes you need to make affect multiple pieces of content. If you are going to do continuous publishing successfully, this means that most of your publishing processes have to be hands off and highly reliable. Merely automating the publishing build does little for you if you then have to do extensive manual quality assurance of the output. You need to build reliability into the content and processes from the beginning so that you can confidently press the publication button without needing to look at the output.

### Linking

One of the most challenging aspects of adding new content or removing old content in a timely fashion is managing the links. The new content

should link to all relevant content in the current content set, and the current content should link to the new content where appropriate. When content is removed, all the content that links to it should be updated, either to remove the link or to link to something else. Any structured writing approach that manages links will help with this, but by far the most efficient way to deal with it is to use the soft linking approach detailed in Chapter 15, *Linking*.

### Conformance

Conformance is fundamental to publishing new content quickly while maintaining quality. The faster and more reliably you can ensure the conformance of content to its governing constraints, the faster you can publish it. Structured writing support for the conformance algorithm thus provides a direct benefit to timeliness.

### Auditing

A huge part of timeliness is knowing when new content, or edits to old content, are required. The world changes all the time, but unless you have an efficient way to determine the impact of those changes on your content set, you can't respond to those changes in a timely manner. Taking a structured writing approach to auditing, as outlined in Chapter 17, *Auditing*, can go along way to ensuring that you always know when changes need to be made, and that you can find all of the content that needs to be changed.

### Extract

The extract algorithm is not just a great shortcut for generating content, it is also a great way to keep up with changes in the real world. If the data source that you are drawing content from is updated, simply repeating the extraction and republishing the content will bring it up to date. If you want to be really up to date, you can also run the extract algorithm dynamically, only pulling content from source when the reader requests it. If you have a fully automated publishing process that does not require any user intervention, you can effectively publish live data right out of the source file.

### Merge

In many cases the extract algorithm works with the merge algorithm to combine content extracted from another source with content written by authors. In this case, you can use changes in the source as a flag to tell you when changes in the written content are required.

### Information architecture

Changes to subject matter don't just affect individual pieces of content. They affect the overall information architecture. If you know that your content is going to change on a regular basis, it makes sense to create an information architecture that is highly adaptable to change. The more of your architecture that can be generated by algorithms, the easier it will be to make architectural changes in response to change. But having your information architecture implemented by algorithms requires structured writing techniques that enable the architectural routines to access the aspects of the content they need to organize it. A bottom-up information architecture is particularly effective at handling rapid and constant change. Implementing a bottom-up architecture, which relies heavily on well structured topics connected by rich linking, benefits enormously from structured writing techniques. By changing the way content is organized and linked, these techniques can allow you to add and remove individual pages from a content set without fear of breaking things.

### Change management

While the audit algorithm will alert you to the need for changes in a timely manner, the execution of those changes still needs to be managed in order to respond in a timely manner without compromising quality. Using structured writing techniques to facilitate change management can help ensure that the required changes get made quickly and reliably.

### Content management

Every change in a content set is a content management action. Unless you design your system very carefully, the cost of each content management action will increase as the size of the content set grows, as each action has

to take account of the possible impacts on a greater number of resources. Using structured writing techniques to automate aspects of content management can help avoid the cost of content management escalating as the content set grows and can make each individual content management action easier and more reliable.

### Collaboration

Collaboration can do a great deal to improve the timeliness of content by making more authors available to contribute content and by moving content creation close to the source of information changes. Structured writing can be a double edged sword when it comes to collaboration. By automating publishing and information architecture, it can make it much easier to coordinate the efforts of multiple contributors. By supporting composability, it can make sure that the contributions of different collaborators work well together. And by enforcing content constraints, it can help assure that all contributors create content that does the job it is supposed to do. At the same time, however, many structured writing systems are difficult to learn and use, making it difficult for collaborators to contribute. They can also be expensive, making it uneconomical to include occasional contributors as authors. Both of these problems can be avoided by focusing heavily on the functional lucidity of your structures and markup, which not only avoids the heavy learning curve of structured writing, but removes the need for complex editing tools.

The way in which you select and integrate all of the other structured writing algorithms that you use has a big impact on your ability to deliver in a timely fashion. Optimizing individual pieces may not gain you the time you are looking for, or may not maintain your quality standards while delivering quickly. You need to think careful about where bottlenecks can occur in your overall process and about any points in the chain where complexity may go unhandled and end up being dumped on the reader.

While much of your content structure design activities will be done in terms of the specific algorithms you want to support, a concern with timeliness demands that you think about how those structures affect the timeliness and quality of the entire process. Complex document domain/management domain reuse structures, for instance, may save

time by avoiding duplicate work, but you need to think about the time it takes authors to use these features, and any quality traps that lurk in their use. You also need to think about whether the use of such features could leave you overly dependent on particular writers with particular training to make changes when they are needed. The most technically efficient system is of little value if the only person who can execute it is on vacation. All structured writing systems require special skills in certain key roles, but it pays to avoid setting up your system in a way that creates skill bottlenecks for executing short term content changes.

One of the key features of the subject domain is that subject domain structures support multiple algorithms with the same markup. This can be an enormous benefit in ensuring timeliness. Not only does it simplify the content changes that need to be made to respond in a timely fashion, it hands more of the management and production phases over to algorithms, which are always faster than people. It also means that content updates require no knowledge of the management systems, which helps avoid a skills bottleneck for content changes.

---

# Chapter 26. Repeatability

Content is one of the hardest products to test. One of the biggest barriers to consistently producing quality content is being able to test that quality has actually been achieved – that is, that the content is serving and/or influencing the reader as you want it to do.

The problem has two parts. The first is that it is hard to observe the effect of content. You just aren't there to watch people read it. The second is that even when you do get to observe the effect of content on the reader, how do you know what aspects of the content achieved the effect you observed? Making these observations on individual pieces of content is of limited value. Yes, you can do continual A/B testing, taking down content that does not perform well and continually trying other things until you finally create a piece of content that performs well. But then how do you reproduce that success with the next piece of content you write? How do you determine which aspects of the successful content contributed to its success and should be emulated in future content?

Measuring content is not like measuring minivans or cans of peas, where every example is supposed to be exactly alike. Every piece of content is supposed to be different. It is great to find that you have a successful ad or blog post or manual topic, but to repeat that success, you have to look beyond the particularity of an individual item to see what it has in common with other successful items. Some part of the success of individual items doubtless lies in the features they and they alone possess – your review of a Harry Potter movie is going to get more hits than your review of an art-house flop for reasons entirely unrelated to the quality of your reviews – but much of it also comes from meeting specific user needs in a highly accessible ways, and there is often a pattern to that that can be repeated. Did your review of Harry Potter get more or fewer views than the next site's review? Do your reviews of art-house flops get more views than their's? What is it about how you write your reviews that makes them more popular, regardless of the popularity of the movie? Maybe its your acerbic wit, of course, which is not easy for another writer to reproduce, but maybe it is that you present the information that movie goers really care about in a format that is clear and easy to read. If so, that pattern should be repeated across all your reviews. With recipes, maybe there is a pattern to a recipe that makes it particularly easy for a reader to choose which dish to prepare and to prepare it successfully, and those characteristics should be repeatable across a whole set of recipes. While literary charm doubtless counts for something – and for more in some forms of content than in others – saying the right things in the most accessible way is still the basic bread and butter of content quality, and structured writing is the best way to ensure that you can deliver that repeatably.

Trying to derive lessons from a single piece of successful content is little more than an educated guess. You are abstracting from a single data point. To be certain which aspects of the content are doing the most to achieve it aims, you need to observe multiple samples that exhibit the same basic features. It is when you observe that multiple pieces of content exhibiting the same features are all successful that you know with some degree of reliability that it is those features that make the content work. Once you know this,

you are in a position to reliably produce new content that will be similarly successful. You have achieved repeatability.

To achieve repeatability, you need to partition those elements that you want repeated and constrain your authors to follow them. But is it not enough to constrain content patterns after the fact, when you already know what works. If you don't have many examples of the same pattern to test, you can't draw any useful conclusions about which elements of the pattern are working and which are not. You not only have to constrain in order to repeat, you have to constrain in order to measure. Without constraints, you don't know what you are measuring.

In order to generalize the measurements you are taking, so that you can draw conclusions that apply to more than one piece of content, you need to make sure that each piece of content you are measuring has the same structures and features. That is the only way you can measure if it is the structures and features, rather than the individual texts, that are performing well. You need consistent conformant content structures that express those features of a text that impact is quality. Until you have that, you have no reliable basis on which to extrapolate any measurements you do of individual pieces of content. In order to achieve repeatability in content creation, therefore, you need structured writing.

The most fundamental part of content quality is to give the reader the information they need in a form they can use. A confounding aspect of content quality is that every reader is different. They are doing different thing, they have different experiences, and they have different vocabularies.

For many writers in corporate environments, lack of knowledge about the readers, their tasks, and their backgrounds, is one of the biggest problems in determining what content is required and what form it should take. While direct contact with the customer is undoubtedly the best way to address this, you can actually learn what content to create without ever meeting the reader, as long as you can measure the performance of your content, generalize your results, and repeat the structures that perform best.

If the bread and butter of content quality is to provide the right pieces of information in the most accessible way, and if you have some means of measuring the impact of your content, then you can use strictly constrained patterns and testing to establish which set of information and which presentation of that information works best for your audience. While seeding this process based on knowledge of the reader is obviously to be preferred, ultimately, this is the most reliable way to learn about what content works for your reader, even if you don't actually know why it works.

Without known good patterns and reliable tests, only personal knowledge of the reader will help the individual writer craft content that may meet their needs. But this means that every writer is doing the reader research and the associated information design work every time, and with a limited body of information and few if any opportunities to test their design. If they spend the kind of time that is required to get this research and design work right, then this is a huge overhead for the organization, and a huge amount of

repeated effort. If they don't spend the time to do it right, then the content probably won't meet the reader's needs. As much as we talk about the potential cost savings of information reuse, the potential cost saving and quality improvement from the reuse of known good information design patterns verified by testing is enormous.

Organizations often look to content reuse as the principle source of cost savings in the content system. But as we have seen, content reuse can be quite expensive to implement. Cost savings are not guaranteed, and quality problem can result from an overzealous approach to reuse. Repeatability, which we can think of as the reuse of patterns rather than individual pieces of content, can be a huge time-saver, and one that, if applied correctly, can bring big quality gains at the same time. And the kinds of constrained patterns which provide repeatability also direct the complexity of information design away from contributors, opening the way to greater collaboration by bringing in authors who are experts in the subject matter but not information design, and who do not have the time or inclination to use complex content management or reuse systems. You should consider carefully whether repeatability, rather than reuse, should be the first place you should look to reduce costs in your content system.

Every structured writing domain provides support for repeatability within its domain. The use of a style sheet in a word processor ensure repeatability in the formatting of headings and lists. Document-domain languages like DITA and DocBook provide repeatability in document structures (but only to the extent that they constrain the use of such structures). To get repeatability in the rhetorical structure of content, however, you need to turn to the subject domain.

---

# Chapter 27. Publishing

All structured writing must eventually be published. Publishing structured content means transforming it from the domain in which it was created (subject domain, document domain, or the abstract end of the media domain) to the most concrete end of the media domain spectrum: dots on paper or screen.

Publishing is a complex process, particularly in an environment where you may be practicing single sourcing, content reuse, or many of the other structured writing algorithms we have looked at. The best way to handle that complexity is to partition and distribute it to appropriate people and processes.

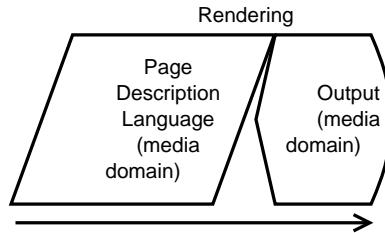
In this chapter, I am going to describe a partitioning of the publishing process into four basic algorithms which I have mentioned in passing in earlier chapters: the synthesis, presentation, formatting, and encoding algorithms. These four stages are formalized in the SPFE architecture, which I will talk about later, but I think they are a fair representation of what goes on in most publishing tool chains, even if those tool chains don't divide responsibilities exactly as I describe them here, or make such clear separation between them as I do.

We have noted that the successful partitioning of a process depends on the ability to communicate everything that is needed for a partition to do its job, and that one hallmark of a successful partitioning is how clear and functional the required communication is. In partitioning of the publication process that I will describe here, each step deals with a different concern and each step produces an output, in the form of a structured document or set of documents, that is structured in terms of the concern it deal with. This then becomes the input to the next step. The structures that these steps produce and consume are stages along the path from the subject to document to media domains. In other words, the publication process is structured to move content through the domains, just as we described in Chapter 2, *How ideas become content*.

## The Rendering Algorithm

There is actually a fifth algorithm in the publishing chain, which we can call the rendering algorithm. The rendering algorithm is the one responsible for actually placing the right dots on the right surface, be that paper, screen, or a printing plate. But this is a low-level device-specific algorithm and no one in the structured writing business is likely to be involved in writing rendering algorithms. The closest we ever get is the next step up, the encoding algorithm.

The rendering algorithm requires some form of input to tell it where to place the dots. In writing, this usually comes in the form of something called a page description language. Like it sounds, this is a language for describing what goes where on a page, but in higher level terms than describing where each dot of ink or pixel of light is placed. A page description language deals in things like lines, circles, gradients, margins, and fonts.



One example of a page description language is PostScript. Here is the PostScript code for drawing a circle:

```
100 100 50 0 360 arc closepath
stroke
```

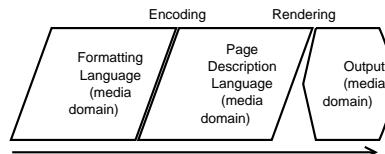
This code is basically moving a virtual pen over a virtual output device; the equivalent of a hand guiding a pen over paper. But it is a much lower-level operation than we need to worry about in structured writing.

## The Encoding Algorithm

Since most writers are not going to write directly in a page description language, the page descriptions for your publication are almost certainly going to be created by an algorithm. I call this the encoding algorithm since it encodes your content in the page description language.

While it is possible that someone responsible for a highly specialized publishing tool chain may end up writing a specialized encoding algorithm, most encoding algorithms are going to be implemented by existing tools that translate formatting languages into page descriptions languages.

There are several formatting languages that are used in content processing. They are often called typesetting languages as well. XSL-FO (XSL - Formatting Objects) is one of the more commonly used in structured writing projects. TeX is another.



Here is an example of XSL-FO that we looked at in Chapter 11, *Single Sourcing*:

```
<fo:block space-after="4pt">
 <fo:wrapper font-size="14pt" font-weight="bold">
```

```
Hard Boiled Eggs
</fo:wrapper>
</fo:block>
```

You process XSL-FO using an XSL-FO processor such as Apache FOP. Thus the XSL-FO processor runs the encoding algorithm, producing a page description language such as PostScript or PDF as an output.

Writers are not likely to write in XSL-FO directly, though it is not entirely impossible to do so. In fact some boilerplate content such as front matter for a book does sometimes get written and recorded directly in XSL-FO. (I did this myself on one project.) But when you are constructing a publishing tool chain, you will need to select and integrate the appropriate encoding tools as part of your process.

The job of the encoding algorithm is to take a high level description of a page or a set of pages, their content and their formatting, and turn it into a page description language that lays out each page precisely. For publication on paper, or any other fixed-sized media, this involves a process called pagination: figuring out exactly what goes on each page, where each line breaks, and when lines should be bumped to the next page.

It is the pagination function, for instance, that figures out how to honor the keep-with-next formatting in an application like Word or FrameMaker. It also has to work out how to deal with complex figures such as tables: how to wrap text in each column, how to break a table across pages, and how to repeat the header rows when a table breaks to a new page. Finally, it has to figure out how to number each page and then fill in the right numbers for any references that include a particular page number.

This is all complex and exacting stuff and depending on your requirements you may have to pay some attention to make sure that you are using a formatting language that is capable of doing all this the way you want it done.

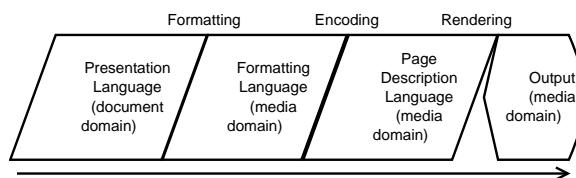
Also, you are going to have to think about just how automatic you want all of this to be. In a high-volume publication environment you want it to be fully automatic, but this could involve accepting some compromises. For example, in book publishing it is not uncommon for writers and editors to make slight edits to the actual text of a document in order to make pagination work better. This is very easy to do when you are working in the media domain in an application like Word or FrameMaker. If you end up with the last two words of a chapter at the top of a page all by itself, for instance, it is usually possible to find a way to edit the final paragraph to reduce the word count just enough to pull the end of the chapter back to the preceding page. This sort of thing gets much harder to do when you are writing in the document domain or the subject domain, particularly if you are single sourcing content to more than one publication or reusing content in many places. An edit that fixes one pagination problem could cause another, and a major reason for writing in those domains is to take formatting concerns off the author's plate.

For Web browsers and similar dynamic media viewers, such as E-Book readers or help systems, the whole pagination process takes place dynamically when the content is loaded into the view port, and it can be redone on the fly if the reader resizes their browser or rotates their tablet. This means the publisher has very little opportunity to tweak the pagination process. They can guide it by providing rules such as keep-together instructions through things like CSS, but they obviously cannot hand tweak the text to make it fit better each time the view port is resized.

The formatting language for these kinds of media is typically Cascading Style Sheets (CSS).

## The Formatting Algorithm

The job of the formatting algorithm is to generate the formatting language that drives the encoding and pagination process. The formatting algorithm produces the media domain representation of the content from content in the document domain.



In the case of HTML output, the formatting algorithm generates HTML (with connections to the relevant CSS, JavaScript, and other formatting resources). This is the end of the publishing process for the Web, since the browser will perform the encoding algorithm internally and the computer operating system will likely take care of the rendering. In the case of paper output, the formatting algorithm generates a formatting language such as TeX or XSL-FO which is then fed to the encoding algorithm as implemented by a TeX or XSL-FO processor.

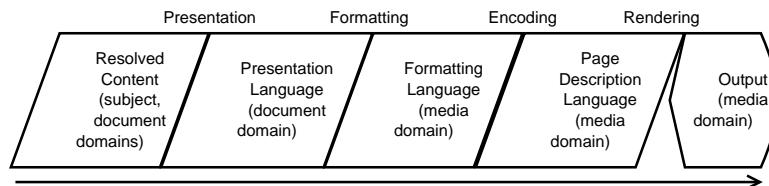
In some cases, organizations use word processing or desktop publishing applications to tweak the formatting of the output by having the formatting algorithm generate the input format of those applications (typically RTF for Word and MIF for FrameMaker). This allows them to exercise manual control over pagination, but with an obvious loss in process efficiency. In particular, any tweaks made in these applications are not routed back to the source content, so they will have to be done again by hand the next time the content is published.

This algorithm is usually the province of publication designers. One of the most elementary structured writing algorithms is to separate content from formatting (Chapter 9, *Separating Content from Formatting*) which means removing formatting as one of the writer's concerns. Almost every structured writing implementation will involve writing formatting algorithms, however. Even if you use off-the-shelf languages like DITA or DocBook, you will have to write or modify formatting algorithms to get

the specific formatting you want. We looked at some examples of basic formatting algorithms in Chapter 10, *Processing Structured Text*.

# The Presentation Algorithm

The job of the presentation algorithm is to determine exactly how the content is going to be organized as a document. A pure document domain document is a representation of the presentation of the content. The job of the presentation algorithm is to produce a pure document domain version of the content. This may mean producing the entire presentation from purely subject domain content, or simply handling the occasional subject domain structure in a largely document domain file.



The organization of content involves several things:

- |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Ordering | At some level, content forms a simple sequence in which one piece of information follows another. Authors writing in the document domain typically order content as they write, but if they are writing in the subject domain, they can choose how they order subject domain information in the document domain.                                                                                                                                                                                  |
| Grouping | At a higher level, content is often organized into groups. This may be groups on a page or groups of pages. Grouping includes breaking content into sections or inserting subheads, inserting tables and graphics, and inserting information as labeled fields. Authors writing in the document domain typically create these groupings as they write, but if they are writing in the subject domain, you may have choices about how you group subject domain information in the document domain. |
| Blocking | On a page, groups may be organized sequentially or laid out in some form of block pattern. Exactly how blocks are to be laid out on the displayed page is a media domain question, and something that may even be done dynamically. In order to enable the media domain to do this, however, the document domain must clearly delineate the types of blocks in a document in a way that the formatting algorithm can interpret and act on reliably.                                               |
| Labeling | Any grouping of content requires labels to identify the groups. This includes things like titles and labels on data fields. Again, these are typically created by authors in the document domain, but are almost always factored out when authors write in the subject domain (most labels                                                                                                                                                                                                        |

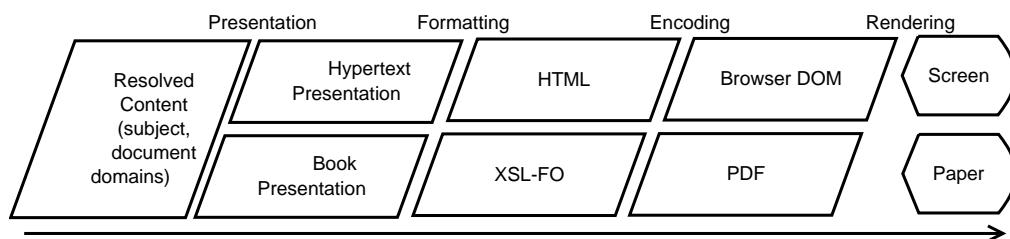
indicate the place of content in the subject domain, so inserting them is a necessary part of reversing the factoring out of labels that occurs when you move to the subject domain).

### Relating

Ordering, grouping, blocking, and labeling cover organization on a two dimensional page or screen. But information is related to other information in complex ways which we can express by creating non-linear relationships between pieces of content. This includes hypertext links and cross references.

## Differential presentation algorithms

As we saw in Chapter 11, *Single Sourcing*, the organization of content is an area where the document domain cannot ignore the differences between different media. Although the fact that a relationship exists is a pure document domain issue, how that relationship is expressed, and even whether it is expressed or not, is affected by the media and its capabilities. Following links in online media is very cheap. Following references to other works in the paper world is expensive, so document design for paper tends to favor linear relationships where document design for the web favors hypertext relationships. This is an area, therefore, in which you should expect to implement differential single sourcing and use different presentation algorithms for different media.



## Presentation sub-algorithms

Many other structured writing algorithm, are executed as part of the presentation algorithm. Among them:

### The linking algorithm

How content is linked or cross-referenced is a key part of how it is organized in different media, and a key part of differential single sourcing. We looked at the linking algorithm in detail in Chapter 15, *Linking*.

### The information architecture algorithm

Part of the presentation of a document or document set is creating the table of contents, index, and other navigation aids. Creating these is part of the presentation process.

Because these algorithms create new resources by extracting information from the rest of the content, it is often easier to run these algorithm in serial after the main presentation algorithm has run. This also makes it easier to change the way a TOC or index is generated without affecting your other algorithms. For more on the information architecture algorithm, see Chapter 21, *Information Architecture*.

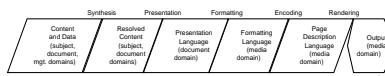
## The Synthesis Algorithm

The job of the synthesis algorithm is to determine exactly what content will be part of a content set. It passes a complete set of content on to the presentation algorithm to be turned into one or more document presentations.

Among other things, the synthesis domain resolves all management domain structures in the content (unless some are to be retained for downstream post-publication algorithms to work with). This means that it processes all inclusions and evaluates all conditions. The result is document domain or subject domain content with all of the management structures removed and replaced with the appropriate document or subject domain structures and content.

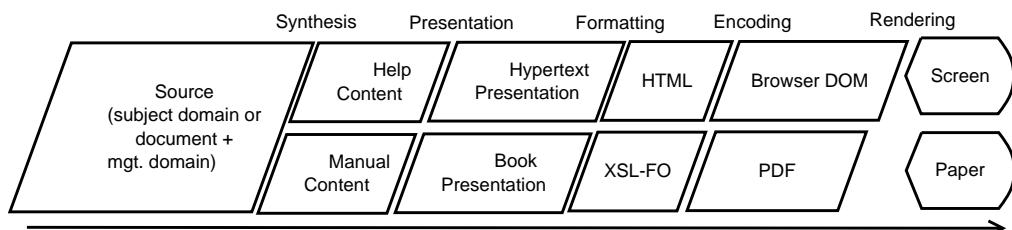
In the case of document domain content, processing the management domain structures yields a document-domain structure which may then be a pass-through for the presentation algorithm (that is, the document domain markup may already express the desired presentation).

In the case of the subject domain content, processing management domain structures yields a definitive set of subject domain structures which can be passed to the presentation algorithm for processing to the document domain.



## Differential synthesis

We noted above that you can use differential presentation to do differential single sourcing were two publications contain the same content but organized differently. If you want two publications in different media to have differences in their content, you can do this by doing differential synthesis and including different content in each publication.



# Synthesis sub-algorithms

A number of structured writing algorithms are executed at the synthesis stage. In order to keep things well partitioned, it is often advisable to execute each one as a separate sub step in the synthesis stage.

## The reuse algorithm

Pulling in reused content is part of the synthesis process. We looked at the reuse algorithm in Chapter 12, *Reuse*.

## The extraction algorithm

In some cases you may wish to extract information from external sources to create content. This can include data created for other purposes, such as application code or data created and maintained as a canonical source of information, such as a database of features for different models of a car. Extraction is part of the synthesis process. See Chapter 19, *Extract*.

## The merge algorithm

In one sense, every structured writing algorithm is a merge algorithm. As we saw in Chapter 10, *Processing Structured Text*, most algorithms consist of factoring into content information or metadata that was factored out as we moved the point of recording from the media domain to the document domain and the subject domain. But it is also possible, and often useful, to combine information from different sources to create a new set of content. Merging content is done at the synthesis stage of publication. See Chapter 20, *Merge*.

## Deferred synthesis

For static presentation, all synthesis happens before the material is presented. But if you are presenting content on the web, you can defer parts of the synthesis algorithm to the browser, which can synthesize and present content by making calls to web services or other back-end data source, or by making a request to code running on the server to synthesize and present part of the page. We looked at this in Chapter 28, *Active content*.

## Combining algorithms

As we have seen, structured writing algorithms are usually implemented as sets of rules that operate on structures as they encounter them in the flow of the content. Since each

algorithm is implemented as a set of rules, it is possible to run two algorithms in parallel by adding the two sets of rules together to create a single combined set of rules that implements both algorithms at once.

Obviously, care must be taken to avoid clashes between the two sets of rules. If two set of rules act on the same structure, you have to do something to get the two rules that address that structure to work together. (Different tools may provide different ways of doing this.)

In other cases, though, one algorithm needs to work with the output of a previous algorithm, in which case, you need to run them in serial.

In most cases, the major algorithms (synthesis, presentation, formatting, encoding, and rendering) need to be run in serial, since they transform an entire content set from one domain to another (or from one part of a domain to another). In many cases the sub-algorithms of these major algorithms can be run in parallel by combining their rule sets since they operate on different content structures.

## Architecture of a publishing tool chain

Overall the architecture of a publishing tool chain should be designed to facilitate the partitioning and distribution of complexity in the content system. Almost all of the structured writing algorithms we have talked about are executed at some point in the publishing process. (The exceptions are those that are concerned with aspects of conformance, which operate on the content store independent of any publishing events.)

The partitioning of the system needs to make sure that information is passed from one partition to another (whether the task of that partition is executed by a person or an algorithm). This means that the execution of that partition needs to happen at the right time and place – the time and place in which the information it needs is available, and before the time and place where the information it produces is needed. The SPFE structure – Synthesis, Presentation, Formatting, Encoding – seems to provide a good general framework into which the various algorithms and their timing fit reasonably well. However, you should not get hung up on either the names or the order of operations that this architecture describes. The goal is to partition and distribute the complexity of your content system so that none of the complexity goes unhandled and no one person or process is asked to handle more of the complexity than they have the time and resources to manage. Any architecture that accomplishes that for you particular content system is a good architecture.

A significant chunk of the complexity of the content system is the complexity of the publishing process itself, and the creation and management of all its working parts. The creation of reliable code to implement each of the structured writing algorithms is a complex task, and the partitioning and distribution of that complexity should be a major design and/or tool selection consideration.

Two keys to partitioning the complexity of code are to keep code units small and simple, and to reuse code when you can. It is a sound principle that each piece of code should do one thing and one thing only. Taking a pieces of content written in the subject domain or the document domain, particularly with some management domain added in, is a complex business. As we have seen, it involved the execution of many algorithms. Writing a single program to execute all of those algorithms at once would be complex and would violate the principle of simplicity and would leave little room for the reuse of code.

Every step in a content publishing chain reads in one or more structured content files and produces one or more structured content files in a different domain, or, at least, nearer to the media domain and dots on a page than the input file. There are two ways to separate the algorithms of your publishing chain.

- If two algorithms work on non-overlapping structures in the content file, you can run the two sets of rules in parallel. If you use tools that implement the rule-based model of content processing that is described in this book (tools such as XSLT) then you can simply include the two rule sets in one program and run the program on the source file. The output of this process should be a new structured content file nearer to the media domain.
- If an algorithm works on the whole source file, create a single program and run it on the source file, creating a structured content file that will be the input to the next algorithm.

The implication of this is that different algorithms work on content at different points on the continuum from the subject domain to the media domain, which is, of course, the case. But as we have noted in many of the algorithms we have looked at, there is a way to execute the algorithm in any of the structured writing domains. As far as processing goes, the linking algorithm for subject domain content is different from the linking algorithm for document domain content. But if you use subject annotation rather than link markup in your content, this does not mean you use the subject domain linking algorithm instead of the document domain linking algorithm. It means you use the subject domain linking algorithm and the document domain linking algorithm, one after the other. At least, if you create a layers publishing architecture like the one I have outlined here, the subject domain linking algorithm create document domain linking markup as an output, which is then processed by document domain linking markup to produce media-appropriate links in the media domain. (Which means you don't need several media-specific versions of the subject domain linking algorithm.)

In Chapter 10, *Processing Structured Text* we saw examples of algorithms that moved subject domain content to the document domain, and other algorithms that then moved that same content from the document domain to the media domain for publishing. This is the template of all publishing processes. In the real world, though, you may have more than one intermediate stage in the journey. This architecture is sometimes referred to as a publishing pipeline. Each step in the publishing process is kept simple by only doing one job and then passing the content on to the next step in the pipeline.

This architecture means that each individual program you have to write is relatively simple and straightforward, which makes it much easier to design, write, debug, and maintain. Many small self-contained programs operating as a pipeline will usually be both cheaper to create and maintain and more robust than a single monolithic program that tries to do everything in one step.

The pipeline approach also allows for a great deal of reuse of existing code. For example, if your content is written in the subject domain, you need to get it all the way to the media domain for publication. But along that road to the media domain, it passes through the document domain. There are a number of robust document-domain publishing tool chains available: DocBook, DITA, reStructuredText, LaTeX. You could write a presentation algorithm to transform each of your subject domain structures into any one of these formats and then simply use their existing tool chains for the rest of your publishing pipeline.

Other opportunities exist for code reuse at a more granular level. For instance, all of your subject domain structures are going to need paragraphs and lists and other basic text structures, as well as their individual subject-specific structures, and the same subject annotations are likely to occur across your whole content set. You can use the same definitions of basic text structures and subject annotations across your entire content set, and write just one rule set for those text structures and subject annotations that is used everywhere, greatly reducing the amount of code you have to write.

---

# Chapter 28. Active content

When you publish to electronic media, you can create active content, that is, content that has behavior as well as formatting. Some examples:

Personalized content

You can select and arrange content for individual readers based on the things you know about them. For instance, if you are logged into Amazon it customizes elements of every page based on your previous purchases, your wishlist, and things you previously browsed.

Dynamic arrangement

Part of the presentation algorithm is arranging content on the page or screen, but with online media you can allow the reader to arrange the content. For instance, you can publish tables that readers can sort for themselves.

Adaptive content

Similarly, you can create content that adapts itself dynamically to the view port in which it is displayed. For instance, displaying in multiple columns on a wide view port, and in a single column on a narrow one.

Progressive disclosure

You can present content in a way that only shows part of the content on the screen initially but reveals more when the user clicks on a link or takes another action. For instance, you might show the high-level of a procedure and provide a link that opens detailed steps for those who need them. This is a way to cater to audiences with different levels of preparedness.

Feeds and dynamic sources

You can include content that comes from an external source which updates independently of your content such as a feed or a web service.

Interactive media

You can include apps, widgets, and other media that the user can interact with.

Active content is simply the postponement of one or more of the structured writing algorithms to the time of reading. Personalized content is executing the synthesis algorithm on the server when you request a page, using personal details as query terms to select content for the individual. Changing the layout of the page when you resize your browser or rotate your phone is executing the presentation algorithm in the browser.

Allowing the reader to change the font size is allowing the reader to configure and execute the formatting algorithm in the browser. Pulling a live stock quote into a page that mentions a company is having the web server or the web browser executes the synthesis algorithm when the page is served/displayed.

In principle, therefore, you support active content in exactly the same way you support those algorithms in any form of structured writing: by creating your content in the appropriate domain and with the appropriate structures to reliably support the algorithms you want to run. Reliability is key here, of course. If you are creating static books or web pages you may be able to get away with reviewing the results before they are published and even hand tweaking the output to fix any issues, but you can't do that with active content. You need to be able to trust the algorithm 100%, which means you need to trust the content it is working on 100%. This makes the conformance algorithm crucial to an active content project, and, as we have seen, different domains support the conformance algorithm with varying degrees of reliability.

Another factor to consider, though, is that while active content is essentially just postponing the point at which you execute various structured writing algorithms until the content is in the user's hands, you don't necessarily want to postpone running those algorithms on the whole page. You usually only postpone them for the parts of the page that you want to make active. For instance, if you want real-time stock quotes embedded in your business stories whenever a company is mentioned, there is no reason to postpone the synthesis, presentation, and formatting of the entire page; you only need to postpone the synthesis of the stock quote itself. And if you want that quote to continue to update in real time while the page is displayed, you certainly don't want to be redrawing the whole page every time the price changes.

The display of stock quotes depends on subject domain metadata that unambiguously identifies the company mentioned in the content. For example:

{Microsoft} (company "NASDAQ:MSFT") is a large software company.

In a normal static publishing flow, this subject domain information would be resolved into a static piece of content by the presentation algorithm:

**Microsoft** US\$60.29 +0.34 (+0.57%) is a large software company.

But for active content the main presentation algorithm needs to pass through the subject-domain information to the browser so that that bit of synthesis can be executed in real time when the content is displayed.

This means that only those parts of the content that you want displayed as active content need to be captured in the structured writing domain, and in the specific structures, required to reliably run the applicable algorithms. The rest of the content can be in a different domain. In practice, this usually means that you have subject domain structures

to support the active content embedded in document domain structures. Of course, having all your content in the subject domain is also an option, in which case you would perform the presentation algorithm on the content you want to deliver statically, while passing on the subject-domain representation of the content you want to be active to the system that will perform the active content algorithms.

However, this is not necessarily, or even likely, to be a straightforward pass through. The downstream system that is going to execute the active content algorithm, such as a JavaScript app running in the web browser, may expect or require the subject domain information it uses in a particular format. For instance, you might have to embed the ticker symbol in a call to a function that returns stock quotes:

```
<p>Microsoft ()
is a large software company.</p>
```

So your server side presentation algorithm needs to transform your subject domain annotation into this format.

```
match company
 continue
 $fnccall = 'getStockQuote(' + @specifically + ')'
 output ' ('
 create span
 create attribute onload = $fnccall
 output ') '
```

The document domain represents the desired presentation of a piece of content. Therefore the only kinds of active content algorithms that you can execute on ordinary document domain structures are those that leave the presentation alone and only affect the formatting. For example, you can allow readers to select the font size of a web page, which is a purely formatting concern.

If you want to do active content that dynamically changes the presentation of the content, or that changes the text of the content (as in our stock price example) then you need either subject domain structures or document domain structures that are explicitly designed to support active content algorithms.

For example, a generic table structure does not support the action of allowing the reader to sort on any column. Sorting by column only makes sense if the content is inherently sortable.

item	legs	price
table	4	\$400
stool	3	\$20

shooting stick	1	\$75
chair	4	\$60

For example, in the table above the columns are sortable. The reader could choose to sort the table on the item name, the number of legs, or the price, all of which might be useful arrangements, based on their needs.

But consider what might happen if you sorted this table on the second column:

1.	Don protective clothing.
2.	Clear the area.
3.	Block all entrances.
4.	Activate the destruct sequence.

Thus you can't just decide to make all the tables on a page sortable by column. Unless the document domain structure explicitly states which columns of a table are sortable and which are not, you can't implement this kind of active content.

To implement column sorting at the document domain level, therefore, you need some sort of sortable table structure which assures that the sorting behavior is only applied to columns where it makes sense in tables where it makes sense.

While the document domain typically requires different markup for different algorithms, the subject domain typically does not. In the subject domain you capture the semantics of the subject matter, which are the same no matter what algorithms you are applying to the content.

In the subject domain, the product list would be a structured data set with known semantics (probably maintained as a separate database).

```
products:: item, legs, price
table, 4, $400
stool, 3, $20
shooting stick, 1, $75
chair , 4, $60
```

Knowing what the semantics are, you would know whether that data set is sortable and therefore whether it can be presented as a sortable table.

Creating your content in the subject domain gives you the greatest flexibility to generate active content in ways that are appropriate to the subject matter and the device. And because the subject domain does not require any different structures to support active content, your writers don't have to understand or even think about how the active content might work, effectively partitioning the complexity of active content from your writers.

As we have seen before, an additional benefit of partitioning content from its presentation is that it allows you to experiment with different forms of presentation. This allows you to test if active content is working or not or change the form of active content you use, all without involving authors or changing your content.

This does not mean that active content is always going to be a free gift of the subject domain. Apart from the fact that you still have to design and implement the content behavior, you are going to have to make sure that you are capturing the subject domain semantics that are needed to drive the active content behaviors you want. Those semantics may not be different in kind from any other subject domain semantics, but you may need to break things down in greater detail than you might have to for other algorithms. You will also need to make sure that you get a high degree of conformance to these structures from your writers, as it is difficult to validate the correct operation of every active content algorithm on every content set at run time. The success of your active content strategy is going to depend heavily on the quality and consistency of your input data.

---

# Chapter 29. Translation

Many organizations deliver content in multiple languages. Translation is a major source of complexity for those organizations. It is not only the complexity of doing the translation, but the complexity of integrating the translation process into the overall content process, and of integrating the translated content into the publishing process. Translation is a huge subject in its own right and I have neither the space nor the expertise to do it full justice. Therefore I will make just a couple of points about how structured writing can help partition and transfer certain aspects of the complexity of translation.

## Extracting content for translation

Translators generally do not work in the original source format of the content. A translation agency works with many customers and thus content in many file formats, so it would be onerous for translators to be fluent in all of them. Besides, the translators don't need any of the features of these programs. In many cases, though, they do need features specific to translation, such as access to translation memory. Thus text to be translated is often extracted from its source format, translated in a separate tool, and then reinserted into the original source application.

There is an XML standard that is used to do this round-trip, called XLIFF. It records the metadata necessary to extract and replace the content in its original format after publishing. The ability of your source format to work with XLIFF can therefore be an important consideration. (XLIFF represents a neat partitioning and transfer of the complexity of file formats in translation.)

If your source format is a desktop publishing format, however, inserting the translated content back into the source format may leave the layout looking wrong since the translated text may be longer or shorter, or may require a completely different layout from the original text. This need to redo the formatting of the translated content can add significantly to translation time and costs. Using a structured writing format that factors out the the formatting can reduce these costs by allowing each translated version to be formatted algorithmically. But as we saw in Chapter 9, *Separating Content from Formatting*, there are various degree of separation that you can achieve using different kinds of structure to factor out more of the formatting. The different presentation and formatting needs of different languages and cultures could require a higher degree of separation that you might otherwise have created, so make sure to account for this when choosing which content structures to use.

## Avoiding trivial differences

Trivial differences in how the same thought is worded make little difference to a reader, but they can run up translation costs. Structuring content to avoid these trivial differences can reduce translation costs.

There are two principle ways you can do this with structured writing. The first is to reuse the same piece of content every time the same thought is expressed. As we have seen, though, this approach comes with complexities of its own. The smaller you make the pieces of content you reuse, the more reuse you can get, but the more pieces you have to manage and the more pieces the writer has to look through to find reusable content.

The second is to factor out the content that repeats the thought. We have seen examples of this a number of times in this book. As we moved our recipe example into the subject domain, we factored out the titles of the recipe sections, eliminating any trivial differences in the titles writers might use. In Chapter 12, *Reuse* we factored out a repeated safety warning by adding a `is-it-dangerous` field to our procedure structure. This not only removes the need for authors to think about the reuse of the warning, it also factors out any issues regarding the file name for the different language versions. (Something similar could be achieved using the keys approach we also looked at in that chapter.)

## Isolating content that has changed

When a text is revised, such as when you bring out a new version of a product, some of the text is changed, but much of it remains the same. You can save time and money on translation if you only translate the content that has changed. To do this, though, you need a way to isolate the content that has changed from that which has not, and a way to integrate them again after the changed material has been translated. Structured writing allows you to clearly define structures in your content set and deliver just those structures that have changed.

## Continuous translation

If you write a book and, when it is finished, send it for translation, you translated version will be ready to release quite a long time after your first language version is ready. This delay can represent serious missed opportunities. To bring out first language and translated versions simultaneously, or close to it, you need the translation to occur simultaneously with the development of the first language content.

Continuous translation is not just about getting all version of a book released at the same time. On the web, you may be releasing new pieces of content into your content collection every day. If you are maintaining translated versions of that content, you want them to go out at the same time. You may also want to provide links between the different language version of the same content as searches and links do not always send readers to their own language version of your content. A structured approach to publishing and information architecture can accommodate these requirements.

---

## **Part III. Structures**

---

# Table of Contents

30. Rhetorical Structure .....	234
Presentation-oriented rhetorical structures .....	239
Rhetorical metamodels .....	241
Meta-models vs generic models .....	243
Making the rhetorical structure explicit .....	244
Structure and repeatability .....	245
The importance of the rhetorical model .....	247
31. Mechanical Structure .....	248
Flat vs. nested structures .....	249
Annotating blocks .....	252
Agreeing on names .....	252
Different rules for intermediate languages .....	253
Secondary structures of interpretation .....	254
Child blocks vs. additional annotations .....	256
32. Blocks, fragments, paragraphs, and phrases .....	262
Semantic blocks .....	262
Information typing blocks .....	265
Rhetorical blocks .....	267
Granularity .....	270
Fragments .....	270
Paragraphs and phrases .....	271
33. Wide Structures .....	274
Tables .....	274
Alternatives to tables .....	279
Alternate presentation .....	279
Subject domain structure .....	279
Record data as data .....	280
Code .....	280
Pictures and graphics .....	281
Inline graphics .....	288
34. Subject domain structures .....	290
Keep it simple and lucid .....	296
35. Metadata .....	298
The recursive nature of metadata .....	299
Where should metadata live? .....	300
Ontology .....	302
36. Terminology .....	304
Top-down vs. bottom-up terminology control .....	305

---

# Chapter 30. Rhetorical Structure

Throughout the section on algorithms we have talked a lot about structure. Structure is how you constrain content so that you can partition and redirect the complexity of the content system to algorithms. Successful partitioning depends on the ability to transfer the necessary information between partitions without loss. This is the role of structure. So lets look at the elements of structure and how you go about designing a structure to support the partitioning of complexity you want to achieve for your content.

I want to repeat that phrase, designing a structure to support the partitioning of complexity, because it is very important. The worst thing you can do when setting out to design a structure for structured writing is simply to look at the content and try to mark up the structures you find there. That is not likely to give you a structure that supports the algorithms you want to run or that partitions responsibilities appropriately in your content system. The mechanical structure of content is designed and imposed for one reason and for one reason only: to partition the complexity of the content system. If you design your structures without thinking about the partitioning of the complexity of the content system, you will still end up partitioning it, but you may not partition it in a way that is advantageous for your writers, your readers, and the other players in your content system.

That said, the place to begin when designing a content structure is rhetoric, with the way that a piece makes its argument or assembles and organizes information to inform the reader. The work of the writer is rhetoric and the point of partitioning the content system is, in large part, to allow the writer to focus on rhetoric, while still providing the information needed for the other partitions in the content system to operate without dropping complexity.

Achieving an effective rhetoric in the content we present to readers is the ultimate aim of any content system. The result of any failure to handle complexity in the content system is felt largely in the form of impaired rhetoric. (Impaired ergonomics – formatting that is hard to read for instance – is a secondary source of difficulty for the reader.) When we factor out formatting in the transition to the document domain, it is to allow for a greater focus on rhetoric. When we factor out or enforce rhetorical constraints in the transition to the subject domain, it is to achieve a higher level of conformance, validity, and repeatability in rhetoric. The movement from the media domain to the subject domain is a movement from appearance to rhetoric. The aim of the content system is always to produce better rhetoric, and therefore applications of structured writing are about improving rhetoric, either by removing the complexities that detract from writers focusing on rhetoric or by explicitly supporting the consistent and repeatable development of superior rhetoric.

Whether you decide to model a specific rhetorical structure for your content or whether you decide to factor out a specific rhetorical structure by recording facts independent of

presentation so that you can generate a differential rhetorical structure with algorithms, or so that you can test and tweak aspects of the presentation without having to rewrite content, the the place to start in designing content structures is by defining the rhetorical pattern or patterns you want to deliver.

The rhetorical structure of a piece of content is how it tells its story. For many types of stories, the optimal rhetorical structure is quite consistent and often well known. In other cases, the best rhetorical structure can be determined both by a careful consideration of what needs to be said and by experience and testing with readers. In other words, there is a right way, a best way, to tell a wide range of stories. This best is not necessarily universal. Best for your subject matter and your audience may be different from best for mine, just as the best recipe presentation for *Wine Weenie* is different from the best format for *The Teetotaler's Trumpet*, but for you and your readers there is a definable, testable, repeatable best.

Maintaining a consistent high-level of conformance to the optimal rhetorical structure for a particular subject can be complex, however, especially if you have a variety of contributors. Content quality is greatly enhanced when the rhetorical structure is well defined and followed consistently. Lapses of quality occur then the complexity of maintaining the optimal rhetorical structured for content is not properly managed and the unmanaged complexity falls through to the reader in the form or content that is not rhetorically optimal. Also, a well defined rhetorical structured provides an effective baseline against which to compare and measure proposed improvements. Using an explicit predefined rhetorical structure helps enhance and maintain content quality.

We can see rhetorical structure as made up of information requirements and presentation requirements – what needs to be said and how best to say it. Sometimes information requirements dominate the structure and sometimes it is presentation requirements. Sometimes there is no regular set of facts to relate across all instances, but a particular approach to presentation is known to work particularly well.

The recipe example that I have used so often in this book is an example of both information requirements and presentation requirements.

The presentation requirements for recipes in general are the listing of ingredients, including the specific and precise way that the measurements of those ingredients are presented, and the list of presentation steps. Information requirements for recipes produced by particular publications include wine matches, prep time, servings, and number of calories.

One of the key presentation requirements is the presentation of the list of ingredients as a separate section with precise measurements. Yes, this is also an information requirement. But notice that all recipes (or almost all) break out this list and separate it from the preparation steps. The ingredients are mentioned again in the preparation steps, so why not just put the measurements in the steps and omit the listing of ingredients? This is an important rhetorical decision. It is not like deciding whether to pull out the prep time and

number of servings into separate fields or leave them in the introduction. It has a much more important rhetorical purpose.

We separate out the list of ingredients for a recipe, because the cook's first step in making a dish is not the first item in the list of preparation steps. The first step is to make sure that you have all the ingredients you need. The rhetorical structure of a recipe could be to make this an explicit first step:

recipe: Hard Boiled Egg

introduction:

A hard boiled egg is simple and nutritious.

preparation:

1. Make sure you have the following ingredients on hand:
  - \* 12 eggs
  - \* 2qt water
2. Place eggs in pan and cover with water.
3. Bring water to a boil.
4. Remove from heat and cover for 12 minutes.
5. Place eggs in cold water to stop cooking.
6. Peel and serve.

But since this is a universal first step for every recipe, it has been implicitly factored out in the rhetorical pattern of a recipe and is supported by a separate list.

There are other implicit steps as well. Another is to collect and measure your ingredients before you start cooking. Some cooks measure and lay out all their ingredients before they start cooking. Others measure and add as they go. By simply providing a list of ingredients rather than making this an explicit step, a recipe accommodates both behaviors.

In fact, there is a step that is prior to all of these: the step of deciding whether to make the dish or not. This is why most recipes have some form of introduction to the dish and a picture of what the finished dish or a typical serving will look like. Obviously no one is ever going to start a recipe with "Step 1. Decide if you want to make this dish". But that decision is actually the first step a cook makes when they come to a recipe, and the rhetorical structure of the recipe is designed to support that step. (There is an important generality here about what supporting a task means. Tasks are things people do. Procedures are text structures. Supporting a task is a far broader rhetorical problem than writing a procedure.)

It is also interesting what a recipe does not do. A recipe does not call out a list of the pots and pans and other utensils you will need. Sometimes the instructions will be quite explicit about what kind of pot or spoon to use for a step, but nonetheless, most recipes do not list tools the way they list ingredients, perhaps because they assume all kitchens are similarly equipped. Knitting patterns, which are highly similar rhetorical structures

to recipes, and frequently consumed by the same people, tend to explicitly call out the specific tools to use. Whatever the reason, this is as much a defined and established part of the rhetorical structure as recipes are in inclusion of a specific list of ingredients. It is the established way in which information on the preparation is presented, blessed and approved by generations of cooks.

But while there is a general rhetorical consensus about the general pattern of recipes, individual organizations, may require more specific rhetorical structure. As we saw in Chapter 5, *Writing in the Subject Domain*, a wine magazine may require every recipe to have a wine match. A health-oriented magazine may require every recipe to contain a complete set of nutritional information. Other organizations may have specific requirements about how recipes are to be presented, such as requiring ingredients to be presented in a table rather than a list. The specific set of requirements of an organization – their unique constraints – constitute a formal content structure.

In some cases, rhetorical structures are immediately obvious because they have a visual shape. The various components of a recipe just happen to look physically different on a page (which is why recipes are the most popular structured writing example). There is the picture; the introduction, which is a block; the ingredients, which are a list; and the steps, which are a numbered list. The recipe pattern is visually distinctive even without looking at a word of the text.

However, rhetorical structures are not about elements that are visually distinct. They are about the different types of information that are required, the way they are expressed, and the order they are presented in. There may be considerable variation in the second two properties. Whether we would count these variations as options within one rhetorical pattern or as defining different rhetorical structures should probably depend on their effect. Any organization and means of expression that has the same rhetorical effect we can reasonable count as variations on a single rhetorical structure.

When you look at a page that appears to be just a sequence of paragraphs with perhaps some subheadings thrown it, it is easy to assume that there is no particular rhetorical structure present. But this is not necessarily true at all. If a consistent set of information is being presented for a particular purpose, and we can find (or reasonably imagine) that same set of information being assembled for the same purpose to describe another object of the same type, then we have a rhetorical structure. Similarly, where there is a deliberate strategy for laying out an argument or demonstrating or supporting a process, we have rhetorical structure. And where we have a repeatable rhetorical structure, we can define a structured content type for a specific business purpose.

By structured content type here I mean a set of formal computable structures into which text is inserted and by which text – its creation and interpretation – is constrained. This is not quite the same thing as the mechanical structure of the content, which we will look at in Chapter 31, *Mechanical Structure*. The mechanical structure is the implementation of the structured content type, but the structured content type can be described independent

of how it is implemented, and can indeed be implemented in more than one way. You can even define a structured content type without creating or using a mechanical structure to implement it, though obviously with the mechanical structure, you can hand over any part of its validation or processing to algorithms.

You should not expect that, once you define a structured content type, you will be able to take the mechanical structure that implements that type and simply wrap it around every existing piece of content without changing a word. (The point of structured writing is to improve content, not to faithfully represent its current state.) Content that has been written in an unstructured format, even if it obeys a consistent rhetorical structure, may not fit the precise structure you have defined, particularly if you have factored out certain parts of the desired presentation when structuring your content in the subject domain.

What you will find when you start to move content that generally follows a rhetorical structure into a defined structured content type is that a lot of the content does not fit the defined type particularly well. You will find some instances that only partially fit, but which omit information commonly found in the rhetorical structure (and perhaps required in the structured type). You will find that some instances contain information not found in most instances of the pattern, and not supported by the structured type. You will find information not expressed in the way that the structured type expects.

These discoveries mean one of four things:

- The discernment of rhetorical structures is incorrect and you are trying to make content with a different pattern fit your structured type. You need to define a new structured types for this new rhetorical structures.
- The definition of the structured type is incorrect. You need to modify the structured types to more correctly reflect the rhetorical structures.
- The content is a variation of the rhetorical structures that is deliberately not supported by the structured type. The content needs to be edited to fit.
- The content is deficient. It does not meet the rhetorical structure and it needs to be upgraded so that it fulfills its purpose correctly, as defined by the structured type.

Interpreting the mismatch between existing content and the structured type can make or break your entire structured writing project. There is a huge temptation to treat existing text as canonical and try to shape the model to fit it. But as I have stressed several times, the purpose of structured writing is not to represent existing texts, but to partition and redistribute the complexity of the content system, ultimately resulting in better content for the reader. If your current content processes are so good that all your existing content fits your new structures perfectly, then you are not realizing any gain in content quality and you are wasting your time by adding additional mechanical structure. Finding content that does not fit the model is not a sign that the model is broken, but that the process is working.

This does not mean that the models never need to be changed. But it does mean that you change the models to match the things you discover about the best rhetorical structure for your content to achieve your business goals, not to make your existing content, or even the new content that authors want to write, fit the model.

This means that applying structure to your existing content is not a trivial or mechanical task. The purpose, after all, is to improve the quality of the existing content, and that is going to mean additional research and writing work to bring the content up to standard.<sup>1</sup>

## Presentation-oriented rhetorical structures

In some cases, as we have seen, it is possible to factor out the rhetorical structure of an item and move the content almost entirely into the subject domain. This is possible with a recipe, for instance (see Chapter 5, *Writing in the Subject Domain*). In these cases, any reasonable rhetorical design can be created from the subject domain content by the presentation algorithm because the rhetorical design consists of a particular arrangement of facts.

But not all rhetorical models are reducible to an arrangement of facts. This is clearly true of philosophical essays and even of books like this one. In works of this sort, the rhetorical structure – the course of the argument – cannot easily be reduced to a repeatable structure. But there are certainly cases in which rhetorical structures can be highly repeatable and yet does not consist merely of an arrangement of facts. In other words, there are rhetorical models that focus on the optimal presentation of information in certain circumstance, independent of the specifics of the subject matter. A well known example is the pyramid structure used in newspapers, which clusters the key points of a story at the top.

There is a presentation-oriented rhetorical pattern that is useful in technical communication (and perhaps in other fields) that I call the think-plan-do pattern. Many technical communication tasks simply involve telling a user how to perform specific functions on specific pieces of machinery. But there are cases in which the user's task has highly complex input conditions and potentially far-reaching consequences. In this case, the technical communication task goes well beyond telling the user how to operate

---

<sup>1</sup>Let's make this distinction clear: people often convert content from one file format to another, including for binary formats to markup formats. This is a mechanical process, though one that may require some cleanup. It does not, in itself, impose any additional constraints on the content. It merely changes the syntax that expresses existing structures. This kind of conversion is often possible to document domain formats like DITA and DocBook. This does not mean that the resulting DocBook or DITA output will correctly express the full range of constraints or structures that these formats are capable of. You can also do a reliable transformation from one subject domain format to another (say from a relational database to XML markup). But you cannot do a reliable mechanical transformation of media domain content to the document domain or of document domain content to the subject domain. The subject domain imposes constraints that may be expressed rhetorically in the document domain, but are not expressed mechanically. These conversions are writing tasks, not something than can be done mechanically.

the machine. It is about helping them to correctly plan their actions to achieve the desired business outcome.

You can approach this problem by simply collecting all the relevant facts that the user would need to make a correct decision. But a mere listing of relevant facts is not helpful to a user who does not fully understand the complexity of the task or the seriousness of its potential consequences. For example, a user may well not understand the security implication of a particular configuration option of a computer system. The safety of that option may depend of a variety of factors, such as who has access to the system, what software is running on the system, what data it contains, and how other settings are configured.

If the user does not appreciate the seriousness or complexity of the issues involved, they may skip all of the additional information and go straight to the beginning of the procedure. If they do, a mere listing of relevant facts may scare them out of trying to change the setting (which may be just as unsafe as changing it, and may have other negative consequences).

A potential rhetorical structure addressing this problem is to walk the user through each of the decisions that need to be made in order to plan their changes correctly. This can consist of a number of carefully designed discrete questions designed to help the user figure out which issues apply to their situation and, if they do apply, how to deal with them.

In other words, the model presents a formal planning methodology in the form of a set of questions which break down the planning of the change into manageable pieces that the user can successfully comprehend and act on. It supports cognition by breaking a complex subject down into manageable pieces.

Depending on the material, it may be possible to find a common patterns in the subject matter of these questions. (The exact same set of questions need to be considered for each configuration setting, for example.) But in many cases, the questions that need to be asked are particular to the individual case. It is the rhetorical device of breaking the planning process into a set of discrete questions that is most important to improving the quality of the content and ensuring that the reader is successful.

So far in this book I have mostly presented document domain models as rather loose collections of generic document structures mainly used to separate content from formatting or to facilitate content reuse. But this example shows that the document domain can also be used to model a specific rhetorical strategy.

In some cases, you may find that specific subjects require a specific rhetorical strategy, but do not particularly lend themselves to the subject-domain approach of breaking out a consistent set of information to be provided for each instance of a subject. In these cases, creating a document domain model that enforces the appropriate rhetorical strategy may be the best approach.

# Rhetorical metamodels

There are different ways of thinking about the rhetorical structure of content. Above, I describe the topic pattern of a recipe as consisting of a picture, an introduction, ingredients, and a list of preparation steps.

However, we could notice that there are a great many other type of information with a similar pattern. For instance, a knitting pattern usually has a picture of the garment, an introduction describing the project, a list of the yarns and needles required, and a list of steps for knitting and assembling the pieces. Lots of other things look similar. Instructions for assembling flat pack furniture, for example, or planting flowers in your garden.

These are not the same rhetorical structure. You would not confuse a recipe with a knitting pattern. And each of them can have specific information fields that would make no sense for the others. A pot roast will never have washing instructions. A flat pack bookcase will never have a wine match. Nonetheless, they all have the basic pattern of picture, description, list of stuff you need, steps to complete. We might call this the make-thing-out-of-stuff-with-tools pattern.

The make-thing-out-of-stuff-with-tools pattern is what we might call a meta-model. It is not based on seeing similarities between texts, but on seeing similarities in the rhetorical patterns of texts. The meta pattern is not intended for creating content directly, but it can potentially provide hints that help us develop individual topic patterns.

Not only are there meta-patterns for topics, like the make-thing-out-of-stuff-with-tools meta pattern, there are also meta patterns for the different types of information that go into a meta pattern, such as the picture, description, list of stuff you need, and steps to complete. These are sometimes called “information types” (a confusing term, since text at any scale expresses information, and therefore the structure of information at any scale is an information type).

Two notable examples of these information type meta patterns are found in Information Mapping and DITA. Information Mapping proposes that documents are composed of just six information types: procedure, process, principle, concept, structure, and fact.<sup>2</sup> Documents are then constructed of some arrangement of information blocks of one of these six types, which it calls a map. In other words, Information Mapping proposes that every rhetorical structure is always composed of some combination of these six information types.

DITA proposes something similar, but it proposes just three types: concept, task, and reference,<sup>3</sup> which, confusingly, it calls topic types. Like information mapping, DITA assembles documents out of these topic (information) types using a map.

---

<sup>2</sup><http://www.informationmapping.com/fspro2013-tutorial/infotypes/infotype1.html>

<sup>3</sup>Or, at least, it originally proposed these three types. The DITA specification now includes other topic types, some of which are much more concrete than these original three.

In the concept/task/reference metamodel, our recipe topic pattern would consist of one concept topic (the introduction), one reference topic (the list of ingredients), and one task topic (the preparation steps). And our make-thing-out-of-stuff-with-tools meta-pattern would similarly consist of one concept topic (description), one reference topic (list of stuff you need), and one task topic (steps to complete). (DITA's information model does not include pictures. It just provides a mechanism for including them in textual topic types.)

What neither DITA nor Information Mapping provide is any way to model the larger recipe pattern. DITA will let you write a map to combine a concept topic containing an introduction (which presumably is where you would include the picture), a reference topic containing a list of ingredients, and a task topic containing preparation instructions. But it does not give you a way to specify that a recipe topic consists of one concept topic, one reference topic, and one task topic. In other words, DITA does not provide any way to define larger types or the overall rhetorical structure of documents.

What DITA does do is provide a way to specify a concrete instance type of any of its meta-types. A list of ingredients has a specific structure that is not the same as the list of pieces in a flat-pack furniture box. It consists of an ingredient name, a quantity, and a unit of measure. The unit of measure is vital in an ingredient listing because not all ingredients are quantified in the same way. You don't measure eggs the same way you measure flour, for instance.

To express this constraint, DITA will let you specialize the reference topic type to create a list-of-ingredients topic type that imposes (and records) this constraint. You could then construct a recipe using an introduction-to-recipe topic (a specialization of concept), a list-of-ingredients topic (a specialization of reference), and a preparation-steps topic (a specialization of task). However, it still would not give you a way to specify that a recipe consists of these three topic types in this order. (More on DITA specialization in Chapter 41, *Extensible and Constrainable Languages*.)

Actually, it is possible to define a recipe topic types in DITA, but this involves having a different idea about how atomic the basic DITA topic types are. Some DITA practitioners might say that a recipe is not a map made up of three information types, but a single task topic. In this view, a task topic is much more than what Information Mapping would call a procedure. It allows for the introduction of a task, a list of requirements, and the procedure steps all within the definition of a single topic. (I have asked a number of DITA practitioners how a recipe should be modeled in DITA and have received each answer from multiple people.)

One of the reasons for this uncertainty about what an atomic topic is in DITA is DITA's focus on content reuse. DITA topics are not only units of information typing, they are units of reuse. The approach in which a recipe is a single topic leaves you with fewer, larger units of content, which makes individual topics harder to reuse. The atomic unit of content that is small enough to maximize potential reuse is much smaller than the atomic

unit of content that contains a complete rhetorical pattern. The atomic unit of reuse is smaller than the atomic unit of use.

Because DITA has no mechanism for describing model larger than a topic, a DITA practitioner is left with a choice between modeling for maximum reuse and modeling to constrain a topic type to rhetorical structure. In practice, it seems that different DITA users make different decisions about how atomic their topic types should be, based on their business needs.

## Meta-models vs generic models

Ideally, a meta-model should just be a model of models. You should not be able to use it for anything other than to create concrete models. It should not only suggest those things that each specific model should have in common, but also the unique things that are specific to particular instance of the pattern. For instance, it should in some way suggest that a recipe instance of the make-thing-out-of-stuff-with-tools meta pattern might want to include a wine match. (A thing-goes-with-other-thing relationship, perhaps.)

In practice, a meta-models often turn out to be simply a list of those things that all instances of the meta-model have in common. In many cases, instead of inventing an entirely new notation for describing meta-models, people just create a model with only the common properties. Thus the expression of the meta-model takes the form of a generic model, which means that it is perfectly possible to write content using that generic model. Thus while DITA's concept, task, and reference topic types are intended as meta-models to be specialized into concrete models, they are implemented as generic models which can be used directly.

A great many DITA users don't specialize at all. They write all of their content in the base task, reference, and concept topic types (or the even more basic "topic" topic type, of which task, reference, and concept are actually specializations). This means that the topic type imposes no specific rhetorical pattern. But at the same time, the generic pattern can be confining. For instance, DITA's default topic model does not allow you to have two procedures in a single task topic.

Are meta-models useful for defining topic patterns? If a concrete topic pattern describes the kinds of information that are needed to help a particular audience perform a particular task, do we arrive at that pattern more easily by derivation from a meta-model or from observation of multiple concrete examples of actual topics?

The obvious problem with the current generation of content meta-models is that none of them alert us that a recipe might need a field for a wine match. It is not impossible to imagine that a meta-model could do this. A meta-model could observe that objects are commonly used with other objects and lead us to ask what other objects is a steak dinner used with. There are obviously multiple aspects of this question. A steak dinner is used with a knife and fork. A steak dinner is used with a table and chair. A steak dinner is used with family and friends. A steak dinner is used with a glass of wine. How do we

characterize each of these thing-used-with-thing relationships in a meta-model, and how do we decide which of these types of thing-used-with-thing types is relevant to a recipe?

Perhaps, for instance, we might decide that because a recipe describes a foodstuff, thing-used-with-thing relationships are relevant when the other thing is also a foodstuff. In other words, we might decide that a thing-used-with-like-thing relationship is part of the meta-model.<sup>4</sup> (I am not, by the way, suggesting that this is a useful part of a meta-model, I merely wanted to illustrate the problem of defining a meta-model that would comprehend all the specific models we might care about in the real world.)

This is getting complicated enough for me to conclude that, while the ontologists may one day come up with a such a model and a reliable way to derive concrete content models from it, for most writers, information architects, and content strategist, building a concrete topic model from the observation of instances is probably the preferable method.

Creating a good subject-domain structured model for content is actually pretty simple when you get down to it. You simply ask yourself five questions:

1. What information does the reader need to fulfill whatever purpose this unit of content is meant to serve?
2. What is the best way to express each of these elements?
3. What is the best way to organize these elements so that the reader can quickly recognize that this is the content they need and can effectively find the information then need in the content?
4. What constraints do I need to apply to the content to make sure that what the author creates conforms with the answers to the first three questions, and how will I test their conformance to those constraints?
5. What level of detail and precisions do I need in the content structures I create to make sure that the structured writing algorithms I want to apply to the content can run reliably.

Answering those question is not necessarily easy, of course. But the process itself is simple and you should not make it any more complex than it needs to be.

## Making the rhetorical structure explicit

I noted above that there can be a rhetorical structure in a piece of text that is just a sequence of paragraphs. You can discern the topic pattern in those paragraph and model

---

<sup>4</sup>Rob Hanna's Enterprise Content Metamodel[<https://www.oasis-open.org/committees/download.php/41040/Enterprise%20Content%20Metamodel.pptx>] does attempt to do something like this for business information, attempting to describe the relationships between pieces of business content based on the business functions they serve as a basis for deriving specific information types.

that pattern in a content type, and still present the output as a sequence of paragraphs. Presumably, in each instance of the topic type, those paragraphs would now be more consistently expressed with fewer errors and omissions than before, but the presentation itself would be the same.

Alternatively, you may choose to make the rhetorical structure more explicit to the reader as well as to the writer. In this case, the sequence of paragraphs might be replaced with a distinct combination of headings, graphics, tables, lists, pictures, and text sections that would repeat in every topic of that type.

The question, of course, is whether making the rhetorical structure explicit in this way improves the content. In its favor, the more explicit rhetorical structure makes it easier for the reader to recognize the type. (As we noted above, you can recognize a recipe by its shape, without reading a word.) This makes it easier to identify relevant content, which is particularly important on the Web. It can also make it easier to scan the content to pick out the parts you need. The argument against this treatment is that it can lead to a noisier page that is harder to read straight through.

Whether you want to make the rhetorical structure of your pages explicit in these ways, therefore, is a matter to be decided on a case-by-case basis. But don't fall into the trap of supposing that because you have chosen a plain presentation, that means there is no rhetorical structure, and therefore no structured type. The rhetorical structure of the content is a separate thing from the presentation of the content, and the aim of structured writing is to improve the rhetorical structure, not just to make the presentation more uniform.

## Structure and repeatability

However much success we may have in defining common rhetorical structures, most content does not surrender to the analytical knife entirely. When we define a rhetorical structure, we are essentially defining something repeatable. While one could hypothetically take a piece of exposition and define the structure that it follows (and in a sense, this is what ontology does), this is only useful in a structured writing sense when that structure is repeatable, when there is another piece of content that can follow the same structure, enabling us to reuse the design work and testing that we have done, and allowing algorithms to take over parts of the processing of that structure.

Where the rhetorical structure of a piece is unique – specifically when it is irreducibly unique, not just unique because you have not recognized its similarity to the structure of other pieces – there is no benefit to the imposition of an external structure definition. Irreducible rhetorical uniqueness can occur at any scale. Sometimes it occurs at the scale of an entire book, such as this one. Sometimes it occurs in a single descriptive paragraph in a reference work that otherwise consists entirely of repeatable key/value pairs. (Which is why so many content types have a “description” section for everything that cannot be

easily modeled. The content of such a section is not uniquely descriptive, it is simply not easily reducible to fielded data.)

This variation in the degree of unique vs repeatable rhetorical structure in content is the reason why we need different structures for different types of content. If we attempt to fit all content into one structure, however broad that structure may be, we are either failing to model, and therefore to manage or apply algorithms to, much of the repeatable structure of our content, and therefore diminishing the repeatability of our process, or we are squeezing unique rhetorical structures into the mold of a repeatable structure, distorting their rhetoric. (This is a common problem with strict information typing systems like DITA's task/concept/reference, which often don't fit the rhetorical pattern the writer is trying to create.)

We can usefully consider rhetorical repeatability in three categories:

Repeatable data	The same information is required in each instance. For instance, a recipe requires the name, quantity, and unit for each ingredient of a dish.
Repeatable argument	The way in which the information is conveyed to the reader is the same each time. For instance, the think-plan-do approach to task support content.
Repeatable relationships	The way in which the subject discussed in a piece are related to their subject matter is the same each time. For instance, the consistent use of the same subject annotations across an entire content set, including everything from strict reference content to highly discursive conceptual material reflects a consistency in what types of subjects are important in an information set and how they are named.

Note that these types of repeatability are not exclusive to an entire piece of content. Rather, each piece of content contains some mix of these types. Thus one of your content types may be a generic document-domain structure that is used for highly discursive material with no repeatable data or repeatable argument, but it can still use the same subject annotations as the rest of the content set.

Remember that the ultimate aim is to partition and redirect complexity in the content system. Part of that complexity is the varying levels of repeatability in content – a problem seldom seen in other forms of data management. If you want to deal effectively with all of the complexity in your content system, then this variation in degrees of repeatability is one of the things you need to deal with. If it is not dealt with successfully, either the repeatability of repeatable content will be lost or diminished, or the uniqueness of unique content will be compromised. Either way, that is complexity falling through to the reader in the form of impaired content quality.

# The importance of the rhetorical model

If the aim of structured writing is to partition and redirect complexity in the content system without letting any of that complexity leak out of the process and fall down to the reader, then the correctness and consistency of rhetorical models is a core concern. Poor rhetoric means poor content, and poor content means that the complexity of achieving consistent and correct rhetoric has been dropped somewhere in the content system.

A content system relies for its effectiveness on the ability of its principle authors and occasional contributors to maintain a consistent rhetorical standard. The three tools that it has available to do this are:

- Minimize intrusions into the attention of writers while they are writing. Any attention given to other matters while writing is attention taken away from rhetoric.
- Guide writers to help them provide the correct rhetoric. In other words, reuse the rhetorical design work that you have done, and that you have tested and refined with readers.
- Partition out the rhetorical aspects of a composition by collecting facts in a subject domain structure which can then be transformed into the appropriate rhetorical form by algorithms.

Content inherently varies in how structured it is, so the more structured of these techniques only work for some part of your content set. You need all three of these techniques to provide the most comprehensive rhetorical support across your content system.

---

# Chapter 31. Mechanical Structure

Mechanical structure is the way we record, encode, and enforce the rhetorical structure, of a formal content model. While the rhetorical structure is concerned with how the content fulfills its purpose, the mechanical structure has to deal with specific representational issues and get the nuts and bolts of content recording right – without dropping any complexity in the process.

However detailed you may have been in delineating the rhetorical structures you want to achieve, you will find there is more detailed work to do when it comes to the specifics of the mechanical structure, and particularly with the mechanics of factoring out structures in order to partition and transfer content system complexity.

The mechanical structure of structured writing is a computer data structure. However, conventional computer data structures like relational database tables do not work well for content because they are too regular to fit the shape of content. Creating structures that are regular enough for algorithms to deal with yet irregular enough to fit written language is an interesting problem to which more than one solution has been proposed.

Raw text is just a stream of characters. Inside that stream of characters, there are rhetorical structures like headings, bibliographical entries, bold text, chapters, ingredient lists, links, wine matches, tables, function signatures, and labeled lists. The question is, how do you express these various structures within the raw stream of characters?

The most common answer is that you divide the text up into a series of blocks, and divide those blocks into smaller blocks until all the structures you want to capture are contained in blocks. This is not the only way to do it. Some file formats, such as WordPerfect, use independent stop and start markers to delineate structures, meaning the boundaries of structures can overlap. But while this kind of structure can work for the media domain, it is very difficult for many structured writing algorithms to work with. Therefore most structured writing today uses the nested block approach.<sup>1</sup>

---

<sup>1</sup>It is technically possible to implement independent stop and start markers even in languages that are mainly block based. You simply define empty blocks for the start and stop markers. Both DITA and DocBook do this for things like delineating arbitrary bits of content for reuse and defining arbitrary spans of content for indexing. Both of these uses strike me as a really bad idea and I would recommend against their use in any content that has any lifespan beyond its first publication.

recipe: Hard Boiled Egg

introduction:

A hard boiled egg is simple and nutritious.  
Prep time, 15 minutes. Serves 6.

ingredients:

- \* 12 eggs
- \* 2qt water

preparation:

1. Place eggs in pan and cover with water.
2. Bring water to a boil.
3. Remove from heat and cover for 12 minutes.
4. Place eggs in cold water to stop cooking.
5. Peel and serve.

There are definitely cases in which the use of non-overlapping blocks for the mechanical structure of content does not fit with the rhetorical structures that we find in our content. This is more of an issue for the academic study of texts than for structured writing as a means of partitioning and redirecting complexity in the content system, but it can occur in both fields. Allowing overlapping fields, however, adds complexity to many of the structured writing algorithms, meaning it is almost never a worthwhile approach to partitioning our content.

For our purposes, therefore, I am going to deal with the mechanical structure of content strictly in terms of nested non-overlapping blocks.

## Flat vs. nested structures

Even when we decided on the nested blocks approach we are still left with some fundamental choices about mechanical structure. The first is flat vs. nested structure.

We noted in Chapter 4, *Writing in the Document Domain* that in HTML, you have six levels of heading (h1 through h6) whereas in DocBook you have only `title`. In DocBook, you can divide a document up into sections and nest sections inside sections. You can then print the titles of sections inside sections in a smaller font than the titles of first level sections. You get to have differences in heading size without having six different heading tags.

But the DocBook model assumes that the real structure of a document is a hierarchy of nested sections and that the size of titles announces the steps up and down that hierarchical tree. HTML makes no such assumption. It will let you put a `<h4>` immediately after an `<h1>` if you want to. It treats documents as essentially flat structures punctuated by headings of various sizes as and where appropriate.

Which model of a document is correct? Which corresponds best to the rhetorical structure of the document? You can think of a document as being organized hierarchically, with major ideas expressed in sections, sub-ideas supporting the major ideas in subsections, etc. There are many documents that fit that model. But you can also think of documents as being more like a journey in which headings function more like road signs. A city gets a big sign, a hamlet a small sign, and a town a medium sign. But the town is not inside the city, not the hamlet inside the town, and there is no guarantee that on leaving the city you will come to the town before you come to a hamlet.

Studies by Peter Flynn indicate that most authors think of the documents they are writing much more in terms of a punctuated linear model than a hierarchical model.

The classical theory, derived from computer science and graph theory, is that the document is a hierarchical tree (actually inverted: a root-system) and that all necessary actions can be seen in terms of navigation around the tree, and of insertion into and withdrawal from the nodes which form the branches and leaves.

The conventional writer, however — and we expressly exclude the markup expert, as well as the experienced technical authors who responded to the survey — is by repute probably only marginally aware of this tree; but we have been unable to measure this at present. In this view, the document is seen as a continuous linear narrative, broken into successive divisions along semantic lines, and interspersed with explanatory material in the form of figures, tables, lists, and their derivatives.

—Flynn2009

And as Flynn's research illustrates, there is a good chance that the author is not thinking of their argument as consisting of a strict hierarchy of points either, but as a sequence of points with the occasional insertion of headings to break up the text or perhaps signal a change in emphasis or subject matter.

Clearly defining the rhetorical structure of a document demands dividing it up into blocks such as introduction, ingredients, and preparation. As we have noted, treating these parts of the rhetorical structure as blocks is essential to establishing the context in which our block names are defined and by which our processing rules tell the difference between one type of block and another. This clearly involves asking the author to think a little bit more hierarchically than they might otherwise, which means imposing a little bit of complexity on them for the sake of all the partitioning and redirection of complexity we

need to do for the content system overall. Some semantic and rhetorical blocks are going to be naturally more hierarchical than others as well. But there is no need to do any more of this than the needs of our partitioning system demand.

If the constraints that we want to express in structured writing demand hierarchy, while functional lucidity demands more of a punctuated linear model, how do we reconcile these two opposing requirements in markup language design?

This is of greatest concern in the design of document domain languages. The structure of media domain languages is largely dictated by the shape and relationship of the media-domain object they are modeling. In the subject domain, we have abstracted content out of strict document order. Hierarchy in the subject domain tend to match the hierarchy of relationship in the subject matter itself.<sup>2</sup> In the document domain, however, it is a real concern. The document domain consists of abstractions of document structures and the nature of their relationship to the structure of thought in the text is not obvious.

The options available are:

- Create a really flat document domain language. Examples are HTML and Markdown. The problem here is that they impose few constraints, and the lack of context-setting hierarchy makes it hard to model different types of document structures without creating hundred of tags – which would negate any functional lucidity that you gained by keeping the language flat.
- Create a hierarchical language that has a really permissive structure, so that you can put boxes inside boxes in lots of different ways. An example of this is DocBook. The problem here is that the possible permutations make writing algorithms difficult and you often need to impose additional constraints on your authors that are not expressed or enforced by the markup itself. This again diminishes functional lucidity, and compromises conformance. (An interesting property of this approach is that the flexibility of the language means that authors can choose to create documents that are deeply nested or very flat. This is not really a virtue, however, as it is not clear how this choice contributes to improved content quality.)
- Define a smaller, stricter document domain language that is appropriate to the particular types of documents you want to write, possibly as restricted subset of an existing language like DocBook. The main difficulty with this approach is that it involves you in having to do your own language design, which many organization try to avoid. In other words, it involves transferring complexity to an information architect or content engineer. Once you have decided to go this route, going to the subject domain instead may be no more expensive while providing better functional lucidity and conformance.

---

<sup>2</sup>Though this is not universal. Addresses, for instance, which are based on hierachal locations, are modeled as flat ordered lists. The order reflect the hierarchy, but the nesting of city inside country and of street within city is not reflected in the structure of an address record.

- Define a strict hierarchical document domain language that expresses the constraints you need and make people learn it. This works if you are able to recoup the expense of training your authors. It does not work if you want to include occasional authors in your pool of contributors.
- Move content creation to the subject domain.

## Annotating blocks

Every block is annotated to tell us, and algorithms, what it means, which is to say, what constraints it meets. The most basic of these annotations is the name of the block, which describes its basic type. The name of a table block is “table”, which tells us that its type is table. Blocks may also have other annotations that either refine its type or provide additional information about the block.

These additional annotations don’t have to be in the same domain as the block name. We saw an example of this in Chapter 5, *Writing in the Subject Domain*:

```
<section publication="Wine Weenie">
 <title>Wine match</title>
 <p>Pinot Noir</p>
</section>
<section publication="The Teetotaler's Trumpet">
 <title>Suggested beverage</title>
 <p>Lemonade</p>
</section>
```

Here the `section` element defines a block in the document domain, and the `publication` attribute adds an annotation in the management domain.

## Agreeing on names

For structured writing to work, it is essential that everyone involved understands and agrees on what the names and annotations mean. These annotations tell downstream people and processes what constraints each block obeys. If the names and the values of the other attributes don’t mean what we have all agreed they are to mean, communication between the partitions of the content system breaks down and complexity leaks out of the seams and descends on the reader.

Confusion and disagreement about what the names and annotations of a particular language mean are not uncommon. Large document domain languages like DocBook and DITA have large vocabularies, and many of the names they offer are quite abstract. Questions about the right way to tag certain passages are common in the communities

around these languages, and opinions can vary considerably in some cases. These disagreements don't only affect low-level structures. In DITA, for example, it is common to debate if a topic that is operational but not procedural is a concept or a task. Some writers choose to use only generic topics because they don't feel the models of the task, concept, and reference topics fit the content they are creating.

Having precise definition of terms is important, therefore, in developing a structured writing language. But it is equally important that the language be functionally lucid. The authoring algorithm requires that creating structure should not come at the detriment of the writing itself. This requirement is not met if writers are constantly having to puzzle out or debate the right way to mark something up. As we have seen, conformance is chiefly served by designing structures that are easy to conform to. This does not only mean defining what the names mean, but choosing the structures that are easiest for authors to name correctly.

In the software world, meta models and abstraction are powerful tools for modeling systems. They provide clear high-level rules for the design of specific structures and create opportunities to reuse code for objects with a shared base model. But these tools can also lead to very abstract naming schemes and even to abstract structures. In the content world, such abstract names and structures can be formally correct but lack the kind of functional lucidity required for effective authoring.

The problem of defining a mechanical structure to express a rhetorical structure is not only one of defining a correct representation of the content. We also have to design and name structures that can be written by our intended set of authors without imposing a heavy burden on their attention. In other words, designing content structures, at least those intended for use by authors, is as much about interface design as it is about data structure design. And interface design is all about partitioning and distributing complexity.

Clear concrete and specific names, and an organization of blocks that intuitively fits the subject matter, all make for easier authoring. There is no reason that such structures cannot be derived from abstract models, or that they cannot be mapped to abstract models after the fact, but it is important not to let the abstractions intrude into the world of the author.

## Different rules for intermediate languages

Of course, functional lucidity only matters for the formats that authors actually write in. As we have seen, the publishing algorithm typically consists of multiple steps, and each one of those steps can create a format that is closer to the media domain than the one before it. It is perfectly possible to design a document domain structure for the sole purpose of serving as a step in the publishing chain. Separate authoring

formats are created for authors to actually write in (perhaps subject domain formats or simplified highly constrained ad-hoc document domain formats). Content is transformed from these formats to the document domain format by the presentation algorithm and then the document domain format is translated in to various different media domain languages by the formatting routine. An arrangement like this eliminates the need to compromise between different demands in designing a single language, generally making each language in the chain simpler and more constrained, which in turn makes it each one easier to validate and to process.

## Secondary structures of interpretation

Under normal circumstance, the structures that constrain the content also constrain its interpretation. But there are cases in which annotations are added to structures to constrain a different interpretation of the content from the one specified by the structure that contains it, and not only of individual fields, but of a structure as a whole.

Consider these examples of HTML Microformats from Wikipedia<sup>3</sup>. The first example shows an address formatted as a list.

```

 Joe Doe
 The Example Company
 604-555-1234

 http://example.com/


```

Here the phrase “The Example Company” is contained in li tags. This is part of a list structure delineated by ul tags, so the markup is largely structural in the document domain. The li does not really tell you anything about what the content itself is about. It does not tell you anything useful beyond what document structure it belongs to.

The second example adds vCard microformat markup:

```
<ul class="vcard">
 <li class="fn">Joe Doe
 <li class="org">The Example Company
 <li class="tel">604-555-1234


```

---

<sup>3</sup><https://en.wikipedia.org/wiki/Microformat>

```
http://example.com/


```

This example adds subject domain metadata in the form of the class attributes. For example, it says that the phrase “The Example Company” is a reference to an organization (`org`). This annotation is not modifying or refining the constraint expressed by the `li` tag. It is saying something else entirely.

But this is not just about suggesting a different way of interpreting The Example Company (as an organization name, as well as as a list item). There is actually a complex subject domain structure being expressed by the vCard markup. Not only is the list item The Example Company annotated as `org`, the list that contains it is annotated as `vcard`. The constraint of interpretation provided by `org` is actually dependent on it being part of a `vcard` structure.

In other words, the secondary structure created by the annotations in the sample above is equivalent to pure subject domain markup like this:

```
vcard:
fn: Joe Doe
org: The Example Company
tel: 604-555-1234
url: http://example.com/
```

The microformats are overlaying a second structure on the list structure. In the world of HTML, this makes sense. HTML needs to be a standardized document domain language so that browsers can display it for human reading. Humans don’t need the vCard annotations to recognize that the content is an address, but algorithms do. (This is part of dumbing it down for robots, as we discussed in Chapter 7, *Quality in Structured Writing*.) So the microformat adds a second, hidden, subject domain structure to the document for readers who are algorithms rather than people.

Structured writing constrains both the creation and the interpretation of content. In the normal case we expect that the creation of the content would be just as constrained as the interpretation. After all, it is hard to rely on the interpretation of structure if the creation of the structure is not constrained. However in this case the interpretation of the data is more constrained than the creation.

Authoring our content this way would obviously be inefficient and error prone. But this is only a problem if the content is actually authored in this format. If the content is authored in a format in which its creation is constrained to the same extent as we wish the output to be constrained, it does not actually matter that the resulting output constrains interpretation more than it constrains creation. Our concern as content creators is simply

to make sure that any content we produce that promises to abide by a constraint actually does so, whether the format we deliver it in actually imposes that constraint or merely annotates it.

So, we can confidently produce this information using subject domain markup and then deliver it as HTML with hCard annotation markup using a presentation algorithm something like this (as with all example algorithms in this book, this is pseudocode):

```
match vcard
 create ul
 attribute class = "vcard"
 continue

match fn
 create li
 attribute class = "fn"
 continue

match org
 create li
 attribute class = "org"
 continue

match tel
 create li
 attribute class = "tel"
 continue

match url
 create li
 create a
 attribute class = "url"
 attribute href = contents
 continue
```

## Child blocks vs. additional annotations

As noted above, sometimes the name of a block is not sufficient to fully describe the constraints it meets. In these cases, we can add additional annotations to a block. But we can also achieve much the same thing by adding child blocks to the block. A child block belongs to the main block, so it is part of it and can constrain the interpretation of the main block just as much as an additional annotation does. One of the issues in designing mechanical structures for content, therefore, is when to use additional annotation on a block and when to use child blocks.

Different markup languages have different levels of support for additional annotation on blocks, so this issue is affected by the markup language you choose. SAM, for instance, only supports a limited fixed set of additional annotation on blocks, and therefore any additional constraint of interpretation you want to do in a SAM-based markup language has to be done with child blocks.

In XML, though, there is broad (though not unlimited) support for additional annotations on blocks in the form of attributes. In the XML world, therefore, there is considerable choice, and considerable debate, about when and where you should use elements vs. attributes in your content models.

Consider, for example, this XML element that contains no content but two attributes:

```
<author-name first="Mark" last="Baker">
```

the element is named `author-name` and it has two attributes named `first` and `last` that contain my first and last name respectively.

Why is this marked up like this and not like this:

```
<author-name>
 <first>Mark</first>
 <last>Baker</first>
</author-name>
```

Both these constructs express the same information in a way that clearly constrains how author names are to be written and how the markup of author names is to be interpreted by algorithms.

Is one of these options correct and the other incorrect? When should you use attributes and when should you use elements?

Consider our vcard example. It could be written this way, using just elements:

```
<vcard>
 <fn>Joe Doe</fn>
 <org>The Example Company</org>
 <tel>604-555-1234</tel>
 <url>http://example.com/</url>
</vcard>
```

Or it could be written this way, using attributes:

```
<vcard
 fn="Joe Doe"
 org="The Example Company"
 tel="604-555-1234"
 url="http://example.com/"
/>
```

The first says that `fn`, `org`, `tel`, and `url` are independent structures that belong as members to a `vcard` structure. The second says that the `vcard` structure has a number of data fields – annotations – that complete its meaning.

Does this distinction matter terribly? Both allow you to get at the information you want. Both constrain the creation and the interpretation of data. There are limits to the version that uses attributes. You can't have more than one attribute with the same name, whereas you could have more than one member elements of the same name (multiple `tel` elements for someone with more than one telephone number for instance). Also, XML specifies that attributes are unordered, so can't restrain either the order in which writers create them or the order in which the parser reports them to a processing application.

Given this, you may be wondering why people bother with attributes, since you can do the same things with elements and have both more flexibility and more capacity to impose constraints. Yet people continue to use attributes extensively when designing markup languages in XML. When people create XML document types for representing data, rather than for writing document, they almost always use the attribute format, perhaps because it is slightly less verbose and slightly easier to read, or perhaps because as programmers they are accustomed to representing data as key/value pairs linked with = signs.

But for documents it is more complex. To understand why XML even has attributes, and why other languages, such as SAM or reStructuredText, also have similar mechanisms for adding annotations to blocks, we need to go back to the original concept of markup as something written onto a manuscript after the fact. In this view, markup is an addition to the text, not part of it. The content of an element is part of the underlying text. Anything you want to add, therefore, cannot be element content, since that would be adding to the text. Everything else has to be added to element definitions as attributes.

This view is reinforced by the academic interest in markup as a way to prepare texts for study. Again, here, the text is preexisting and canonical. The markup is external to it and so everything that is external to the original text must be contained in the markup itself (as attributes) and nothing that is internal to the original text must be removed or replaced by markup. (This is a form of partitioning in its own right, to serve a particular purpose.) Thus in this fragment of a Shakespeare play marked up by John Bozek we see that the original text is kept perfectly intact:

<ACT>

```
<TITLE>ACT I</TITLE>
<SCENE>
<TITLE>SCENE I. Rousillon. The COUNT's palace.</TITLE>
<STAGEDIR>
Enter BERTRAM, the COUNTESS of Rousillon, HELENA, and LAFEU,
all in black
</STAGEDIR>
<SPEECH>
<SPEAKER>COUNTESS</SPEAKER>
<LINE>
In delivering my son from me, I bury a second husband.
</LINE>
</SPEECH>
<SPEECH>
<SPEAKER>BERTRAM</SPEAKER>
<LINE>And I in going, madam, weep o'er my father's death</LINE>
<LINE>anew: but I must attend his majesty's command, to</LINE>
<LINE>whom I am now in ward, evermore in subjection.</LINE>
</SPEECH>
```

Had this markup employed the normal refactoring of text into markup that we have seen in our discussions of the document domains, then the number and title of scenes would have been factored out so that instead of:

```
<SCENE>
 <TITLE>SCENE I. Rousillon. The COUNT's palace.</TITLE>
```

we might have factored out the scene number and the word ‘SCENE’ like this:

```
<SCENE>
 <TITLE>Rousillon. The COUNT's palace.</TITLE>
```

Going further, we might have noted that the introduction of a scene is invariable the name of its location, so we might have done this:

```
<SCENE>
 <LOCATION>Rousillon. The COUNT's palace.</LOCATION>
```

or even this:

```
<SCENE location="Rousillon. The COUNT's palace.">
```

And similarly, we might have replaced:

```
<SPEECH>
<SPEAKER>BERTRAM</SPEAKER>
```

```
<SPEECH SPEAKER="BERTRAM">
```

Making a few changes like this in the markup would leave us with only the words actually spoken by the actors as the “text” of the play and everything else expressed as elements or attributes.

This actually makes quite a lot of sense, because all of the stage directions and attribution of speakers in a play is actually metadata annotating the speeches of the actors which are the only things the audience is actually supposed to hear.

So is the right way to markup a play to preserve the original printed text – which includes all of the playwright’s metadata – or is it better to separate the playwrights metadata from the speeches which are the ‘real’ play?

But while these question may be important for the scholarly study of text, they are not nearly so important for structured writing. Our concern is to partition the task of content creation, so we can simply choose the format that does that in the most functionally lucid way while ensuring that all the information required by the next partition is accurately captured. The fact that such question exist, however, help us to understand both why a markup language like XML is structured the way it is, and why so many texts are marked up the way they are – and why so many markup languages are designed the way they are.

The way we, as practitioners of structured writing, settle these matters is by asking ourselves which approach best supports the structured writing algorithms that we want to implement, and always remembering that the reliability of every other algorithm depends on how well the conformance algorithms works, and that the conformance algorithm depends to a large extent on how the authoring algorithm designs for conformance and for functional lucidity.

In none of this are we in the least concerned about preserving the canonical nature of a preexisting text. There is no preexisting text. We are all about creating new texts. Because of this we have every reason to prefer to use sub-structures rather than annotations on existing structures, to express things like our vCard example. In fact, SAM, which is designed specifically for structured authoring, only supports this format:

```
vcard:
fn: Joe Doe
org: The Example Company
tel: 604-555-1234
url: http://example.com/
```

SAM only supports a very limited set of annotations on blocks, all of which have predefined meetings. You could, in fact, eliminate annotations on blocks altogether, and use child blocks for everything, but I have supported a limited set of common management-domain block annotations in SAM, mostly to improve functional lucidity.

In summary, when defining the mechanical structure of your structured writing, don't get hung up on what is text and what is markup. In each domain, text and markup together form a body of constrained content which can be successfully created by an author and successfully processes by one or more algorithms. Only when you resolve the content all the way to the media domain do you finally have to sort out exactly which characters appear in which order and decoration to represent that content to a particular audience. When we choose to create content in the other domains it is precisely because we want to exercise more control over these things, and to use algorithms to help us create and manage them. Whether some idea or constraint is expressed by text or markup in those domains should be based solely on what works best in those domains. (Which is precisely why attempting to do structured writing using a WYSIWYG editor is so counter productive, and why it is so important to ensure the functional lucidity of your markup as markup.)

---

# Chapter 32. Blocks, fragments, paragraphs, and phrases

I said that the structure of a structured writing document is made up of nested blocks. The question is, how do I design a content structure as a set of nested blocks. To begin we should look at the different types of blocks you might create, their purpose, and how they relate to each other. While most markup systems don't make a high-level distinction between different types of blocks (in XML, they are all elements, for instance), from a language design point of view it is useful to break down blocks into different types, each of which requires a different kind of design focus.

## Semantic blocks

If we asked a writer to describe the things that make up a document, they would probably name things like paragraphs, tables, and lists. If we asked about a particular type of document, such as a recipe, they would probably say things like introduction, ingredients, and preparation steps. A structured writing language will typically be much more fine grained than this. A table, for instance, may be made up of dozens of smaller structures like rows, cells, and cell contents. But tables, procedures, and lists are the units that have meaning to writers independent of how they are constructed internally. Without knowing anything about the mechanics of structured writing, a writer could design the rhetorical structure of a piece of content as a set of such blocks. They might say, for instance, that a recipe consists of three main blocks: introduction, ingredients, and preparation, without thinking about markup languages at all.

At the risk of adding further burden to an already overloaded term, I am going to call these recognizable objects “semantic blocks” because they are blocks that mean something in whichever domain they belong to. Higher level markup design is essentially a matter of defining semantic blocks and the ways they go together.

An easy example of a semantic block is a list. (Note that I am not using “semantic” to mean subject domain; blocks have semantics in all domains.) A list is a semantic block because “list” is an idea with meaning in the document domain independent of its exact internal structure. A writer can say to themselves, “I want a list here”, independent of any specifics of markup. If a structure has a name like this in the real world, the block that implements it (in the terminology I am coining for this purpose) is a semantic block.

Semantic blocks generally contain other blocks that we might not talk about independently if we did not need to describe the detailed construction of a semantic block. I will call these “structural blocks”. Structural blocks are the construction details of a structured document. By analogy, if a window were a semantic block in architectural terms, the lintel, sash, and jamb are structural blocks.

Don't think of the distinction between a semantic block and a structural block as being hard and fast. The distinction has more to do with design intent than any concrete characteristic. The main point of making the distinction is to encourage you to think of markup design first in terms of semantic blocks. Blocks, with whatever internal structure you require, that will capture the structure of something that is real and meaningful to you. Don't get bogged down in the precise internal structure of semantic blocks until you figure out which semantic blocks you need.

Different markup languages often construct the same semantic block differently. DITA, DocBook, and HTML all define lists, and each of them defines the internals of a list differently. Nonetheless we recognize that each of them is an implementation of the idea of a list and for most purposes you could markup your content with any one of them without any loss of functionality.

A list is made up of structural blocks that build the shape of a list. I'll illustrate this with XML since it makes the blocks explicit:

```


 <p>This is the first item.</p>

 <p>This is the second item.</p>


```

Here the semantic block is the ordered list bounded by the `ol` tags. The `li` and `p` tags inside are structural blocks that together implement the structure of an ordered list. (Since we would tend to regard paragraphs as semantic blocks, albeit relatively simple ones, this illustrates that a semantic block can also be a structural block for a larger semantic structure.)

Other document domain examples of semantic blocks include tables and procedures. (Again you will find that DocBook, DITA, and HTML, not to mention S1000D, and reStructuredtext, all have tables, all with different internal structures, and that both DocBook and DITA have procedures, again internally different. It is possible to disagree greatly about how to structure a semantic block while still recognizing different implementations as examples of the same semantic block.)

In the subject domain, examples of semantic blocks would include the ingredients list from the recipe example we have been using:

```
ingredients:: ingredient, quantity, unit
eggs, 3, each
```

```
salt, 1, tsp
butter, .5, cup
```

and the parameter description from an API reference:

```
parameter: string
required: yes
description:
 The string to print.
```

One characteristic of semantic blocks is that they often tend to repeat as a unit, as this example does in an API reference entry:

```
function: print
return-value: none
parameters:
 parameter: string
 required: yes
 description:
 The string to print.
 parameter: end
 required: no
 default: '\n'
 description:
 The characters to output after the {string}(parameter)
```

They may also be used as a unit in different places in a markup language, or in different markup languages. For instance, the ordered list semantic block may be allowed in more than one place in a document domain language, such as in a section or in a table cell.

In fact, if you have multiple markup languages in your content system, particularly multiple subject domain languages, then it makes a lot of sense to define a common set of semantic blocks for use across all of these languages. Thus if you have five subject-domain languages that all require lists, you use the same list structure across all five of those languages.

This means that you can define the structure of a list once, and define all of the algorithms that work on lists once, and reuse them for all of the subject-domain languages that share those structures. It also means that your authors only have to learn one definition for each semantic structure, which makes it much easier for them to learn several different subject domain languages, since many of the details are the same.

This means that each subject domain language only has to define, and provide processing algorithms for, those structures that are unique to the subject matter. All the other structures you need to create content in your environment are already defined, tested, and

ready to use. This makes it much quicker and simpler to design and implement a new subject domain language when you need one.

Designing in terms of semantic blocks not only helps keep markup design and processing simpler, it also improves functional lucidity. Present the markup language to the writer as a set of familiar objects like lists or tables, or logical structures like ingredient list or parameter description, rather than a sea of tags, and the task becomes easier to understand (and the tags easier to remember).

Semantic blocks also make things easier for tools. An XML editor that implements a WYSIWYG interface to XML authoring may provide tool bar buttons for inserting semantic blocks such as lists or tables. This allows the author to enter these blocks as complete structures rather than having to enter all the tags that comprise them separately.

The structure of a semantic block can be strict or loose. A strict semantic block has one basic structure with few options. A loose one allows a much wider variety of structure inside, sometimes to the point that it acts more as a semantic wrapper than a defined semantic block.

DocBook is an example of a language with very loose semantic blocks. DocBook has the same high-level semantic blocks as any other generalized document domain markup language, but so many tags are allowed in so many places that none of these objects are simple and easy to understand. This supports DocBook's goal of being able to describe almost any document structure you might want to create, but at the expense of functional lucidity and constraint.

How do you balance flexibility with functional lucidity and constraint in creating semantic blocks? Sometimes it is best to have more than one implementation of a particular semantic block. For instance, both DITA and DocBook have two tables model, as simple model and a more complex one based on the CALS table model.

## Information typing blocks

We have looked at examples of semantic blocks whose semantics are in the document domain (lists and tables) and in the subject domain (ingredients list and parameter description). There is another way in which some structured writing systems divide content into blocks, which is according to the type of information they contain. I'm going to call these information typing block, since the practice of dividing content into such blocks is commonly called "information typing" (though this is obviously not the only thing the words "information typing" could refer to, since all structured writing assigns information to types).

An information typing block is a type of semantic block, but information typing blocks introduce a degree of abstraction not found with most semantic blocks. Unless they have been trained in an information typing system such as Information Mapping or DITA,

most writers are not going to naturally describe the rhetorical structure of their content in terms of information typing blocks.

Information Mapping is a structured writing system which views all content as being made up of just six types of information block: Procedure, Process, Principle, Concept, Structure, and Fact. These are information typing blocks. They don't directly describe a physical or logical element of document structure (except for procedure), nor are they specific to any one subject. They describe the kind of idea that the content conveys – they are actually based on a theory about how humans receive information.

Which structured writing domain do information typing blocks belong to? Clearly they are not media, subject, or management domain structures. Are they a kind of document domain structure or something else again? I believe it is more useful to regard them as document domain structures than to invent another domain. Information mapping is a theory about the construction of documents to make them more effective. It regards a document as a mapping of information typing blocks, so Information Mapping's information typing blocks are components of documents, and therefore in the document domain.

DITA also adopted this idea of documents being made up of information typing blocks. In DITA's case, these blocks are named topics, which leads to some confusion since the word topic can be used to refer to both information typing block, and also to a complete document (as in a "help topic" for instance).

DITA has popularized the idea that all content (or all technical content, at least) is made up of just three information typing blocks: concept, task, and reference.<sup>1</sup> (DITA actually defines more topic types than this today.) This idea is appealing because it is simple and

---

<sup>1</sup>There is evidence that DITA is moving away from this vision of information typing. In DITA 1.3, the technical committee puts the emphasis on topic and map as the core types, rather than concept, task and reference.

The DITA Technical Committee wants to emphasize that topic and map are the base document types in the architecture.

Because DITA was originally developed within IBM as a solution for technical documentation, early information about DITA stressed the importance of the concept, task, and reference topics.

Many regarded the topic document type as nothing more than a specialization base for concept, task, and reference.

While this perspective might still be valid for technical content, times have changed. DITA now is used in many other contexts, and people developing content for these other contexts need new specializations. For example, nurses who develop evidence-based care sheets might need a topic specialization that has sections for evidence, impact on current practices, and bibliographic references.

—<http://docs.oasis-open.org/dita/dita-1.3-why-three-editions/v1.0/cn01/dita-1.3-why-three-editions-v1.0-cn01.html#focus-of-dita>

The fact that the example of evidence-based care sheets clearly would include information from more than one of the abstract types, and that it is proposed as a specialization of topic rather than of concept, task, or reference, suggests a significant shift in thinking on this point, and that may indicate a shift away from abstract information typing towards a more concrete subject-domain approach.

it is easy to see a correspondence between these three types and the reader activities of learning (concept), doing (task), and looking stuff up (reference). The question is whether it provides adequate or appropriate constraints for your content, and whether it is useful for partitioning the complexity of your content system.

This simple triptych is also appealing because it promises (though it does not necessarily deliver) easy composability for content reuse. Some people also maintain that it makes content easier to access for readers, though others (myself included) criticizes it on the ground that it tends to break content down too finely for be useful and robs content of its narrative thread. Also, since neither Information Mapping nor DITA provide any mechanism for constraining how information typing blocks go together to form documents, they do not provide much support of creating and maintaining repeatable rhetorical structures. (This does not mean that authors can't create such structures, but the complexity of doing so falls entirely on them, with the added complexity of having to construct the rhetorical structure out of jigsaw puzzle of abstract information typing blocks.) DITA, however, does allow you to create much more specific content types. See Chapter 41, *Extensible and Constrainable Languages* for more details.

This abstract information typing is entirely distinct from the subject-based information typing of the subject domain. DITA and Information Mapping's approaches are broad and analytical, trying to find commonalities across many different kinds of information. The subject domain is very much specific and synthetic, concerned which how specific pieces go together to successfully describe a particular subject. Both approaches break content up into blocks, and the subject-specific blocks of the subject domain can probably be categorized according to the information typology of either DITA or Information Mapping.

From a design point of view, the question is whether is is easier to arrive at a concrete structure for describing a particular subject by simply observing in concrete terms the particular pieces of information that are required to describe that subject, or by first identifying its abstract type and then deriving a specific type by extension or specialization of the abstract type. This is probably an area in which it depends on the designer's habits of mind. For me, defining specific types in concrete terms based on observed information needs is simple and more straightforward, but the result matters more than the process in these things and you should take which ever approach works for you.

## Rhetorical blocks

Neither semantic blocks nor information typing blocks constitute a complete piece of content suitable for the reader to read. That unit is the unit that is rhetorically complete: that accomplishes the information transfer to the reader. I will call it the rhetorical block. Rhetorical structure is the structure of the rhetorical block. The structure of a rhetorical block is made of up smaller semantic and structural blocks, but it is the structural of the rhetorical block that provides control over the completeness and repeatability of a content

type. The question is, does the rhetorical block have a consistent repeatable structure that we can successfully model and control?

Part of the attraction of information typing theory is that you can divide everything you write into information typing blocks. (It does not matter if your information typing system has three, six, or thirty types, everything fits because there is always one type in the system that is effectively “everything else”.) And because maps are not constrained in an information typing system, there are no limits on how you can put information typing blocks together to form narrative blocks. You can apply information mapping to everything you write. You can apply out-of-the-box DITA (with more or less struggle) to everything you write.

The question is, does doing this enable you to better partition and distribute the complexity of content creation in your organization leading to better content being delivered to your readers? Or would that complexity be better partitioned and distributed by a system that constrained the narrative block? As we have seen, there are many advantages to constraining the narrative block, but the key advantage is repeatability. This is not simply because repeatability allows you to produce quality content more quickly and reliably. It is also because it makes content quality testable in a repeatable manner. You don’t just test the quality of an individual piece, but of a repeatable pattern.

Do all rhetorical blocks have a repeatable pattern? Certainly not. This book is a rhetorical block, but it does not have a repeatable pattern that one could define as a specific data structure. (Hopefully there is a structure to its rhetoric, but it is not one that lends itself to modeling as a set of nested blocks.) Much as this book encourages the use of the subject domain for structured writing, it was not written in the subject domain. It was written in SAM syntax in a small constrained document domain language with a number of subject-domain annotations for things like markup languages, markup concepts, and processing tools.

But not all physical books constitute a single rhetorical block. Many books, such as cookbooks, are collections of related rhetorical blocks, such as recipes. Not only can you define a repeatable rhetorical block for a recipe, doing so provides significant benefits for most of the structured writing algorithms and allows you to effectively partition and direct the content complexity of a cookbook publisher or a magazine publisher that inserts recipes into many different publications.

Where you can reasonably define repeatably structured rhetorical blocks in your content, there are good reasons to do so. This can mean one of two things. It can mean taking the rhetorical blocks that you produce now, identifying those that could be repeatably structured, and developing a structured to contain and constrain them. This may involve moving some content around, since the applications of a repeatable structure will inevitably reveal that some content is in the wrong place, some is missing, and some is superfluous. But overall, this is reasonably straightforward process.

On the other hand, it can mean taking material that is currently in long discursive rhetorical blocks (like text books) and moving it to much shorter and more structured

rhetorical blocks (like recipes or encyclopedia articles). Breaking material into smaller rhetorical units is not a new idea. It has been practiced in encyclopedias and periodicals for centuries and it has greatly accelerated in the age of the Web, which has not only improve our access to information, it has radically changed how we seek and use information, creating a style called information snacking in which we reach for discrete pieces of information as and when we need them, confident that we will always be able to rapidly find and read what we need when the time arises.<sup>2</sup> In other words, people prefer to consume content in shorter rhetorical blocks (not because they need to know less, but because they need to know less at a time), so it makes sense to refactor your content and its architecture into a collection of smaller rhetorical blocks.

This transformation to shorter rhetorical blocks is often called “topic-based writing” but that term has become quite confusing because it is also used to mean a system in which instead of writing in long rhetorical blocks, individual writers write independent information typing blocks which are then compiled into long rhetorical blocks, often by somebody else. The confusion caused by giving these two very different approach the same name is that some people have started to think that topic-based writing means writing information typing blocks and then publishing them separately as if they were rhetorical blocks. This confusions is compounded by DITA’s decision to call their information typing blocks “topics” (Information Mapping calls the “blocks”) and by the fact publishing each information typing block separately is the default behavior when you publish a DITA map to the web.

But while a single information typing block may sometimes be a narrative block all by itself, this is often not the case. Whether one believes in the usefulness of information typing theory or not, it is clear that most useful rhetorical blocks contain more than one type of information.

I have attempted to distinguish topics as complete rhetorical blocks from topics as information typing blocks by coining the term Every Page is Page One topic. Why “Every Page is Page One”? Because a complete rhetorical block, regardless of its length, is a block that an information snacking reader will consume independently of any larger work or collection in which it is embedded. It is page one for that reader. And since information snacking readers use search to dive directly down to the individual page they want, every rhetorical block in your collection is going to be page one for some reader. I explore the design of Every Page Is Page One topics in my book *Every Page is Page One: Topic-based Writing for Technical Communication and the Web*.

But does an Every Page is Page One topic have any more reason to follow a constrained rhetorical block structure than a book? Yes, and, as I explore in *Every Page is Page One*, there is a broad tendency for topics to demonstrate a consistent rhetorical pattern once they become smaller and as accessed in and Every Page is Page One fashion. This makes the information in these topics more accessible to the information snacking reader, both

---

<sup>2</sup>I discuss this change of information seeking and consuming habits in my book, *Every Page is Page One: Topic-based Writing for Technical Communication and the Web*.

because it makes them easier to find and recognize, and because it ensures that they do the job they are supposed to do more completely and consistently.

Additionally, as I explored in Chapter 21, *Information Architecture*, Every Page is Page One topics are part of the way you create a bottom-up information architecture (one in which readers enter by search or following a link and navigate the information set from the point they arrived). Building and maintaining a bottom-up information architecture is much much easier if you do it algorithmically using subject-domain linking and information architecture algorithms. Thus creating subject-domain rhetorical block types for as much of your content as will fit a repeating pattern is key to creating and managing a bottom-up information architecture.

Deciding if you want to model and constrain your rhetorical blocks is therefore one of the most important decisions you will make in designing your content system and demeriting how complexity will be partitioned and directed in that system.

## Granularity

There can be a conflict between ease of authoring and ease of content management. Content management may want to manage content down to a fine level of granularity, especially for purposes of content reuse. This content management algorithm may be best served by managing fairly small chunks of content – semantic units rather than narrative units. But for the writer, something less than a narrative unit can be difficult. It can be difficult for the author to get a sense of how the semantic block they are writing will meet the reader's needs when they don't see the rhetorical block it will fit into. It is hard to create parts rather than wholes unless the parts are really well defined. A writer might carry the whole of an essay in their head, for instance, and be able to structure it well on that basis. But if they are making only parts and cannot see the wholes that will be created, it is hard to correctly structure a part without very clear and explicit guidance.

## Fragments

Another division of content that can occur, mostly in relationship to the management domain, is the fragment. By fragment I mean a chunk of text that is not either a semantic block, a rhetorical block, or a narrative block, but is a block that you want to manage independently of the surrounding text.

For example, in a content reuse scenario, you might want to make an item in a list conditional based on which of the list items applies to different versions of a product.

Individual list items are not semantic blocks. They are just structural blocks of a list. When you make list items conditional, what you are actually doing is creating multiple separate lists with some items in common, and recording them as a single list. You might be able to attach reasonably informative metadata to any one of those lists as a whole,

but there is usually not a lot you can say about list items individually. They are fragments of a list. When you apply conditions to them, then, you are applying those conditions to fragments.

In some reuse systems, including DITA and DocBook, it is possible to apply conditions to arbitrary bits of text – three words in sentence for instance. The block that sets off those three words in a fragment.

Some reuse systems also allow you to reuse arbitrary bits of text from other parts of the content set, simply because the text is the same in each case. Those bits of text would be fragments.

In some cases, you turn an existing structural block into a fragment by attaching management domain metadata to it. In other cases, you have to introduce additional markup into the document to delineate the fragment.

Fragments definitely solve some problems. They are also inherently unstructured and unconstrained. It is very easy to get into trouble with fragments. It is easy to create relationships and dependencies that are hard to manage because they don't follow any structural logic. You should approach their use with great caution and restraint.

## Paragraphs and phrases

Paragraphs are the thing that make structured content the most different from other computable data sets. This is not really because of the paragraph structure per se, but because of the phrases within the paragraphs that we want to annotate. It is rare in any other data set to see a structure floating within the value of another structure. But that is exactly what happens when we annotate phrases in a paragraph.

In {Rio Bravo}(movie), {the Duke}(actor "John Wayne") plays an ex-Union colonel.

In this examples, the annotation on the phrases “Rio Bravo” and “the Duke” float in the middle of the paragraph block. Here is the same thing in XML:

```
<p>In <movie>Rio Bravo</movie>,
<actor name="John Wayne">the Duke</actor>
plays an ex-Union colonel.</p>
```

Here the `movie` and `actor` elements float in the content of the `p` element. In XML parlance, this is called mixed content. If fact, XML breaks the structure of elements down into three kinds:

element content

Elements that contain only other elements.

data content	Elements that contain only text data.
mixed content	Elements that contain both text data and elements.

Mixed content is the reason that most traditional data formats are not a good fit for content. They may be able to model element content and data content, but they lack an elegant way to model mixed content.

Even conventional programming languages have trouble with mixed content. In fact most libraries for XML processing invent an additional wrapper around each string of characters in a mixed content element, effectively representing it as if it were written like this (without mixed content):

```
<p><text>In </text><movie>Rio Bravo</movie><text>,
</text><actor name="John Wayne">the Duke</actor><text>
plays an ex-Union colonel.</text></p>
```

But while this makes the content palatable to conventional languages, it is clearly false to the actual structure of the document. Structured writing is essentially about reflecting the structure of thought or presentation in a narrative, and narratives have a structure that is not shared with other data. Indeed, we might say that all other data formats exist as an attempt to extract information from the narrative format to make it easier to process.

Thus we are taught in school that if we are presented a problem in this format:

John had 4 apples and Mary had 5 apples. They place their apples in a basket. Bill eats 2 apples. How many apples are left in the basket?

You solve it by first extracting the data from the narrative:

$$4 + 5 - 2 =$$

But in content processing, we cannot extract the data from the narrative because narrative is the output we are creating. Thus we have to call out the data (to make it processable by structured writing algorithms) while leaving the narrative intact.

When you move content to the subject domain, you will, in some cases, break down paragraphs and isolate the data. This may be done with the intention of recreating paragraphs algorithmically on output, or of switching from a narrative to a data-oriented reporting of the subject matter. Either way, it makes the data easier for algorithms to handle, and thus makes most of the structured writing algorithms work better. (You may have noticed that the subject domain provides the most constrained and elegant solution to many structured writing algorithms.)

Even so, it is rarely possible to do a complete breakdown of all paragraphs in refactoring content to the subject domain. Most subject domain markup languages make considerable

use of paragraphs and other basic text structures, and annotate phrases within the paragraphs were necessary. Only narrative is capable of expressing the full variety and subtlety of the real world relationships between things, and only narrative is capable of conveying these things effectively to most human readers. (We noted in Chapter 35, *Metadata* that data is created by metadata and that metadata can only finally be defined by narrative.) Even things that can be fully described to algorithms with fielded data must be described to most audiences with narrative, and even though companies like Narrative Science are working on how to turn data into narrative, they are far from producing a general solution – and it is hard to see how you would get one if all data is created by metadata and all metadata is defined, ultimately, by narrative.

Subject-domain structured writing extends the reach of more conventional algorithms into the world of narrative to enable specific structured writing algorithms and to provide constraints to improve the quality of the writing. Unlike ontologies, subject domain structured writing does not attempt to capture the whole semantics of a narrative, just to discipline and structure narrative to achieve specific content creation objectives – a particular partitioning of content complexity.

Every domain needs to annotate phrases. Media domain structured writing needs to annotate phrases to describe formatting. The document domain needs to annotate phrases to describe their role in the document. The management domain needs to annotate phrases to assign conditions or extract content for reuse. The subject domain needs to annotate phrases to describe the subject the phrase refers to.

In planning your markup structures, therefore, it is important to think about which structures in your language need to be mixed content and which do not. Finding ways to avoid mixed content without violating the spirit of the essentially narrative nature of writing can pay dividends in an improved ability to express constraints and to execute virtually all the structured writing algorithms.

On the other hand, some of the most important subject matter that you need to model and make available to algorithms cannot effectively be factored out of paragraphs, particularly while maintaining functional lucidity. Be prepared, therefore, to think seriously about the types of phrases that you will need to annotate below the paragraph level and exactly which domain those annotations should be in.

---

# Chapter 33. Wide Structures

The notion of separating content from formatting works quite well when the content is a string of words. A string of words has only one dimension: length. A printed string, of course, has two dimensions: length and height, since each letter has a height and a width. But the height and the width of letters is a pure media-domain concern. Fitting a one-dimensional string of characters into a two dimensional font on a two dimensional page is one of the first things that gets factored out as we begin to structure content. When we separate content from formatting, we separate the font from the character and are left with a string of characters whose length is measured not in inches but in character count.

Once these formatting dimensions are factored out, it becomes easy to create and manage text in the document and subject domains without thinking about how it will eventually flow onto a page or screen.

But when it comes to content that has dimensions that cannot be factored out, things get more difficult. The main problem cases are:

- tables
- graphics and other media
- preformatted text, such as program listings, that have meaningful line breaks

## Tables

Tables are one of the more complex problems in structured writing, particularly in the document domain. A table laid out for presentation in one publication can easily get messed up when an algorithm tries to fit it into another, as in this example from a commercially published book on my Kindle:

**Figure 33.1. Broken table formatting**

The screenshot shows a Kindle device displaying a table with several rows of XML code and their corresponding results. The table has three columns: the first column contains the XML code, the second column contains the result, and the third column contains a detailed explanation of the result.

<code>3 gt 4</code>	false	(1, 2) eq (1, 2) Type error
<code>"abc" lt "def"</code>	true	Unlike general comparisons, if either operand is the empty sequence, the empty sequence is returned. In this respect, the empty sequence behaves like null in SQL.
<code>doc("catalog.xml")/catalog/product[4]/number lt 500</code>	Type error	If number is untyped or nonnumeric
<code>&lt;a&gt;3&lt;/a&gt; gt &lt;z&gt;2&lt;/z&gt;</code>	true	Each operand of a value comparison must be either a single atomic value,
<code>&lt;a&gt;03&lt;/a&gt; gt &lt;z&gt;2&lt;/z&gt;</code>	false, since a and z are untyped sequences of more than one item, a like strings type error is raised. For example, the expression:	a single node that contains a single atomic value, or the empty sequence. If either operand is a sequence of more than one item, a type error is raised. For example, the expression:

This table is a particularly difficult case as it is not only one wide thing (a table), but it contains another wide thing (preformatted program code). I don't know exactly how this table was marked up, or which domain the content was written in, or how the formatting algorithm failed, resulting in the mess above, but including preformatted text in a table cell creates a no-win situation for a rendering algorithm when it tries to shrink a table to a narrower view port. Does it:

- violate the formatting of the program code by introducing extra line breaks
- give the code the space it needs by squeezing all the other columns impossibly narrow
- resize the columns proportionally and let the preformatted text overlap the next column, but truncate it at the edge of the table
- resize the columns proportionally and truncate the preformatted text at the column boundary
- shrink the entire table so everything still formats correctly, even if it is shown in three point type
- let the table expand outside the viewport so that it is either cut off or the reader has to scroll horizontally to read (Web Browsers tend to take this approach, but will it work on an e-reader? It certainly won't work on paper.)
- make the table into a graphic so that the reader can pan and zoom on it like they do with a large picture. (Some e-books seem to take this approach.)

If you are thinking that there is not one good option in the bunch, you are appreciating the extent of the problem. For existing books being transferred to e-readers, of course, there is not much that can be done to salvage the situation. Those books were probably prepared in a word processor on the more abstract edge of the media domain and the tables were prepared for a known page width in the printed book.

The tendency of readers to use small devices, such as tablets, e-readers, and phones for reading means that wide tables are problematic for new content. On a phone, the amount of a table that is visible on screen at any one time may be so small as to make the table essentially unnavigable, and to make it useless for such common table tasks as looking up values or presenting an overview of a subject at a glance.

Tables can cause problems with height as well as width. While most authors would instinctively know not to import a graphic that was six feet tall, we sometimes create tables that are that long or longer. On a web browser, the reader could simply scroll the table. But as soon as you start scrolling, you lose sight of the column headers and it becomes harder to read data across the table. On paper, it is common to repeat the headings at the top of each page when the table flows over several pages. This works, and it is possible to imitate the effect in a web browser by placing the body of the table in a scrollable frame under a fixed set of headings. But what happens on the page if the height of a table row is larger than the height of the page? Then a single row has to be broken over the page break, leading to questions about how you treat the break in the text of each cell in the row. In traditional typesetting, these things can be massaged by hand on a case by case basis. Getting a rendering algorithm to do it gracefully in every case is a very challenging task.

Creating tables in the document domain creates problems even when the intended output is paper and a sufficiently wide viewport is assumed. Since a table divides content up into multiple columns, there is always a question of how wide each column should be relative to the others, and whether or not the table should occupy the full width of the viewport or not. A table with just a few numeric values, for instance, probably should not be full page width because that would spread the numbers out too far and make comparisons difficult. One the other hand, a table with a lot of text in each cell needs to be full width, and needs to have column widths roughly proportional to the amount of text in the each column. But this is tricky because some columns have side heads which means there are far fewer words in the first column than in the others, but you don't want to compress that column proportional to its word count because then the side headings will be unreadable.

In a media domain editor, which shows the formatting of the content as it will appear on paper, writers can create the table at a fixed width of their choice and then drag the column widths around to get the aesthetics of column boundaries right by eye. But tables created like this are not likely to format correctly on other devices, as Figure 33.1, “Broken table formatting” shows. And if you move the content creation out of the media domain and into the document domain, it is no longer possible to present the writer with a WYSIWYG page width for them to adjust column widths by eye. (You can fake it, but the adjustments made on screen will have no relationship to how the table is actually formatted on any

output page.) At this point you have to leave column width calculation to the rendering algorithm. The best you can do it to give it some hints about how to do its job.

This need to give the rendering algorithm hints about how to fit tables to pages has resulted in the creation of some very complicated table markup languages. Here is an example using the CALS table model, courtesy of Wikipedia:

```
<table>
 <title>Table title</title>

 <tgroup cols="3">
 <colspec colname="_1" colwidth="1*" />
 <colspec colname="_2" colwidth="3*" />
 <colspec colname="_3" colwidth="2*" />

 <thead>
 <row>
 <entry>1st cell in table heading</entry>
 <entry>2nd cell in table heading</entry>
 <entry>3rd cell in table heading</entry>
 </row>
 <row>
 <entry>1st cell in table heading</entry>
 <entry>2nd cell in table heading</entry>
 <entry>3rd cell in table heading</entry>
 </row>
 </thead>

 <tbody>
 <row>
 <entry>1st cell in row 1 of table body</entry>
 <entry>2nd cell in row 1 of table body</entry>
 <entry>3rd cell in row 1 of table body</entry>
 </row>

 <row>
 <entry nameend="_2" namest="_1">cell spanning two columns</entry>
 <entry morerows="1">cell spanning two rows</entry>
 </row>

 <row>
 <entry>1st cell in row 3 of table body</entry>
 <entry>2nd cell in row 3 of table body</entry>
 </row>
 </tbody>
 </table>
```

```
</tbody>
</tgroup>
</table>
```

This sample is for a table with one case of a cell spanning two columns and one of a cell spanning two rows. As you can tell, this is not exactly obvious from the markup. In practice, no one is going to create an CALS table by writing the markup by hand. They are going to use the table drawing tools in a graphical XML editor.

The problem with this is that while the view of the table in the editor looks just like the view of a table in a world processor like Microsoft Word, Word's graphical display is based on the actual page currently set up in printer settings and on the actual font that the document will be printed in. It can therefore show how things will fit in the table on an actual page (allowing the author to make media-domain adjustments to the table). An XML editor cannot know what page size will be chosen or what font will be used when a document is printed. So while the display looks like it allows the same media domain adjustments to be made, this is an illusion and the table will not print as shown on screen. This gives a false impression of the real complexity of the table problem, and hiding complexity from the person who is supposed to deal with it is another way of dumping it on the reader.

Other markup languages take a different approach to tables. For instance, reStructuredText allows you to create a table like this:

Header 1	Header 2	Header 3
body row 1	column 2	column 3
body row 2	Cells may span columns.	
body row 3	Cells may	- Cells
	span rows.	- contain
body row 4		- blocks.

Like the DocBook CALS example, it allow you to span rows and columns, and in this case the effect is obvious from the markup. Equally obvious is that editing the content of this table, or creating a table in this style with any significant amount of text in the cells is going to be very difficult. Nor does this form provide a solution to any of the table rendering challenges described above.

# Alternatives to tables

Structured writing is about partitioning and redirecting the complexity of content so that it is always handled by someone with the attention, knowledge and resources to handle it. As the discussion above demonstrates, table markup dumps a lot of complexity on the writer, complexity they are not fully able to handle because they don't control the final formatting of content in all the media and devices it will be presented on. The best way to partition the problem, therefore, is to get away from writers creating tables as much as possible by factoring out the decision to present information as a table. Let the writer capture the information. Move the decision on whether and how to present it as a table down the road.

What can you factor the content into? There are a number of alternatives, depending on what the table was being used for.

## Alternate presentation

In many cases the use of a table simply isn't necessary. There are other ways to present the content with no loss of comprehensibility or quality. Some tables are just ways of formatting lists, particularly lists with two levels of nesting. If lists are an equally effective way of presenting content, choose lists rather than tables when writing in the document domain.

## Subject domain structure

One way to present the list of ingredients in a recipe is to create a table with the ingredient name aligned left and the quantity aligned right. But as we have seen in our recipe examples, you can create a subject-specific ingredient list structure to capture your ingredient information, which you can then format any way you like for output.

```
ingredients:: ingredient, quantity, unit
eggs, 3, each
salt, 1, tsp
butter, .5, cup
```

A structure like this is a table in a different sense of the word: it is a database table and the `ingredients` structure creates a mini database table inside the body of the content. The difference between this table and a media domain table is that we know exactly what type of information each of the columns contains. This allows the presentation algorithm to make intelligent choices about column widths and all the other rendering issues that arise with tables and pass on appropriate hints to the rendering algorithm for rendering ingredient list tables in particular. Of course, it also supports the alternative presentation choice of using a list rather than a table for ingredients.

Another example where tables are sometimes used in the media domain is procedures. Tables are sometimes used to create side heads for step numbers or high level descriptions of a step, which is then detailed in the right column. Instead of this, use explicit procedure markup which can then be formatted different ways for output. Again, if a table is chosen as the output format, knowing that the contents are a procedure allows the formatting algorithm to provide appropriate layout hinting to the rendering algorithm.

## Record data as data

Many reference works have traditionally been presented as tables on paper. But most such works are really databases. They are not designed to be read but to be queried. That is, they are used to look up individual pieces of data in a large set. For a database of this sort, differential single sourcing requires that you provide the best method of querying the data that is available on each media (which includes the method whose interface fits best in the available viewport). In these cases, the data should not be recorded in tables, at least, not in media domain tables. It should be recorded in whatever database format is most suited to the data and to the kinds of queries that the reader wants to make.

If one of the query mechanisms you want to support for this is printed tables on paper, then the content for those tables should be extracted from the database to create the printed table. Again, the additional semantic information available from the database structure allows the formatting algorithm to supply the appropriate rendering hints to the rendering algorithm.

When you have done all of that, you will probably be left with two kinds of tables that you still have to deal with: Small ad-hoc grid layouts, and table which are database tables, but which are one of a kind, rather than something like ingredients, where the same table structure occurs in every recipe. For these, you will need some form of document-domain table markup. Which markup you choose will come down to how much fancy formatting of tables you want to be able to do, and how willing you are to let the rendering algorithm format your tables without extensive hinting from you.

## Code

There are some texts, particularly computer code and data, in which line endings are meaningful. (Poetry is another example, but its issues are simpler than those of code, so I will stick to talking about code.) Code is a form of structured writing and, in many languages, whitespace – meaning line breaks, spaces, and indentation – are part of the markup that defines the structure of the program. When you present code in a document, therefore, you have to respect line endings.

Furthermore, programmers usually work in a fixed-width font, meaning that all the letters are the same width. They tend to line up similar structures with whitespace to make them easier to read, so using a proportional width font for code in documentation will not only

look weird to programmers, it will mess up that formatting. It will also make the code less recognizable as code, which could reduce information scent.

All of which is to say that computer code, data, and other similar formats where line ends are meaningful have to be presented in a fixed width font and with line breaks where they are supposed to be. That makes code samples wide objects, just like tables, with many of the same issues when it comes to rendering them on small devices. One saving grace is that there are not usually any height issues with code samples.

There is not much you can do to help the rendering algorithm when it comes to code. The options for fitting wide code on a narrow display are to shrink to fit, scroll to view, or truncate. It is not particularly likely that you are going to want your rendering algorithm to make a different choice for different kinds of code. Needless to say, putting a code block inside another wide structure, such as a table, is a recipe for disaster, as Figure 33.1, “Broken table formatting” shows. It would be wisest not to allow this in your markup language design.

What is essential is that your document domain or subject domain markup clearly indicates when a piece of text is code. Preferably it should also indicate what kind of code it is, since knowing this can allow the formatting algorithm to do syntax highlighting for code in a known language, and can allow the linking algorithm to detect and link API calls to the API reference. In some cases it might even allow the conformance algorithm to validate the code to make sure it runs or uses the current version of the API.

## Pictures and graphics

Pictures and graphics are naturally wide objects. There are two basic formats for graphics, vector and raster. Raster graphics are made up of pixels, like a photograph, and have a fixed resolution. Vector graphics are stored as a set of lines and curves and can be scaled to meet any output requirement.

Sometimes the publishing algorithm needs to know how big the graphic is and how large it is supposed to be on the page. With raster files, the resolution of the file is set. However, its size may be in question. Is a graphic that is 600 pixels by 600 pixels a 1x1 inch picture at 600 dpi, a 2x2 inch picture at 300 dpi, or a 6x6 inch picture at 100 dpi? This is important if you are inserting a headshot into a document which will be published on both paper and the Web. You want a 1x1 photo at 600dpi for print, but you don't want that blowing up to a 6x6 photo when you add it to a web page which will display it at a typical 96dpi unless something intervenes to scale it appropriately.

Then there is the question of the intended size of the image, which is a design consideration independent of the resolution of the raster file. The intention of the person who created the picture and the intention of the person using it both play a role here. Diagrams showing complex relationships should not be shrunk down to where the relationships are unreadable. Simple diagrams should not be blown up to the size of

a full page. Diagrams containing text should not be reduced or expanded so that the text becomes invisible or disproportionate to the text on the page. The person using the graphic may have some discretion, based on the role they wish the graphic to play, but their choices should stay within the range prescribed by the creator's intention. In other words, it is important to correctly partition the concerns of the creator and user of a graphic, and to make sure that there is effective communication between them, ensuring that none of the complexity of the relationship gets dropped.

If the rendering algorithm does not know how big a graphic is supposed to be, it has limited choices:

- Show a raster graphic at 100% of its resolution, regardless of whether it fits in the viewport or not (which means either cropping it or forcing the reader to scroll if it goes outside the viewport).
- Scale the graphic to the viewport (which may be stretching it as well as shrinking it).

Since neither of these options will produce consistently good results, we generally need to provide the rendering engine with some information to help it render the graphic appropriately.<sup>1</sup>

The simplest way to supply this information is to include it in the markup that inserts the graphic. Thus HTML lets you specify the height and width of a graphic.

```

```

But do these values represent the size of the graphic or the size at which it is to be displayed in a particular media? In other words, do they define the size of the content (how big the image itself is) or do they define the size of the box that the image should fit in (the viewport for displaying the image)?<sup>2</sup>

DocBook allows you to make this distinction. Its `imagedata` tag supports attributes for specifying the size of the viewport (`height` and `width`) and for specifying the size of the image (`contentheight` and `contentwidth`). The specification also contains additional attributes related to scaling and alignment and complex rules about how the rendering algorithm is supposed to behave based on which combination of these

---

<sup>1</sup>Many web designers take an opposite approach, preparing a graphic to the exact size they intend it to be displayed at on a specific web page layout. This is a completely media domain approach, of course. In structured writing, we generally want a more flexible solution. We have all seen what happens to meticulously designed desktop website when they are displayed on a phone screen.

<sup>2</sup>This is actually quite a complicated question, and the meaning has changed between various versions of HTML. For some hints of the complexities involved, see <http://www.w3.org/TR/html5/single-page.html#attr-dim-width>.

attributes is specified.<sup>3</sup> In other words, it contains a sophisticated language to describe the sizing and scaling of graphics. It not only deals with media domain properties, it actually gives media domain instructions.

Working in the media domain is a problem, of course. It interferes with functional lucidity and it is problematic for differential single sourcing. But there is another issue to consider as well. Sometimes the best approach to differential single sourcing is to use the vector version of a graphic for one media and the raster format for another. For instance, you may want to use the vector version of a graphic for print and a raster version for online media.

For all these reasons, we ought to make a distinction between the source of an image and the rendering of that image. For raster images, the source is the original high-resolution file recorded by the camera, the original screen shot, or the original raster file produced by an image editing program. For vector graphics, it is the original vector drawing file. From these source images, various image renderings may be made.

What about the case where you want a vector version for one media and a raster version for another? One approach is to generate a raster version of the appropriate size from the vector version. This could even be done on the fly at build time, but most of the time we will store multiple versions and select the right one to publish. To create the best image in each format, the artist may even create several original renderings of the same image idea, optimizing each for different uses. For instance, the black and white and color versions of a company logo will often be created separately, because the automatic gray-scale rendering of a color logo may not look good at all. You may also want to use different resolutions of the same raster graphic for different media or for different purposes. This may include manually redrawing a graphic to reduce fine detail, rather than simply scaling it mechanically.

In these cases, how do we include the image in our source content? It is not enough to include the source file and scale it, since there are now several source files. You have to go back to the idea of the image – the image that was in the artist's head, rather than any of the individual renderings of that image. How do we go about it?

In DocBook we can use conditional processing to include a different image file under different conditions:

```
<mediaobject>
 <imageobject condition="epub">
 <imagedata
 fileref=".../graphics/assemble.png" />
 </imageobject>
 <imageobject condition="fo">
```

---

<sup>3</sup><http://www.docbook.org/tdg/en/html/imagedata.html>

```
<imagedata
 fileref="../graphics/assemble.svg"
 contentwidth="4in"
 align="left"/>
</imageobject>
</mediaobject>
```

Here the condition attribute on the `imageobject` element specifies a different file to be used for two versions of a book (this book, actually). The `epub` version is for eReaders, most of which cannot render SVG drawings, and so require a raster format (`PNG` in this case), while the `fo` version is for print publication using the XSL-FO page description language and uses the vector format `SVG` for high resolution rendering in print.

But this approach not only involves the use of media domain markup, it combines it with management domain markup. Is it possible to factor all of this out of the authored format?

To do so we can have the author include the idea of the image rather than a rendering of the image. There are several ways to do this. In fact, this is really the same idea as we saw in the reuse algorithm where we factored out the filename of the content to be included and replace it with a semantic representation of the reason for the content. There we factored out an explicit filename from this example:

```
procedure: Blow stuff up
>>>(files/shared/admonitions/danger)
step: Plant dynamite.
step: Insert detonator.
step: Run away.
step: Press the big red button.
```

and replaced it with a management domain key in this example:

```
procedure: Blow stuff up
>>>(%warn_danger)
step: Plant dynamite.
step: Insert detonator.
step: Run away.
step: Press the big red button.
```

Then we refactored that into a subject-domain assertion of fact in this example:

```
procedure: Blow stuff up
is-it-dangerous: yes
```

```
step: Plant dynamite.
step: Insert detonator.
step: Run away.
step: Press the big red button.
```

Each of these techniques can be applied to the insertion of graphics just as well as the insertion of text.

For instance, suppose you have a constraint that whenever a procedure mentions a dialog box you should have a picture of that dialog box, but your software is delivered on three different platforms and the dialog boxes look different on each.

We can factor out the platform specific version of the image by using a key to insert the image:

```
procedure: Save a file
step:
 From the File menu, choose
 Save. The *Save As*
 dialog box appears.

>>>(%dialog.save-as)
```

Now the presentation algorithm can use a key lookup table for the appropriate platform to select the right version of the Save As dialog box for the version of the documentation you are building. If you port the product to a new platform, all you need is a new set of screen shots and a new key lookup table. You don't have to change the content at all.

There is a pretty simple rhetorical pattern at work in this passage, though. When a step mentions a dialog box, a picture of that dialog box is shown. We can exploit this pattern to factor the insert command out of the content altogether by annotating the mentions of UI components in the text:

```
procedure: Save a file
step:
 From the {File}(menu) menu, choose
 {Save}(menu-item). The {Save As}(dialog-box)
 dialog box appears.
```

Now you can insert the correct screen-shot graphic for the current platform with an algorithm:

```
match procedure/step/dialog-box
```

```
$dialog-box-name = contents
$graphic = find graphic where type = dialog box
 and name = $dialog-box-name
 and platform = $current-build-platform
insert graphic
```

There are several benefits to this partitioning of the image problem:

1. Authors do not have to worry about finding or inserting graphics or remembering the rules about when they are supposed to use screenshots. They just have to remember to mark up the names of dialog boxes when they mention them. (If you only want to show screen shots for certain screens, you only supply the screen shots for those screens, and have your algorithm pass silently over any `dialog-box` entry that does not have a screen shot in the collection. This allows you to adjust to reader feedback about whether more or fewer screen shots are needed simply by adding them to the collection, without changing the content at all – partitioning at work.)
2. Updating graphics for UI changes is simpler because you just update the catalog of images. You don't have to go searching through docs to find images that are affected by the change.
3. You don't need any conditional logic in the text to include the right graphic for the platform.
4. In media where the screen shot would not fit in the viewport, you can suppress the image or handle it a different way.
5. If you decide that most reader won't need to see the screenshot, you can use the markup to create a link to a topic describing the dialog box instead of putting a screenshot inline. In a reuse scenario, you might make different choices for content aimed at different levels of users (only including screenshots in material intended for novices, for instance).

In principle, there isn't any difference between factoring out text this way and factoring out graphics. Indeed, because you are not only factoring out text or a graphic, you are factoring out the decision about whether to express a particular idea with text or a graphics, and can potential make different choices about this for different audiences or different media.

However, when factoring out graphics, there is the added issue of the metadata that describes the various properties of the image and its renderings. One way to handle this is to create a metadata file for each image which provides the needed data for multiple renderings of the image and provides the path to each of the renderings.

The simplest way to implement this is to use an `image include` instruction that points to the metadata file instead of to an image file. This is what I did in writing this book. I noted

above that the DocBook example, which conditionally includes two different versions of the graphics for epub and print was from this book. But this book is not written in DocBook, it is written in SAM. In the SAM source file, the image insertion looks like this:

```
>>>(image ../graphics/assemble.xml)
```

This is not the full factoring out of the graphics as described in the previous example. But note that the file that is being included is not a graphics file. Instead it is an XML file. That file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<image>
 <source>assemble.svg</source>
 <fo>
 <href>assemble.svg</href>
 <contentwidth>4in</contentwidth>
 <align>left</align>
 </fo>
 <epub>
 <href>assemble.png</href>
 </epub>
 <alt>
 <p>A diagram showing multiple pieces being
 combined in different ways to produce different
 outputs.</p>
 </alt>
</image>
```

This XML file describes the idea of the image, listing not only its source file and both of its renderings, but even a text description for use when the graphic cannot be displayed. By including this file instead of an image file, I was able to include the idea of the graphic in my content.

When the content was processed, the presentation algorithm loaded and read the assemble.xml file and used the information in it to generate the conditionalized DocBook file which became the source file for the formatting algorithm (which is implemented by the publisher's existing tool chain).

Could I have factored out the filename assemble.xml as well? Certainly. There are a number of other ways that I could have chosen to represent the idea of the graphic in the content. There are times when it makes a lot of sense to do that. If you are including screen shots in a procedure, for instance, the name of a dialog box is a good way of representing the idea of a graphic that is semantically relevant to the procedure itself.

But in the case of the images in this book, their relationship to the text is a little more arbitrary than the relationship of a screen shot to a step in a procedure, so factoring out the filename would have created an abstraction that was actually more difficult to remember as an author. The point is not to be as abstract as possible, but to combine the highest degree of functional lucidity with the constraints that improve content quality, and that will be different for different kinds of material and for different circumstances.

## Inline graphics

One further wrinkle with graphics is that authors sometimes want to place small graphics in the flow of a sentence, rather than as a separate block object. For instance, if giving instructions that involve the use of a keypad or keyboard, some authors may want to use graphics of the keys rather than simply print the character names. Under certain circumstances, this may make the content easier for a reader to follow.

Inline graphics can cause rendering problems. For instance, they may cause line spacing to be thrown off if the height of the graphics is greater than that of the font used. Inline graphics are something a writer can control and make judgments about when writing in the media domain but which may have unexpected and unwelcome consequences when formatted by algorithms from content created in the document or subject domains.

There are two techniques you can use to minimize problems with inline graphics. The first is to avoid their use altogether, where that is practical. If there is another way to present the same material just as effectively, it is better to choose that option.

The other is to factor out the graphic by using a structure to record its semantics. For instance, instead of including an Enter key graphic like this:

3. Press >(image enter\_key.png) to confirm the selection.

Do this:

3. Press {Enter}(key) to confirm the selection.

This leaves open the choice of how to represent the key in the output, and allows for differential single sourcing. For example, on a display that did not support graphics, or where graphics would be too fussy, the presentation algorithm could render this as:

3. Press [Enter] to confirm the selection.

But in for media where the use of a graphic is appropriate, the presentation routine could use a lookup table of key names and graphics to select the graphic file to represent the **Enter** key.

3. Press  to confirm the selection.

This approach allows the document designer to switch out the graphics used for keyboard keys to find ones that work best on different displays or at different scales. This partitions the graphic maintenance problem to the most qualified person. It is also much easier for the author who does not have to stop to think about which graphic to use. The same approach could be used in another common case, which is describing tool bar icons in a GUI application.

4. Press {Save} (button) to save your changes.

This has all the same advantages as mentioned for keys, with the additional benefit that if the interface designer decides to change an icon or to redefine the whole set of icons, you only have to update the lookup table used by the presentation algorithm. This could also be used to sub in different icons for different platforms if your application is run on more than one operating system. This is much more efficient than using conditional text to import different graphics for different configurations.

This is a good example of using the idea of the graphic rather than the graphic. The idea of the graphic is to represent the Enter key or Save button. This could be done in a number of ways, including a photograph of the key, by the use of a special font that creates the look of a key, or by a textual representation of the key such as [Enter]. The idea of a graphic is to represent a subject. So while you can insert the idea of a graphic in the form of a key or a reference to file that records the idea of a graphic and its implementations, you can also simply identify the subject itself.

As always, a common principle is at work here: better to capture the subject than a resource that represents the subject. Resources may change more often than subjects, and you may want different resources to represent a subject under different circumstance. But as long as the content remains current with its subject matter, the identification of the subject will not change.

---

# Chapter 34. Subject domain structures

Not all content can be meaningfully moved into the subject domain. The subject domain require a repeatable rhetorical structure. Sometimes the nature of your content, or your subject matter, or your argument does not lend itself to a repeatable structure. In those cases, the best you can do is work in the document domain. However, you can add subject domain annotations and indexing – marking up significant subjects in your content by their type and value – across all your subject domain and document domain content, allowing you to apply the conformance, change management, linking, and information architecture algorithms across your entire body of content.

But as we noted in Chapter 32, *Blocks, fragments, paragraphs, and phrases*, you can move a lot of your content into the subject domain, even if it does not currently obey a closely defined rhetorical type, and you can move your content into smaller rhetorical blocks that are easier to apply a repeatable rhetorical structure to. In this chapter we will walk through the steps of defining a repeatable rhetorical type using the recipe example we have been looking at throughout the book.

To begin with, lets take a step back and look at what a recipe might look like if it were not presented in its familiar rhetorical pattern:

Hard Boiled Eggs

=====

A hard boiled egg is simple and nutritious.  
Place 12 eggs in a pan and cover with  
water. Bring water to a boil. Remove  
from heat and cover for 12 minutes. Place eggs  
in cold water to stop cooking. Peel and  
serve. Prep time, 15 minutes. Serves 6.

If we were discerning the rhetorical structure of a recipe for the first time we might then look at several examples and try making dishes ourselves, or observe others doing so, and conclude that it was easier to find and follow a recipe if the ingredients were listed separately and the preparation steps were presented one at a time, and we would come up with a structure that looked something like this:

Hard Boiled Eggs

=====

A hard boiled egg is simple and nutritious.  
Prep time, 15 minutes. Serves 6.

Ingredients

-----

=====	=====
Item	Quantity
=====	=====
eggs	12
water	2qt
=====	=====

### Preparation

---

1. Place eggs in pan and cover with water.
2. Bring water to a boil.
3. Remove from heat and cover for 12 minutes.
4. Place eggs in cold water to stop cooking.
5. Peel and serve.

At this point we have discovered the basic rhetorical pattern of a recipe, but we have not done anything to provide formal constraints for authors to make the format repeatable, so we create a rhetorical block that we name `recipe` and require that it consist of three semantic blocks called `introduction`, `ingredients`, and `preparation`, factoring out the titles of those sections in the process. We also define that the `ingredients` will be presented as a bulleted list and the `preparation` as a numbered lists (both document domain semantic blocks, composed of structural blocks that we need not delve into here).

`recipe: Hard Boiled Egg`

`introduction:`

A hard boiled egg is simple and nutritious.

Prep time, 15 minutes. Serves 6.

`ingredients:`

- \* 12 eggs
- \* 2qt water

`preparation:`

1. Place eggs in pan and cover with water.
2. Bring water to a boil.
3. Remove from heat and cover for 12 minutes.
4. Place eggs in cold water to stop cooking.
5. Peel and serve.

So far what we have is a basic presentation-oriented subject domain content type. It specifies how to present a recipe. This structure provide guidance for authors, which helps with repeatability, at the top level, but when we look closer we realize that there are some other things we would like to make repeatable in the rhetorical model, so that we make sure they are included every time.

Two obvious items are prep time and the number of servings. They are important information that we think should be in every recipe, but currently they are only in the

introduction, where writers could forget them. Also, algorithms can't access them, so if we wanted to make a collection of recipes that took less than 30 minutes to make, we would not be able to find them algorithmically. So we pull those items out into separate fields. Now we can verify that they are included and algorithms can find the information (in other words, we have constrained the interpretation of this information).

```
recipe: Hard Boiled Egg
introduction:
 A hard boiled egg is simple and nutritious.
ingredients:
 * 12 eggs
 * 2qt water
preparation:
 1. Place eggs in pan and cover with water.
 2. Bring water to a boil.
 3. Remove from heat and cover for 12 minutes.
 4. Place eggs in cold water to stop cooking.
 5. Peel and serve.
prep-time: 15 minutes
serves: 6
```

But if we look a little closer we can find other information that we might want to make sure is provided consistently and repeatable, and made accessible to algorithms, and that we might need to differentially single source in different media: the list of ingredients. There are actually three elements to a ingredient listing, the name and quantity of the ingredient, and the unit of measurement for that quantity. Different units of measure are used for different types of ingredients and for large and small quantities in cooking, so we have to specify the unit of measure each time. So we use metadata to turn the list of ingredients into a piece of data:

```
recipe: Hard Boiled Egg
introduction:
 A hard boiled egg is simple and nutritious.
ingredients:: ingredient, quantity, unit
 eggs, 12, each
 water, 2, qt
preparation:
 1. Place eggs in pan and cover with water.
 2. Bring water to a boil.
 3. Remove from heat and cover for 12 minutes.
 4. Place eggs in cold water to stop cooking.
 5. Peel and serve.
prep-time: 15 minutes
```

serves: 6

Once we have got all of the current content into repeatable accessible structures, it is time to think about whether there is any other information that we should be including in our recipes.

Usually during this process, people start to say things like “Wait, shouldn’t we tell them X?” Sometimes X is included in one or two of the examples that you are looking at and you realize it would be useful in all of them. Sometimes it occurs to people for the first time that X might be valuable. The process of formalizing the information you are proposing to offer often produces the realization that there is valuable information that you have not been providing or not providing consistently in your current content.

Also, it is at this stage that you think about differential single sourcing, where you might want to capture content that might be presented differently in different domains, and about reuse scenarios in which you might want to capture information that might be presented in one publication but not another.

So at this point we add in wine and beverage matches and nutritional information to the model:

recipe: Hard Boiled Egg

introduction:

A hard boiled egg is simple and nutritious.

ingredients:: ingredient, quantity, unit

eggs, 12, each

water, 2, qt

preparation:

1. Place eggs in pan and cover with water.

2. Bring water to a boil.

3. Remove from heat and cover for 12 minutes.

4. Place eggs in cold water to stop cooking.

5. Peel and serve.

prep-time: 15 minutes

serves: 6

wine-match: champagne and orange juice

beverage-match: orange juice

nutrition:

serving: 1 large (50 g)

calories: 78

total-fat: 5 g

saturated-fat: 0.7 g

polyunsaturated-fat: 0.7 g

monounsaturated-fat: 2 g

cholesterol: 186.5 mg

```
sodium: 62 mg
potassium: 63 mg
total-carbohydrate: 0.6 g
dietary-fiber: 0 g
sugar: 0.6 g
protein: 6 g
```

Once we are satisfied that we have got the overall structure of the rhetorical block correct, it is time to think about subject annotation. We know that there are a number of algorithms, from linking and information architecture to validation and terminology control that depend on the clear annotation of significant subjects mentioned in the text. So we go through the text looking for mentions of subjects related to cooking. We find that there are mentions of food, utensils, and common cooking tasks, so we add annotation markup for them to our model:

```
recipe: Hard Boiled Egg
introduction:
 A hard boiled {egg}(food) is simple and nutritious.
ingredients:: ingredient, quantity, unit
 eggs, 12, each
 water, 2, qt
preparation:
 1. Place eggs in (pan){utensil} and cover with water.
 2. {Bring water to a boil}(task).
 3. Remove from heat and cover for 12 minutes.
 4. Place eggs in cold water to stop cooking.
 5. Peel and serve.
prep-time: 15 minutes
serves: 6
wine-match: champagne and orange juice
beverage-match: orange juice
nutrition:
 serving: 1 large (50 g)
 calories: 78
 total-fat: 5 g
 saturated-fat: 0.7 g
 polyunsaturated-fat: 0.7 g
 monounsaturated-fat: 2 g
 cholesterol: 186.5 mg
 sodium: 62 mg
 potassium: 63 mg
 total-carbohydrate: 0.6 g
 dietary-fiber: 0 g
 sugar: 0.6 g
```

protein: 6 g

This process can raise some interesting questions. Here it might make us realize that every recipe uses utensils. Then we ask ourselves, if we are listing ingredients separately, why are we not making a list of utensils as well? Wouldn't such a list help readers determine if they had the equipment needed to make a recipe? And we might notice that a close cousin of the recipe, the knitting pattern, does list equipment: the precise knitting needles used.

This then causes us to examine whether we need to change the structure of the rhetorical block to include a list of utensils. Sometimes the answer to questions like this is no. For whatever reason, the cooks of the world have determined over the years that while a list on ingredients is an essential part of the rhetorical structure of a recipe, a list of utensils is not. Don't create additional structures just because you can. We create them because they are useful to our readers or our algorithms.

Finally, you may want to think about whether there is any management domain metadata that you need to add to the model. Usually this will be for management or tracking purposes, and it depends on what decisions you have made about where this type of metadata belongs in your system, in order to best partition the complexity of content management in your system.

This metadata may not be part of any current content examples you are looking at. This should be the last step because once you have formalized the rest of the model as you may find that the subject domain data you have delineated is usable for management and tracking purposes as well, so you won't need to add additional fields just for management purposes. For instance, lots of the subject domain information that is already formalized in the model can be used for managing which recipes you include in particular publications, such as quick and easy meals (less than 20 minutes prep time, fewer than 6 ingredients, fewer than 5 steps) or a low-cal vegetarian cookbook (fewer than 160 calories, no meat ingredients).

This example adds some basic tracking metadata:

```
recipe: Hard Boiled Egg
author: bcrocker
rights: full
season: winter, spring, summer, fall
introduction:
 A hard boiled {egg}(food) is simple and nutritious.
ingredients:: ingredient, quantity, unit
 eggs, 12, each
 water, 2, qt
preparation:
 1. Place eggs in (pan){utensil} and cover with water.
 2. Bring water to a boil.
```

3. Remove from heat and cover for 12 minutes.
4. Place eggs in cold water to stop cooking.
5. Peel and serve.

prep-time: 15 minutes

serves: 6

wine-match: champagne and orange juice

beverage-match: orange juice

nutrition:

serving: 1 large (50 g)

calories: 78

total-fat: 5 g

saturated-fat: 0.7 g

polyunsaturated-fat: 0.7 g

monounsaturated-fat: 2 g

cholesterol: 186.5 mg

sodium: 62 mg

potassium: 63 mg

total-carbohydrate: 0.6 g

dietary-fiber: 0 g

sugar: 0.6 g

protein: 6 g

## Keep it simple and lucid

Most subject domain languages are small, simple, and fairly strict in their constraints. This is as it should be. Since you have to design them, and the algorithms that translate them into the document domain for publishing, you don't want them to be elaborate or full of different permutations of structure. The point of a subject domain language is to partition the gathering of information about a subject from all the processes that you might want to perform on that content, from differential single sourcing, to linking, to information architecture, to content reuse.

If you find yourself needing a similar language for a related subject, it is usually better to create a new equally small, equally strict language for that subject rather than trying to make one language cover both. Subject domain languages get both their power and ease of use from the simple and direct way that the language related to its subject matter. Trying to make one language cover more than one subject takes away from these properties. If you have ever tried to fill out a government form in which different people or entities are supposed to fill out different fields in different ways, you know how difficult it can be to be sure you have filled out all the sections, and just the sections, that apply to you. It is far better to have one form for each case (though obviously you need to make sure that it is very clear which case each form applies to).

A subject domain language should communicate with the author in terms that they understand. This means that the names of structures should make sense to them, but it

also means that how the formal structures break things up should make intuitive sense as well. For an author with experience in the field, a subject domain language should be such a good fit that they don't really feel like they have to learn anything to use it. This vastly increases the functional lucidity of the language leaving more of the author's attention free to focus on content, while at the same time providing constraints and guidance that make sure that things are complete and consistent.

Be careful not to take things too far, though. Once you get started, it is easy to get carried away with breaking things down into finer and finer pieces to formally describe the subject matter in finer and finer detail. Remember that all of this is wasted unless it helps better partition and distribute complexity in the content system by enabling the algorithms you need to do that partitioning. Remember too that writing and conformance are among the algorithms. Repeatable rhetorical structures as the heart of a successful structured writing system, so make sure that you create the patterns that make your rhetorical patterns repeatable. But don't go further than that. Making your markup mysterious, difficult, or tedious to create in the name of more precisely modeling the subject matter does more harm than good if reduces functional lucidity without creating a compensating increase in quality or efficiency. Again, this is an improper partitioning of the complexity of content: increasing complexity without any benefit in transferring complexity to a more appropriate person or process.

---

# Chapter 35. Metadata

We live in the age of metadata, so much so that the word metadata has almost come to replace the word data itself and has come to be applied to almost any form of data that describes a resource. For example, we hear a lot about law enforcement getting access to metadata related to phone calls, which simply means the data about which number called which number and for how long.

The standard definition of metadata is data that describes data, but that definition misses the central point. Metadata does not merely describe data, metadata creates data. Metadata turns an undifferentiated set of values into useful data by constraining the interpretation of those values.

In the document domain, the ingredients of a recipe are just list items. The content of those list items are just strings of characters.

```
section: Ingredients
* 12 eggs
* 2 qt water
```

Adding subject domain markup allows us to specify to algorithms exactly what these strings mean.

```
ingredients:: ingredient, quantity, unit
eggs, 12, each
water, 2, qt
```

This subject domain markup is metadata, and it turns what were just list items into a set of ingredient data. It is not that the data existed and the metadata came along afterward to describe it. It is that the data exists as data only because the metadata is there to describe it. While a human reader can recognize that the values in the list are ingredients, based on their familiarity with the recipe format, an algorithm sees only strings of characters until we apply the metadata that lets it recognize them as ingredients.<sup>1</sup>

Structured writing is writing that obeys constraints and that records the constraints it obeys, so as to create an interface between partitions in your content system. In other

---

<sup>1</sup>I am talking here about ordinary algorithms: simple rules written by humans to govern the processing of well defined data. As I noted in Chapter 7, *Quality in Structured Writing*, advances in AI are making inroads on enabling robots to read the way humans do. An AI is an algorithm, and a sufficiently advanced AI algorithm might not need this level of explicit metadata to recognize list items as ingredients. Technically, though, such an AI would still be depending on metadata to interpret this data. It would just be using the same metadata that humans use, which we might fairly, if briefly, characterize as a combination of grammar and memory. Until such AIs are available to us, though, structured writing allows us to add more explicit metadata to content to make it accessible to simpler algorithms.

words, structured writing is the application of metadata to content which turns content into data by constraining the interpretation of the content. If structured writing is writing that obeys constraints and records the constraints it obeys, Metadata is the means by which those constraints are recorded. Metadata constrains the interpretation of values in content, making them accessible as data.

This is the basis for all partitioning of the complexity of content creation. The metadata we attach to content allows us to pass the content to different people or processes without letting any of the complexity drop. (This includes search engines and other downstream processes that we do not control but can inform.) We cannot partition complexity safely if we drop any of the complexity in the process. Metadata is how we ensure that all the complexity has been successfully transferred from one partition to another.

Saying the the subject domain metadata makes the text of the ingredients list more precise data is not the same as saying that a list structure is not metadata also. It is document domain metadata. It formally identifies a piece of text as a list item. It allows us to partition the formatting of lists from the writing of recipes, and all kinds of other content. This makes it easy to write algorithms that recognize lists which allows you to reliably format list items in whatever media you choose to publish in. The operations we can perform on list item data are obviously far less sophisticated than those we can perform on ingredient data, but list item data are still data and still created by metadata.

There is a very important point here: The same set of values, the same string of letters, words, and numbers, can be turned into different kinds of data by applying different kinds of metadata to them. This means that we can choose what kind of data we turn our content into by choosing what type of metadata we apply to it. Moving content from one structured writing domain to another means turning it into different kinds of data by applying different metadata to it. By turning it into different kinds of data, we make it accessible to different kinds of algorithms.

## The recursive nature of metadata

Metadata is a confusing concept because metadata is recursive. If metadata is the data that describes (and thereby creates) data, it is also data itself. And since data is created by metadata, this means that metadata is itself created by metadata. In other words, if the line `ingredients:: ingredient, quantity, unit` turns the ingredient lines into metadata in this markup:

```
ingredients:: ingredient, quantity, unit
 eggs, 12, each
 water, 2, qt
```

Then what makes `ingredients:: ingredient, quantity, unit` a piece of metadata and not just another string of characters? Whatever it is, it is also metadata, and

the content depends on that metadata as well, since it cannot exist without its metadata and that metadata cannot exist without the metadata that defines it.

In structured writing, we add structure to content to replace the things we have factored out. That structure is metadata to the data that is the text of the file. But if we store that file in any kind of repository, the information that identifies the file in that repository is metadata to the file as a whole. If the structure of the file is described by a schema, the schema is also metadata for the file.

But we're not done yet because the specification of the schema language is the metadata that tells you what the schema means. And then of course, there is the specification of the markup to consider. The XML specification is part of the metadata tree for every XML document in existence. And we are still not be done, because the XML specification uses a formal grammar description language, called BNF. The BNF specification is metadata for the schema language description.

How do we break out of this infinite recursion of metadata? Data is information that has been formalized for interpretation by algorithms. Fortunately, human beings can understand natural language without that degree of formalization. Eventually, then, we reach a point where the last piece of metadata is not described by metadata but by human language. That human language document essentially bootstraps the whole metadata cascade that eventually yields pieces of data that can be unambiguously interpreted by algorithms.

So, every piece of data has a spreading tree of metadata supporting it, which, if traced to its roots, eventually leads to plain language documents that explain things in human terms. Thus the XML specification combines plain English definitions with BNF, if we go to the BNF specification we will find the plain English definitions that describe BNF.

## Where should metadata live?

One of the great questions about metadata is where it should live: with the data it describes or separate from it? We looked at this issue a little in Chapter 23, *Content Management*, but now that we have a better understanding of what metadata is, we can give a more complete answer to the question.

The issue of where metadata should live is closely related to the issue of how responsibilities are partitioned in your content system. Since metadata is how complexity is transferred safely from one partition to another, the responsibility for creating the metadata lies with the person or process in the originating partition, while the metadata requirements are dictated by the needs of the receiving partition. By adjusting how the system is partitioned, you can adjust how onerous the metadata requirements are on any one actor in the system. The location of the metadata, therefore, is essentially dictated by the partitioning. It is located in the place that achieves the desired partitioning of complexity for the particular content system.

Most early graphic file formats only stored the image. Most modern format also store extensive metadata about the image. The pictures you take with your digital camera include lots of information about the camera and the settings that were used to take the shot, all of which can help rendering algorithms and graphic editing applications handle the raw image data better. Having that metadata embedded in the file ensure that the picture and its metadata stay together. Separating them would greatly complicate the system. Keeping the image metadata in the image file is better partitioning than keeping it separate.

Unfortunately, tools are often designed with other priorities in mind. For one thing, many tool developers think almost exclusively in relational database terms. The idea that you could store metadata anywhere other than relational tables is foreign to them. For another, system vendors have a vested interest in a partitioning of the process that requires every user to be interacting with their system all day long because this forces people to buy every contributor a licensed seat for their tool. Both these things encourage them to implement models in which the metadata is separate from the content, ensuring that you need access to the system to have access to the metadata.

For example, should the history of a file be stored in the file or in the repository? Storing it in the file lessens the file's dependence on the repository and makes it more portable. But a repository vendor may prefer to sell you a system in which to uninstall their repository would be to lose all your file history. If file status information is only stored in a workflow system, for instance, it is very hard to move away from that system. If it is stored in the file, it is easy to move away, and also to edit when not connected to the system, which can save you on licenses.

In the case of the photo, the metadata is in the file because the camera is the best placed instrument to record it. This is the best partitioning of the complexity of the recording and transferring this information, both in terms of its convenience of creation and in terms of ease of access and management. The location of the metadata should be determined by the best partitioning of the content system, not the convenience of a tool vendor.

Writing your content in the subject domain means that more of your metadata is stored in the same file as the content, increasing its independence and portability. Also, as we have seen, the use of subject domain structures can lessen the need for management domain structures for algorithms like single sourcing and content reuse, which reduces the need for external management domain metadata. All of this contributes to improved functional lucidity, referential integrity, and change management.

But this does not mean that all metadata related to a piece of content belongs in the content file. For example, when we import a graphic into a document, we often give it a caption or a title and we specify the size it should be displayed at. These are all metadata about the graphic, but it would not make sense to include this information in the graphic file itself. This information is not actually describing the graphic itself, it is describing its relationship to the current document. In a different document, the same graphic might be displayed at a different size with a different caption or title. This information is therefore

included in the file that imports the graphic, not the graphic file itself. In other words, the metadata is stored on the right side of the relationship between the two content objects. This is simply good partitioning. (In Chapter 33, *Wide Structures* we looked at another way of partitioning this metadata, again based on the nature of the relationship between the files, and on the way that the complexity of the content system is partitioned.)

Storing the metadata on the right side of the relationship means that there are definitely types of metadata that belong on the repository or content management side of the relationship as well. If you store your content in a version control system (VCS) such as GIT (something that is increasingly popular in the “treat docs like code” movement), the VCS will record the difference between each version of the file you submit as well as who did each commit of the file. This will allow the VCS to do valuable management things like tell you exactly who changed an individual line of a file and on what date they changed it. Storing such metadata in the file itself would make it impossible complex to author. Again, the metadata is stored on the right side of the relationship and the partitioning of complexity is correct.

## Ontology

Finally it is worth saying a word about ontology. Ontology (in the information processing sense) is an attempt to create a formal mapping of the relationships between entities in the real world such that algorithms can draw inferences and reach conclusions about them.

In many ways, therefore, an ontology is an attempt to do for algorithms what content does for humans. After all, one of the main reasons that we read is so that we can understand the world better, understand what various objects and institutions are and how they relate to each other, statically and in action, so that we can decide what to do.

In some sense, therefore, ontology is the ultimate in subject domain markup. Indeed, one should also be able to generate human-readable content from an ontology, given a sufficiently sophisticated algorithm and a sufficiently sophisticated ontology.

All of this is very much outside our scope in this book. Subject domain markup is an attempt to capture certain aspects of the subject matter of a work. But it is not an attempt to model the argument of a work. Consider the passage:

In {Rio Bravo}(movie), {the Duke}(actor "John Wayne") plays an ex-Union colonel.

Here the subject domain markup formalizes the fact that Rio Bravo is a movie and that “the Duke” is a reference to the actor John Wayne. It does not model the relationship between the two. An ontology would want to model the “starred in” relationship between John Wayne and Rio Bravo, whereas subject domain structured writing is normally content to leave this to the text.

Similarly, this subject domain markup does not bother to denote that Union is a reference to both a country and its armed forces, and that colonel is a rank in those armed forces. It does not denote these things because this particular markup language is concerned with movies and these facts are entirely incidental to the movie business. Actors, directors, and movies are significant subjects in the movie review domain. The names of nations and their armies that figure in the plot of individual movies are incidental in that domain. A full ontological treatment of the passage above, however, would need to model those relationships.

Structured writing does make certain aspects of content clear to algorithms, but not with the intention of making it possible for the algorithms to make real-world inferences and decisions based on the information in that content. It only does what is necessary to partition and redirect content complexity in a content system in which human authors to use algorithms as tools to improve the quality of the content they prepare for human readers.

---

# Chapter 36. Terminology

Whether metadata is stored internally in the content or externally in a CMS, it is important to be consistent in the terms used. You cannot constrain the interpretation of content if you don't contain the metadata values you use to define it. Similarly, it is important that the terminology used to name subjects in the content itself be named consistently to avoid confusion for readers. And if you are supplying a large-scale top-down navigation scheme for your content, that scheme is also going to be concerned with the names of things, and with choosing the right terms to name the subjects so that readers can find them.

For all of these reasons, large scale content projects need to exercise some control over terminology. (Control of terminology is important for translation as well, but that is outside the scope of this book.) Managing terminology is a complex problem, and it is one of those problems where a simplistic approach can result in lots of dropped complexity landing on the user in the form of incomprehensible language, incorrect classification, or poor connections between units of content. The biggest danger here is thinking of terminology as a simple data management problem. Separating words from their role in sentences and paragraphs certainly makes them easier to fit into the traditional rows and columns of data management, which makes the problem look a lot simpler than it really is. Once you see it this way, it seems straightforward to get a bunch of people in a room, spin up a list of words and their definitions, and declare it as your corporate taxonomy, but this is a false partitioning of the problem.

The principal difficulties in establishing terminology are:

- Human beings have a fairly small “use vocabulary” and we reuse words all the time. In safety critical functions like air traffic control or the operating room, we train everyone to use a special unambiguous vocabulary (which is generally undecipherable to the layperson) but for most uses, the words we want to control are likely to be used in multiple ways that are not easy to disambiguate formally.
- People in different fields (even within the same organization) often use different terms for the same concept. What the chef calls “pork”, the farmer calls “pig”. What the English call “boot”, North Americans call “trunk”. Trying to force everyone to use the same term means forcing them to say things that don’t make sense in their own field.
- People in different fields (even within the same organization) often use the same words to mean different things. What the conference organizer calls a function is not what the programmer calls a function. What a programmer calls a function is (in more subtle ways) not what a mathematician calls a function.
- People in one field may have ten terms that make fine-grained distinctions among things that people in another field lump together under a single term. For instance, programmers make a distinction between subroutines, functions, methods, and

procedures. I know of one documentation project on which it was mandated that all of these should be called “routines”. This was OK most of the time, but because a problem when function pointers were introduced into the product. (You can’t use any of the other terms with “pointer”.) Enforcing a single term obscured a distinction that turned out to matter.

In short, very little of our terminology is truly universal. The meaning of words changes depending on the context you use them in and the audience you are addressing. This is, in fact, at the core of how language works and why it is different from other forms of data. Content tells stories, and what words mean depends on the context in which they are used in the story. Stories are not as precise as formal data, but the only way we have to define formal data is with stories. This is why the spreading tree of metadata that supports and explains any point of data always ends with a human-language document. All of structured writing is an attempt to bring some small part of the orderliness and manageability of data to stories to improve the quality and consistency of stories so that we can communicate more effectively.

Thus terminology control is important, because it is key to making our structures and metadata work, but at the same time it must be done with care and sensitivity and a real sense of the limits within which it is possible to control the terms we use to tell stories. Good taxonomies always confine themselves to specific domain and define their terminology within the confines of those domains, but even within all but the most strictly controlled domain (such as the operating theater or the control tower) there are still commonly many shades of meaning on individual words which are only fully disambiguated by the story.

We have seen how we can use context to determine the meaning of block names in structured writing. Terms don’t have to universally unique to be clear if we can identify them unambiguously in context. Terms that cannot easily be controlled universally can often be controlled in context. In other words, the terminology problem becomes much more tractable when it is partitioned appropriately. Structured writing is the ideal tool for partitioning the terminology problem by providing the context in which terms are understood. We have seen that annotations can be made more precise by identifying the type of subject being named. Here again we are adding context to terminology to allow us to control it better in its local domain. One of the virtues of the subject domain is that every subject domain content type and indeed every subject domain block type provides context for controlling the interpretation of terminology within it. (Which is, once again, an instance of using structured writing to constrain the interpretation of content.)

# Top-down vs. bottom-up terminology control

There are two ways to partition the terminology control problem: top-down or bottom up. The principle tools of top-down management are controlled vocabularies and

taxonomies. A controlled vocabulary is essentially a list of terms and their proper usage within a specific domain.

A taxonomy is a more elaborate scheme for controlling and categorizing the name of things. Taxonomies are frequently hierarchical in nature, defining not only the terms for individual things but the names for the classes of things. Thus a taxonomy does not just list sparrows and blue jays and robins, it also classifies them as birds, and birds as animals, and animals as living things. A good taxonomy should be specific to the domain for which it is intended. (Blue Jays and Cardinals occupy a very different place in a baseball taxonomy than in an ornithological taxonomy.) As a classification scheme, a taxonomy may be used not only as the basis for controlling vocabulary, but also as a basis for top-down navigation of a content set.

Alternatively, you can control terminology from the bottom up. To do this, you use subject domain annotations in your content to highlight key terms and place the usage in the appropriate domain.

In {Rio Bravo}(movie), {the Duke}(actor "John Wayne") plays an ex-Union colonel.

In the passage above, the annotations call out the fact that “Rio Bravo” is the name of a movie and that “the Duke” is the name of an actor called “John Wayne”.

This is taxonomic information. It places “Rio Bravo” in the class “movie” and “John Wayne” in the class “actor”, with the added information that “the Duke” is, in context, an alternate term for the actor John Wayne. By placing these terms in these classes, it makes it clear that “Rio Bravo” in this context is not the Mexican name for what Americans call the Rio Grande or any of the several American towns named Rio Bravo, and that “the Duke” refers to John Wayne and not to The Duke of Wellington (who was often called by that nickname) or any of the possible meanings of “Duke”. In other words, the taxonomy is embedded in the content. The subject domain internalizes taxonomic metadata.

To appreciate why this might be useful, we can look at some of the difficulties of maintaining and enforcing vocabulary constraints.

As mentioned above, constraining vocabulary is hard because the same term can mean different things in different contexts, and different terms can mean the same thing in different contexts. If you attempt to build a taxonomy from the top down it can be very difficult to anticipate the various meanings a term may have in different contexts. Even if you study existing texts, there is no guarantee that you will exhaust all the possibilities, and such searches are tedious and time consuming.

Secondly, even once you have defined your taxonomy from the top down, there is the question of how it is going to be enforced. You can require authors to use terms from the taxonomy, but how is that going to work? Do you expect them to carry the entire

taxonomy around in their heads? And as they are writing, do you expect that they will recognize that they are using a word not on the taxonomy every time they refer to a subject covered by the taxonomy? Any attempt to comply with such requirements is going to create a lot of mental overhead for writers, and, as we have noted (Chapter 8, *Writing*) dividing author's attention has a negative impact on content quality. In other words, you are dumping a highly complex task that depends on a huge amount of data, onto each individual writer and requiring them to pay attention to it continuously in every word they write. This is the antithesis of good partitioning of complexity in the content system.

There are mechanical solutions that attempt to catch terminology problems, but the fact that words can mean so many things in different contexts means no such process can get it right all the time.

An alternative to defining and enforcing a taxonomy from the top down is to let it emerge, in a disciplined way, from the content itself. The key to this is that in structured writing, particularly in the subject domain, you annotate those things that are significant to your content. If the terminology you are trying to enforce is not terminology that is significant to your content, you are probably wasting your time.

If you have writers annotate the significant subjects in their content as they write, they will be annotating those very terms whose vocabulary you want to control. So when they mention the name of a bird like blue jay or robin, they annotate it as {blue jay} (bird) or {robin} (bird). By specifying the type of the subject you are partitioning the vocabulary in context.

Of course, this does not ensure that writers are using the right terms for birds. It only highlights the terms they are actually using. To achieve consistency, you will need to audit the list of birds mentioned in your content. To get the current list of bird names from the content, you simply have an algorithm scan your content for terms annotated as "bird" and compile them into a sorted list. (This is one of those query-oriented algorithms that I mentioned at the end of Chapter 10, *Processing Structured Text*.) The person responsible for terminology control can then quickly see if any incorrect or unexpected terms are being used, and can either edit them or call them to the writer's attention. This partitions the vocabulary control problem in a way that makes it easier for both roles to do their job.

And notice that we are not introducing a new role here. Even if we demand that authors follow the taxonomy up front, they are still going to make mistakes, so if you are not willing to live with that dropped complexity, you are going to need a terminology audit anyway. This approach simply makes the audit easier to perform by highlighting the use of terms, and assigning them to the correct domains, in the text.

But this approach not only works to audit conformance to an existing taxonomy. It can be used to create and maintain a taxonomy. If an author mentions a new bird that you would not have thought to include in your taxonomy, it will show up in the list for the next audit. The terminology control person can then decide if that bird should be added to the taxonomy or not.

“Adding to the taxonomy” does not necessarily imply that a separate top-down taxonomy is being maintained apart from the content itself. It is possible to regard the current annotated content set itself as the taxonomy. After all, it contains all the approved terms and their types. Any list of terms you generate from the content are just reports on the taxonomy, not the taxonomy itself. By storing the report from the previous audit, you have a basis of comparison to see if any terms have been added to the taxonomy since the last time the audit was run.

This approach may not always give you a sufficiently firm control over your taxonomy, however, so you may use annotations in your content simply are an indication that new words are being used, and then add them (or not) to the official taxonomy list.

If you maintain an official taxonomy separate from the content, it is trivial to have an algorithm compare the audit list to the official taxonomy and alert you every time a new bird is mentioned. Effectively, now, your taxonomy is bubbling up from your content. Your authors are not having to worry about whether the terms they are using are in the taxonomy or not, as long as they mark up what type of thing they are naming. You can set up the audit and compare algorithm to run on a regular basis (nightly perhaps) and have it alert your terminologist every time a new term is added to the content. The terminologist will be able to evaluate if it is being used incorrectly, or if it should be added to the taxonomy. This way, the terminology audit is almost entirely automated, with action required from the terminologist only when an unfamiliar term is used in context.

It is possible that some authors will forget to annotate some birds, or will annotate them incorrectly (as something other than a bird). But we can easily catch most of these mistakes as well. Incorrect annotations will tend to show up as anomalous entries in other annotation categories. If “sparrow” suddenly shows up in the “bards” category beside “Shakespeare”, the terminologist will get an alert that a new bard had been mentioned and it is easy for them to spot that it was annotated incorrectly.

In the cases where there are genuine name conflicts between two different domains, you can make a list of such conflicting names and use an algorithm to compile a list of all tagged instances of those names to review for incorrect tagging. For failure to tag at all, you can have an algorithm scan the entire content set for unannotated instances of annotated terms. If you get a lot of false hits (the same words using in a different context) you can annotate them to be ignored:

It reminded me of {Robin} (ignore) Hood.

The point of partitioning and redistributing the complexity of a problem is to direct the complexity to those best placed to handle it. This sometimes means directing complexity to a central person or process with specialized or particular knowledge, such as knowledge of the organization’s formatting standards. But just as often it means distributing the complexity outward or down to people in the field who have the contextual knowledge needed to make correct decisions. Taxonomy is a third case, one

in which only the writers in the field can tell you what ideas need to be expressed and only a central process can coordinate and disseminate information about what terms have been used and decided upon. Any process that does not allow communication in both directions, and does not allow writers to make good decisions in context, will end up dumping complexity on the reader. Equally important is how complex each person's task becomes, how many things you are asking them to do at the same time. Even if writer and terminologist are the same person, the approach to terminology control outlined above separates the tasks in time, so that the writer does not have to keep the taxonomy in their head as they write. In fact, it means that no one has to keep the taxonomy in their head at all.

Another useful conformance tool that can come out of the bottom-up approach is a stop list. A stop list is a list of terms that should not be used, but sometimes are. It can be used by an algorithm to scan content for inappropriate vocabulary. Stop lists can only really be created bottom up. You can't anticipate or ban every term anyone might ever come up with. You should only ban terms that are both problematic and which occur frequently. (The chance of false hits – of banning terms that are perfectly legitimate in other contexts – rises with every word you add to the stop list.) With a bottom up approach to terminology control, you get an accurate measure of which terms are being misused, and the frequency and nature of the misuse. This is an excellent basis for compiling a useful stop-list.

Also, because subject annotation can specify the type of a term (that is distinguish between {Blue Jays} (baseball-team) and {Blue Jays} (bird) you can make your stop list type specific: banning words used in one sense but not another. This can greatly reduce false hits, which is important because people tend to abandon the use of conformance tools if they produce very many false hits, as we can see from the very infrequent use of the grammar checkers in word processors.

This approach does not make all vocabulary constraint problems go away, but it does have a number of advantages.

- It turns an up-front taxonomy development effort into an permanent part of the content development process. This not only reduces the spike in effort, it means that the taxonomy is based on real experience writing real content and that it is continually maintained as subjects and business objectives change. Indeed, if you are already annotating your content to support any of the other structured writing algorithms, you essentially get taxonomy development, maintenance, and control almost for free. (Someone does have to review the reports, make the edits, and add to the canonical taxonomy, of course.)
- It improves functional lucidity by not forcing authors to refer to the taxonomy while writing. If you are already annotating subjects for other reasons, you are imposing no additional burden on authors at all.
- By improving the consistency of the annotations, it makes all the other algorithms that rely on them more reliable as well. (This is one of the greatest virtues of the subject

domain. Subject domain markup can serve multiple algorithms, meaning you get the benefits of multiple algorithms with less cost.)

---

## **Part IV. Languages**

---

## Table of Contents

37. Markup .....	313
Markup vs. regular text .....	314
Markup languages .....	317
Concrete markup languages .....	317
Abstract markup languages .....	317
Instances of abstract markup languages .....	318
Concrete languages in abstract clothing .....	319
The ability to extend .....	320
The ability to constrain .....	321
Showing and hiding structure .....	321
Hybrid languages .....	322
Instances of hybrid markup languages .....	325
38. Patterns .....	326
Design implications .....	327
39. Lightweight Languages .....	332
Markdown .....	333
Wiki markup .....	334
reStructuredText .....	335
ASCIIDoc .....	336
LaTeX .....	337
Subject Domain Languages .....	340
40. Heavyweight markup languages .....	341
DITA .....	342
DocBook .....	346
S1000D .....	347
HTML .....	347
Subject domain languages .....	348
41. Extensible and Constrainable Languages .....	349
XML .....	349
DITA .....	350
Specializing between domains .....	354
DocBook .....	356
RestructuredText .....	357
TeX .....	357
SAM .....	358
SPFE .....	360
42. Constraint Languages .....	362

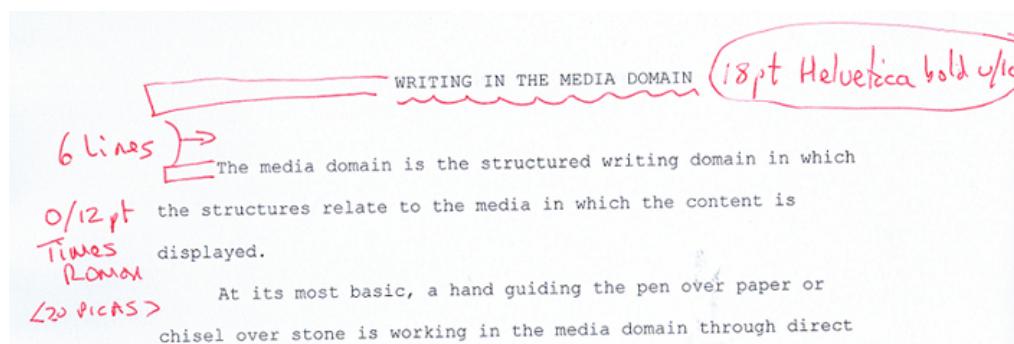
# Chapter 37. Markup

If structures in structured writing consist of nested blocks, the way those blocks are expressed in text and created by writers is (most of the time) markup. Markup is, essentially, the insertion of metadata into a text to constrain the interpretation of that text. The trick is to design markup that is both precise enough and complete enough to provide and constrain the information needed by the next partition in the content system while still being lucid enough for authors to create without imposing on their attention so much that it affects the quality of their content. We have been looking at structured writing examples expressed in markup all through this book. Now let's take a look at markup itself.

Markup has long been part of how we partitioned and distributed content creation tasks. For centuries, scribes worked directly in the media domain, using pen and ink to inscribe words and pictures on papyrus or velum. With the printing press, however, came a fundamental partitioning of the publishing process. Authors no longer worked directly in the media domain. While authors were still directly placing ink on paper, at first by pen and then by typewriter, they were no longer preparing the final visual form of the content. That task has been partitioned off and directed to the typesetter.

To tell the typesetter how to create the final visual form, document designers had to add additional instructions (metadata) to the author's manuscript. The designers did this using typesetter's marks, and the process was called "marking up" the document. We still use "marking up" to describe how structured writing is done today and the term "markup language" to describe the languages used for most structured writing.

**Figure 37.1. Printer's markup**



The writer preparing a manuscript for typesetting was working in the document domain, indicating basic document structures like paragraphs, lists, and titles, without any indication of how they should look in print. The designer then wrote a set of instructions for applying formatting to those structures – a formatting algorithm. Then the typesetter executed that algorithm by setting the type which the printer then used to print final output.

This is pretty much exactly what we do today when we create an HTML page and specify a CSS stylesheet to supply the formatting instructions. Those instructions are then executed by the browser to render the content on screen or paper.

Actually, we are getting ahead of ourselves here. A better analogy to old style typesetter's marks is an HTML page with the styles specified inline.

```
<p style="font-family: serif; font-weight: bold;
font-size: 12pt">
```

You can see that this markup is very very similar to the old typesetters marks in Figure 37.1, “Printer’s markup”.

All writing programs have to store the writing in files. There are two possible file types they can use: binary and text.

For all intents and purposes, a binary file is one that can only be read or written by a computer program, usually the program that created it. Open up a binary file in a text editor and you won’t be able to make heads or tails of it. And even if parts of it look like plain text, editing those sections and saving the file is likely to result in a corrupt file that the original application can no longer open.

A text file, by contrast, is one that you can open in a text editor and actually be able to read and write without breaking it. But to express structure in a text file, you need a way to interpolate information about structure into the text. The way we interpolate structure is with markup – special sequences of text characters that are recognized as defining structure rather than expressing text.

In the following snippet of HTML, the markup characters are shown in bold while the plain text is shown in regular type.

```
<h1>Moby Dick</h1>
<p>Herman Melville’s <i>Moby Dick</i> is a long
book about a big whale.</p>
```

A markup parser recognizes the markup characters and builds a structure that has the text and nested structures as content. A processing application then access that data, applying rules to the structures define by the markup, as we saw in Chapter 10, *Processing Structured Text*.

## Markup vs. regular text

Some markup languages make the distinction between markup and regular text completely explicit. An example of explicit markup is an HTML tag. Tags are set off by opening and closing angle brackets:

**<h1>**

HTML uses open angle brackets < to indicate the start of markup and closing angle brackets > to indicate the end of markup and a return to regular text. The use of a slash after / the opening < indicates and end tag, which marks the end of a structure:

**<h1>Moby Dick</h1>**

Actually the recognition of markup in HTML is a little more complicated than that, but that is more detail than we need to get into here. The point here is simply that there are certain sequences in the text which trigger a processing program (generally called a “parser”) to recognize when markup starts and when it ends.

What if you want to enter these “markup start” characters into the text of your document? You can’t just type them in because the parser will think they are markup. To fix this, markup languages either define “escape” characters, that signal the parser to treat the following character as text, or they include markup for inserting individual characters in a way that won’t be confused with markup characters. HTML takes the second approach. To include a < character in HTML, you use what is called a “character entity.” A character entity is a code for a character. It begins with & (another markup start character), followed by a character code and ending with a semicolon. The character entity for < in HTML and XML is &lt;. (“lt” is short for “less than”, the name of the < character.)

**<p>In HTML, tags start with the &lt; character.</p>**

This will display as:

In HTML, tags start with the < character.

Since & is also a markup start character, we need to replace it with a character entity as well if we want to include it literally. To include a literal & you use the character entity &amp;.

**<p>In HTML, character entities start with the & character.</p>**

This will display as:

In HTML, character entities start with the & character.

To include the literal sting &amp; therefore, you would write &amp;amp;.

**<p>The character entity for an ampersand is &amp;amp;.</p>**

This will display as:

The character entity for an ampersand is &amp;;

Other markup languages do not make such an explicit distinction between text and markup. For example, in Markdown a numbered list is created by putting numbers in front of list items:

1. First
2. Second
3. Third

Here the numbers are markup. That is, the Markdown processor recognizes them as indicating a list and will translate them into a structure in HTML like this:

```

 First
 Second
 Third

```

But numbers followed by a period are only markup in a certain context – the beginning of a line. Elsewhere, they are plain text. There is no need to escape numbers followed by periods when they occur elsewhere in the text. Thus the following markdown file:

1. First comes 1.
2. Second comes 2.
3. Third comes 3.

will translate to HTML as:

```

 First comes 1.
 Second comes 2.
 Third comes 3.

```

In Markdown, then, markup is not fully explicit. It is a pattern recognized in context. Rather than thinking of markup as being something entirely distinct from text, therefore, it is better to think of markup as being a pattern within a piece of text that delineates its structure. In some cases those patterns may be absolute, meaning the same thing everywhere, and sometimes they may be contextual, meaning one thing in one location and something else in another location. Sometimes the markup characters may be entirely

distinct from the text characters, and sometimes a pattern in the text may serve as markup as well.

## Markup languages

A set of markup conventions taken together constitutes a markup language. Markdown, DocBook, and JavaDoc are all markup languages. However, each of these languages recognizes markup in a different way. & may be a markup start character in HTML and XML, but it is just a plain text character in reStructuredText.

We can usefully divide markup languages into three types which I will call concrete, abstract, and hybrid.

## Concrete markup languages

A concrete markup language has a fixed set of markup patterns that describes a fixed set of content structures. For example, Markdown is a concrete markup language that uses a markup that is designed to mimic the way people write plain text emails. Here is the passage about *Moby Dick* written in Markdown:

```
Moby Dick
=====
```

Herman Melville's Moby Dick is a long book about a big whale.

In Markdown, a line of text underlined with equal signs (=) is a level one heading. A paragraph is a block of text set off by blank lines. Emphasized text is surrounded with underscores or asterisks.

In Markdown, these patterns correspond directly to specific document structures. You cannot invent new structures without inventing a new version of Markdown.

## Abstract markup languages

An abstract markup language does not describe specific concrete document structures directly. It describes abstract concrete structures which can be named to represent structures in any domain.

XML is an example of an abstract markup language.<sup>1</sup> The markup in an XML file does not directly indicate things like headings or paragraphs. Instead, it indicates a set of

---

<sup>1</sup>The formal term for a language like XML is “meta language”, a language for describing other languages. In calling XML an “abstract” language, I am focusing on a different property, its use of structures that are not parts of a document but

abstract structures called elements, attributes, entities, processing instructions, marked sections, and comments.

None of these abstract structures describes document structures in any of the structured writing domains. Instead, specific markup languages based on XML (or its cousin, SGML) indicate subject, document, management, or media domain structures as named instances of elements and attributes.

Here is the *Moby Dick* passage again, this time in XML (more specifically, in Docbook):

```
<section>
 <title>Moby Dick</title>
 <para>Herman Melville's <citetitle>Moby Dick</citetitle>
 is a long book about a big whale.</para>
</section>
```

The structure described by the XML syntax here is that of an element (`section`) which contains two other elements (`title` and `para`), one of which contains text (`title`), and one of which (`para`) contains a mix of text and another element (`citetitle`). There is no separate syntax for titles or paragraphs as there is in Markdown. Everything is an element and every element has a name. To define specific document domain structures, you use named elements. This allows you to create any set of named elements you like to represent any structure you need.

Unlike a Markdown parser, an XML parser does not see paragraphs or titles. It sees elements. It passes the elements it finds, along with their names, down to a processing application which is responsible for knowing what “section”, “title”, and “para” elements mean in a particular markup language like DocBook. The parser is common to all XML-based languages, but the processing application is specific to DocBook. Thus while processing a concrete language like Markdown is generally a one step operation, processing an abstract language like XML is a two step operation, with the first step being to parse the file to discover the structures defined by elements and the second step to process those structures according to a specific set of rules applicable to a particular markup language.

## Instances of abstract markup languages

This means that DocBook is a instance of the abstract language XML. XML defines abstract structures. DocBook defines concrete structures by giving names to XML

---

generic containers. A meta language needs such abstract containers. But I find that the term “meta language” is not helpful to most readers, so I have chosen instead to focus on this property of using abstract structures as opposed to the concrete structures of a language like Markdown.

elements. Many common markup languages are instances of XML<sup>2</sup>. XML is virtually the only abstract language used for content these days, so it is the only abstract language I am going to talk about.

Since specific markup languages like DocBook are instances of XML, I need to revise my earlier statement: We can usefully divide markup languages into **four** types: concrete, abstract, instances of abstract, and hybrid. In fact (spoiler alert), lets revise it again: We can usefully divide markup languages into **five** types: concrete, abstract, instances of abstract, hybrid, and instances of hybrid.

You can't write directly in a abstract language like XML, only in an instance. You can write directly in some hybrid languages, though not all of them.

Or to put it another way, as a designer of markup languages your can either:

- Design a concrete language from scratch (or modify and existing one)
- Use an abstract language (probably XML) to design a concrete language.
- Use a hybrid language to design a concrete language.

As a writer, you will either use:

- A concrete language with ad-hoc syntax (like Markdown)
- The concrete parts of a hybrid markup language (like ReStructuredText) without extensions
- A concrete language (like DocBook) based on an abstract language (probably XML)
- A concrete language (like the recipe markup language used in many of the examples in this book) based on a hybrid language (SAM in this case)

## Concrete languages in abstract clothing

The key defining characteristic of an abstract language is the use of abstract named structures like XML elements. All XML elements share a common markup start sequence followed by the element name. This creates a named block of content. But concrete languages can use named blocks too. For example, JavaDoc, a concrete language for describing Java APIs, uses named blocks using @ as a markup start character:

```
/**
```

---

<sup>2</sup>Sometimes also referred to as “applications” of XML, though this usage was far more common in the days of SGML.

```
* Validates a chess move.
*
* Use {@link #doMove(int theFromFile, int theFromRank,
int theToFile, int theToRank)} to move a piece.
*
* @param theFromFile file from which a piece is being moved
* @param theFromRank rank from which a piece is being moved
* @param theToFile file to which a piece is being moved
* @param theToRank rank to which a piece is being moved
* @return true if the move is valid, otherwise false
*/
boolean isValidMove(int theFromFile, int theFromRank,
int theToFile, int theToRank) {
 // ...body
}
```

In this sample, `@param` and `@return` are named blocks. But in JavaDoc, there is a fixed set of named blocks that are defined as part of the language. You can't create a new language by defining your own block names. By contrast, XML itself defines absolutely no element names. Only instances of XML, like Docbook, define element names.

A particularly notable example of a concrete language in abstract clothing is HTML. HTML looks a lot like an instance of XML, but it is not. An XML parser cannot parse most HTML. HTML is nominally an instance of SGML but never did quite conform to it. HTML generally requires a specific HTML parser, such as is found in all browsers. XHTML is a version of HTML that is an instance of XML. HTML5 actually supports two different syntaxes, one of which is an instance of XML and one of which is not, meaning that it has both a concrete syntax and a syntax which is a instance of an abstract language. (This is consequence of having made a mess early on and having to live with it for evermore. A lesson for all markup language designers.)

## The ability to extend

The downside of concrete languages is that their concrete syntax defines a fixed set of structures. If you want other structures, there is no way to create them short of inventing your own concrete language, or a variant of an existing one, and coding the parser and all the other tools to interpret that language. Designing new concrete languages is non-trivial because you need to make sure that any combination of characters that the writer may type is interpreted in an unambiguous way. Some versions of Markdown, including the original, contain ambiguities about how certain sequences of characters should be interpreted, which obviously detracts from its reliability and functional lucidity.

If you want to define your own structures to express the constraints that matter to your business, you need an easier way to do it. Abstract languages like XML make this much easier. You just write a schema describing the structures you want, and any algorithms

you need to process those structures. (I'll talk about schemas in Chapter 42, *Constraint Languages*.)

## The ability to constrain

Extensibility allows you to add structures to a language but does not place restrictions on where they can occur.

Extensibility allows you to have elements called `ingredient` and `wine-match`. Constraints allow you to require that `ingredient` only occurs inside an `ingredients` structure and that the content of the `ingredients` structure must be a sequence of `ingredient` elements and nothing else. Constraints lets you say that writers can't put `wine-match` in the `introduction` or as a step in the `preparation`, they can only put it as a child of `recipe` after the `servings` field and before the `prep-time` field. Constraints allow you to require that every `recipe` have the full list of nutritional information.

Constraints are what allow you to partition the content system by creating reliable interfaces between different people and processes. All markup languages have constraints. With a concrete language, you get the constraints that are built into the language. Abstract languages allow you to define your own structures, and your own constraints. However, as we shall see in Chapter 41, *Extensible and Constrainable Languages*, not all languages that are extensible are also constrainable.

## Showing and hiding structure

To reliably create structured content, writers need to see the structures they are creating. In the media domain a WYSIWYG interface shows you the media domain structures you are creating by visually rendering them on screen. But what about in the other domains? The document domain creates abstract document structures that are deliberately separated from their formatting. The subject domain creates subject-based structures that don't have a one-to-one relationship with any organization or formatting of a document. The management domain creates structures that have nothing to do with the representation of content at all. How does the author get to see these structures when writing in these domains?

This is a big problem with XML, the only abstract language in widespread use today. XML tends to hide structure. As an abstract language, an XML document is a hierarchy of elements and attributes – not the concrete subject, document, management, or media domain structures the author is supposed to be creating. Those concrete structures are present in the markup because their names are there, but they are not visually distinguished the way the basic document structures are in a concrete language like Markdown. And XML syntax is verbose, meaning that there is a lot of clutter in the raw text of an XML document, which makes it hard to discern both the structure and the

content. XML's verbose syntax and strict hierarchy were designed to make it easy to parse and to help guard against transmission errors. It was not designed to be a format to write in. Thus it does a poor job of partitioning parsing issues from authoring issues.

To remove that clutter, many authors use XML editors that provide a graphical view of the content similar to that of a word processor. But while graphical XML editors removes visual clutter, they also hide the structure. Even if the author is supposed to be working in the document domain or the subject domain, the editor is now displaying content in the media domain. This greatly reduces the functional lucidity of the document domain or subject domain language and encourages backsliding into the media domain.

And then there are the problems that arise when you try to edit the graphical view of an XML document. Underneath is a hierarchical XML structure, but all you can see is the flat media-domain view of the graphical editor. Editing or cutting and pasting structures you can't see can be frustrating and time consuming. You can learn to do it, but even when you learn, the process is still more complicated than it should be. This is a case of unmanaged complexity being dumped on the author.

Concrete markup languages like Markdown, on the other hand, show you the structure you are creating and are simple to edit.

## Hybrid languages

There are significant advantages and significant disadvantages, then, in both concrete and abstract languages. Hybrid languages try to find a middle way.

By hybrid, I mean a language that combines both abstract and concrete markup in one language. A hybrid language has a base set of concrete syntax describing basic text structures but it also has abstract structures such as XML's elements and attributes that can be the basis of extensibility and constraint.

An example of a hybrid markup language is reStructuredText. Like Markdown, it has a basic concrete syntax for things like lists and paragraphs. But it also supports what it calls "directives", which are essentially named block structures. For example, a codeblock in reStructuredText looks like this:

```
.. code-block:: html
:linenos:

for x in range(10):
 print(x+1, "Hello, World")
```

reStructuredText provides an extension mechanism that allows you to add new directives. But while reStructuredText directives are similar to XML elements, reStructuredText

defines a core set of directives for common document structures. The code-block directive above is not an extension of reStructuredText, it is part of the core language.

Because it defines a large set of document-domain directives, reStructuredText is inherently a document domain language. You could, of course, add subject-domain directives to it. Most document-domain languages in use today include some subject-domain structures, reflecting the purpose they were originally designed to serve. Nonetheless, reStructuredText is inherently document domain.

Another important note about reStructuredText is that it has no constraint mechanism. You can add new directives, but you can't constrain their use, or the use of the predefined directives.

I have developed a hybrid markup language which is designed to be both extensible and constrainable. I call it SAM (which stands either for Semantic Authoring Markdown or Semantic Authoring Markup, as you please). SAM is the language I have been using for most of the examples in this book.

Here is the *Moby Dick* passage written in SAM:

```
section: Moby Dick
```

```
Herman Melville's {Moby Dick} (novel) is a long
book about a big whale.
```

In SAM, as in Markdown and most other concrete markup languages, a paragraph is just a block of text set off by whitespace. Thus there is no explicit structure named p or para.

At the beginning of a line, a single word without spaces and followed by a colon creates an abstract structure called a block. The word before the colon is the name of the block. Thus section: above creates a block structure named “section” just as in XML an element named <section> would create a structure named “section”. Blocks can contain blocks or text structures such as paragraphs and lists.

The hierarchy of a SAM document is indicated by indentation. Thus the paragraph in the sample is contained in the section block. Using indentation to indicate containers helps make the structure of the document visually clear and removes the need for end tags, which reduces verbosity, making the text easier to read.

Within a paragraph, curly braces markup a phrase, to which you can attach an annotation in parentheses. Here the phrase “Moby Dick” is annotated to indicate that it is a novel. SAM also supports decorations, like the underscores in the Markdown example, so in the media domain “Moby Dick” could have been written \_Moby Dick\_.

SAM is not intended to be nearly as general in scope as a purely abstract markup language like XML. It is meant for semantic authoring (which is to say, structured writing). As

such it incorporates a number of shortcuts to make writing typical structured documents easier.

In a typical document, a block of text (larger than a paragraph) typically has a title. So in SAM, a string after a block tag is considered to be a title. That means that the markup above is equivalent to:

```
section:
 title: Moby Dick

 Herman Melville's {Moby Dick} (novel) is a long
 book about a big whale.
```

Unlike RestructuredText, however, SAM does not have an extensive set of predefined named blocks. It only predefines the basic text structures that it provides concrete syntax for. You can write an entire book in reStructuredText without defining any new directives. The most you can write in SAM without defining any blocks is a single paragraph or list.

SAM is designed to have a constraint mechanism, allowing you to write a schema to define what blocks and annotation are allowed in a SAM document. This includes constraining the use of the concrete syntax as well. SAM thus represents a different type of hybrid from reStructuredText which is extensible but not constrainable. This reflects the fact that reStructuredText is intended as an extensible document domain language, whereas SAM is intended primarily as a neutral starting point for designing subject domain languages (though you can design document domain languages in it as well).

Also unlike RestructuredText, SAM is not intended to have its own publishing tool chain. The SAM parser outputs an XML document which can then be processed by XML publishing tools by transforming it into an appropriate document domain language. This book was written in SAM, using a simple document-domain language I created for the purpose, with a number of subject domain annotations. That language was transformed into a semantically equivalent XML document by the SAM parser. That XML document was then transformed into DocBook according to the publisher's specifications (the publisher has a number of constraints on the DocBook they use that are not expressed in DocBook itself). From that point on, the publisher's existing DocBook tool chain took over.

Most concrete markup languages, at least those designed for documents, try to make their marked-up documents look and read as much as possible like a formatted document. SAM is designed to be easy and natural to read, like a concrete markup language, but it is also designed to make the structure of the content as clear and explicit as possible while requiring the minimum of markup. This is why it uses indentation to express structure. Indentation shows structure clearly with a minimum of markup noise to distract the reader's eye.

Because it is meant specifically for authoring, a SAM parser outputs XML, which can then be processed by the standard XML tool chain. Below is how the SAM markup above would be output by a SAM parser:

```
<section>
 <title>Moby Dick</title>

 <p>Herman Melville's <phrase>
 <annotation type="novel"/>Moby Dick</annotation>
 </phrase> is a long book about a big whale.</p>
</section>
```

## Instances of hybrid markup languages

I said above that this book is written in SAM, but that is not quite accurate. As noted above, you can't write anything in an abstract or hybrid language directly. You write in instances of those languages. Thus DocBook is an instance of the abstract language XML. You can write documents in DocBook. We do say, of course, that we write documents in XML, but that statement is, if not wholly inaccurate, certainly non-specific. Saying the a document is written in DocBook tells you what constraints it meets. Saying it is written in XML merely tells you which syntax it uses, which is a whole lot less informative.

So, to be more specific, this book is written in a markup language written in SAM, one that I created for the specific purpose of writing this book. That markup language was then transformed by a processing application into DocBook, which is the markup language that the publisher uses for producing books. From there is was processed through the publisher's regular DocBook-based tool chain to produce print and e-book output.

Choosing the style of markup you will use is very much tied to the choices you make about the structures you want. In the following chapters we will look at a variety of markup languages using a variety of markup styles. As always, of course, your choices should be dictated by what works best for the overall partitioning and direction of complexity that you are seeking to achieve in your content system.

---

# Chapter 38. Patterns

We noted that lightweight markup languages like Markdown rely on patterns in the text rather than explicit markup characters to delineate basic structures like paragraphs and lists. XML, on the other hand, makes a strict distinction between what is markup and what is text. Markup is always recognized explicitly by markup sequences that cannot occur as normal text in any part of the document.

All the same, even XML markup is defined by patterns. If you look at the XML spec you will see that XML is defined by a series of patterns which are described using a pattern notation called EBNF. The difference between XML syntax and Markdown syntax is that in XML, patterns are defined absolutely, so that a markup start character is a markup start character no matter where it occurs, whereas in Markdown, the patterns are defined in context, so that a \* character starts a list item only at the beginning of a line, not the end.

Since markup is all about patterns, we are not confined only to the patterns defined by the markup language itself. There are lots of patterns that occur naturally in content and we can recognize those patterns in our content and act on them with algorithms even if they are not formally marked up in our markup language.

A common example of this is the pattern of a URL. In most Web-based editors, if you enter a URL as plain text (no markup identifying it as a URL), the editor will recognize the pattern and will turn the URL into a clickable link in the HTML output.

Recognizing patterns rather than forcing the author to mark them up explicitly can increase functional lucidity significantly. Consider something as simple as a date. A date is a complex piece of data consisting of at least three elements, the year, month, and day. (It gets more complex if you include the day of the week or the time.) If you need those individual components of a date for processing purposes, you might be tempted to require dates to be marked up like this.

```
date: {2016}(year)-{08}(month)-{25}(day)
```

or

```
date:
 year: 2016
 month: 08
 day: 25
```

This is fully explicit markup. It places every piece of the data in its own explicit structure. However, it lacks functional lucidity.

A date in the format 2016-08-25 is already a piece of structured markup. In fact, it is a standardized piece of structured markup defined by the international standard ISO 8601.

The only problem in identifying a date as such is that in another context 2016-08-25 could be a mathematical expression that resolves to 1983. But we can eliminate this potential confusion if we isolate the context so that we can recognize when an ISO 8601 date is being used, and then use the ISO 8601 format specification to isolate the year month and day components of that data. In other words, all we need to do is this:

```
{1941-12-07}(date) is a day that will live in infamy.
```

It also means that if the author wants to enter the date in another format, we can annotate it with the ISO 8601 format to make the meaning clear to algorithms.

```
{December 7, 1941}(date "1941-12-07") is a day
that will live in infamy.
```

Of course, a sufficiently clever algorithm could recognize December 7, 1941 as a date as well, because that too is a fully predictable pattern. One of the defining features of patterns is that you can recognize the semantic equivalence between two patterns that express the same information. You can increase the functional lucidity of your markup by recognizing a wider variety of semantically equivalent patterns in content without having to impose uniform syntax on authors.

## Design implications

XML recognizes the value of patterns. The XML data types standard<sup>1</sup> defines a number of common patterns, and the XML schema language (XSD)<sup>2</sup> allow you to define patterns for use in your own markup. It calls these patterns “simple types” (as opposed to “complex types”, which are composed of multiple nested elements). Thus you can specify in XSD that a date field is of the simple type xs:date. The xs:date data type is a pattern that is based on ISO 8601, so you can do following in XML (as long as the schema defines the type of the date element as xs:date):

```
<date>1941-12-07</date> is a day that will live in infamy.
```

It is important to understand what this does. If the type of element date in your XML markup language is defined as xs:date then an XSD schema validator will validate 1941-12-07 as a valid date but will reject <date>last Thursday</date> as invalid. It will not break the date down into its separate year, month, and day components. If you need that, you will have to get your algorithm to do that for itself. But your algorithm can do that reliably because the markup constrains the interpretation of the

---

<sup>1</sup><https://www.w3.org/TR/xmlschema-2/>

<sup>2</sup><https://www.w3.org/standards/xml/schema>

date, establishing that it is indeed an ISO 1806 date. Most programming languages will have library functions that know how to manipulate ISO 8601 dates, so you probably don't have to do any work yourself to get the year, month, and day components of a date.

For other patterns, most programming languages have a library called "regular expressions" which you can use to rip a pattern apart and get at the pieces. (XSD actually lets you define new patterns by specifying a regular expression for the pattern you want to match.) Regular expressions will let your algorithms decode pattern in your content if its interpretation is sufficiently constrained by its context, removing the need to ask authors to explicitly break down many types of information into separate fields.

For example, in our most of our recipe examples, we have been explicitly breaking down the items in an ingredient list into ingredient, quantity, and unit of measure like this:

```
ingredients:: ingredient, quantity, unit
eggs, 12, each
water, 2, qt
```

That format is not terribly onerous to write, but there is still a small functional lucidity penalty here. An author might more naturally write:

```
ingredients::
eggs 12
water 2qt
```

Ingredient name, quantity, and unit of measure are still clearly distinct here. Any human reader can see them instantly. The trick is to get an algorithm to read them the same way. This trick can be accomplished fairly easily using a regular expression:[\*2}

```
(?P<ingredient>.+?) (?P<quantity>\d+) (?P<unit>qt|tsp|tbsp) ?
```

(The regular expression I am using here is deliberately simple and does not include a full list of all the possible names for units of measures. A more precise expression would likely be required for production quality code, but it would be distractingly verbose for our present purpose.)

To show you how this works, here is a set of ingredient values:

```
eggs 12
water 4qt
salt 1tsp
butter 2tbsp
```

pork chops 4

Here is the result of applying the regular expression above to those values (this is a report generated by regex101.com, a fantastic site for designing and testing regular expressions):

Match 1

```
Full match 0-7 `eggs 12`
Group `ingredient` 0-5 `eggs`
Group `quantity` 5-7 `12`
```

Match 2

```
Full match 8-17 `water 4qt`
Group `ingredient` 8-14 `water`
Group `quantity` 14-15 `4`
Group `unit` 15-17 `qt`
```

Match 3

```
Full match 18-27 `salt 1tsp`
Group `ingredient` 18-23 `salt`
Group `quantity` 23-24 `1`
Group `unit` 24-27 `tsp`
```

Match 4

```
Full match 28-40 `butter 2tbsp`
Group `ingredient` 28-35 `butter`
Group `quantity` 35-36 `2`
Group `unit` 36-40 `tbsp`
```

Match 5

```
Full match 41-53 `pork chops 4`
Group `ingredient` 41-52 `pork chops`
Group `quantity` 52-53 `4`
```

As you can see, the regular expression has broken each ingredient down into ingredient, quantity, and unit fields, just like our explicit markup did. The conventional way of writing ingredients, in other words, is just as structured as our formal way. It just takes a line of code to pull them out of the string. Once again we are partitioning and transferring a bit of complexity from the author to the information architect or content engineer.

Essentially, any expression that we commonly use in text which follows a well defined pattern is already structured text, and we don't need to structure it again to treat it as structured. All we need to do is to use structure to place it into context so that we can recognize it reliably. For example, our regular expression for breaking down the components of an ingredient listing will only work when applied to text that we know for

sure is supposed to be an ingredient listing. Thus we use normal structured text structures to delineate the ingredients list and each list item. This provides the necessary context to use the ingredient list pattern to pull out the ingredient, quantity, and unit fields that we want without forcing the author to make them explicit in the text.

Avoiding unnecessary markup of already recognizable patterns both simplifies our markup design and increase the functional lucidity of our markup languages.

If we were creating this recipe structure in XML and using XML Schema (XSD) as our schema language, we could create a data type for the ingredient field that would verify that the text entered met that pattern we are expecting:

```
<xsd:element name="ingredient">
 <xsd:simpleType>
 <xsd:restriction base="xsd:string">
 <xsd:pattern value="(.+?)(\d+)(qt|tsp|tbsp)?"/>
 </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
```

This type definition would ensure that if the writer entered ingredients in the wrong form or using the wrong names for units, an error would be raised. For instance, these ingredient list items would trigger an error:

```
<ingredients>
 <ingredient>12 eggs</ingredient>
 <ingredient>water 2 cups</ingredient>
 <ingredient>butter 4 lumps</ingredient>
</ingredients>
```

The first would fail because the pattern would not find the quantity where it expected to. The next two would fail because cups and lumps are not on the list of strings recognized by the pattern (which probably should be extended to allow cups, but not lumps). In the XML schema example, the names of the fields that were present in the original regular expression are omitted because XML schema is not going to break the values apart and put them in separate fields for us. As far as XML schema is concerned, this is just the definition of a single element. It would be up to a processing application to break the pieces apart and make separate fields of them. It is design intent for SAM that its schema language will let you do just this: define multiple fields with a regular expression that tells you how to break a string apart and names the individual fields to extract from the pattern.

By the way, though I have used the ingredients listing as an example here, I probably would not use this pattern in a production system. While the eggs, 12, each

structure is a tiny bit unnatural, its explicitness makes it less error prone. It is less likely that writers will get the order of information wrong, and then have to pause to deal with the error message they get. Asking them to break the items of information apart explicitly here is probably the lesser of two evils. The use of patterns is an aid to functional lucidity and you need to balance naturalness against explicitness in the way that best supports functional lucidity in each case.

---

# Chapter 39. Lightweight Languages

I commented in Chapter 37, *Markup* that XML does a poor job of partitioning authoring concerns from parsing concerns. XML is a fully general abstract markup language and its syntax and its logical model are designed to support the creation of any markup language for any purpose whatsoever. This need to support any possible kind of structures makes for a heavy verbose language.

XML's predecessor, SGML attempted to be both fully general and to allow for the definition of specific markup languages with very light syntax that was easy to author. Unfortunately, the mechanism for creating such languages was very complex and difficult to understand, and made parsing SGML much more complicated. SGML is still used in a few niches, but it never achieved the kind of widespread use that XML has done.

Still, XML remains a problem for authors, and a number of languages have been created to try to address the problems created by XML's verbosity. Collectively these are called lightweight markup languages. Lightweight markup languages are designed to use a lightweight syntax, that is, one that imposes a minimal burden on the readability of the raw text of the document. They are all far less general in their application than XML. In effect, they partition by omission, leaving out capabilities not needed by their users in order to make their syntax and structures simpler and easier to understand. This is fine, of course, as long as you don't need the capability they omit. The key to correct partitioning here is to choose the language that has the best balance between the capability you need and the simplicity of authoring you want.

The primary appeal of lightweight markup languages rests on two related properties.

- They have a high degree of functional lucidity at the syntactic level (easy to write) and often at the semantic level (what it means) as well. It is usually possible to read the raw markup of a lightweight language more or less as if it were a conventional text document.
- They can be written effectively using a plain text editor (as opposed to an elaborate structured editor with a graphical editing view). This means that the editing requirements are lightweight as well.

Most examples also come with a simple processing application that creates output directly in one or more output formats. This means that they have a lightweight tool chain that is easy and inexpensive to implement.

There are a number of lightweight markup languages. Some of the more prominent include:

# Markdown

The most prominent of the lightweight languages, and arguably the lightest-weight, is Markdown. Invented in 2004 by John Gruber as a way to quickly write simple web pages using syntax similar to that of an text-format email, it has spread to all kind of systems and now exists in multiple variants that have been adapted for different purposes.

“Adapted for different purposes” mostly means that people have created versions with specific syntax and semantics in addition to those of Gruber’s first version. For instance, the code sharing site GitHub has adopted “Git Hub flavored markdown” as the standard format for user-supplied information on the site, such as project descriptions and issues, and has added syntax specific to tracking issue numbers and code commits for projects, allowing the automatic generation of links between commits and the issues that relate to them.

Markdown is a simple document-domain language. While its semantics are essentially a subset of HTML, it is more squarely in the document domain than HTML since it lacks any ability to specify formatting or even to create tables (though various MarkDown flavors have added support for tables).

One of the recurring patterns of technology development, and certainly markup language development, is that when some simple format becomes popular because of its simplicity, people start to add “just one more thing” to it, with the result that it either becomes more complex (and thus less attractive) or more fragmented (and thus harder to build a tool chain for). Markdown is definitely going the route of fragmentation at the moment (though a standardization effort, in the form of CommonMark is also under way). There is even a project to add semantic annotation to MarkDown as part of the Lightweight DITA project.

None of this is a reason not to use MarkDown where its structures and syntax make it an appropriate source. MarkDown provides useful constraints on the basic formatting of a web page both by factoring out direct formatting features and by providing a very limited set of document domain features. These constraints help keep all contributors to a site from indulging in extravagant non-standard formatting or overly elaborate text structures. It successfully partitions basic Web formatting, though not much else.

Markdown is also used in conjunction with static site generators such as Jekyll which use Markdown for basic text structures in concert with templating languages like Liquid (essentially a set of management domain structures, yielding a result that is comparable to a document/management domain hybrid in its capabilities, though not in its style). Tom Johnson provides a side by side comparison of DITA and Jekyll in a series of posts on his blog beginning with <http://idratherbewriting.com/2015/03/23/new-series-jekyll-versus-dita/>.

Markdown does not provide any kind of subject domain structures or constraints at all. This may be a welcome feature when comparing it with more complex document domain

languages, many of which do include some subject domain structures which can be confusing to some writers, or which writers may abuse to achieve formatting effects.

The inspiration for its syntax, text-format emails, has faded to obscurity, so it is not clear that everyone automatically knows how to write Markdown, as was the original design intent, but a lot of it remains obvious and intuitive, meaning that, within its limits, Markdown has good functional lucidity. It works well if you don't need any of the stuff it partitions away.

## Wiki markup

Another popular lightweight format is Wiki markup, introduced by Ward Cunningham in 1995 as the writing format for WikiWikiWeb, the first Wiki.<sup>1</sup> Wiki markup is similar to Markdown in many respects (most lightweight languages share the same basic syntax conventions, based on the imitation of formatted document features in plain text documents). What makes Wiki markup distinct is how it is tied into the operation of a Wiki. One of its most notable features is how linking is handled. In the original WikiWikiWeb markup, any word with internal capitals was considered a "WikiWord" and instantly became a link to a page with that WikiWord as the title. Such a page was created automatically if it did not already exist. This was an extremely simple implementation of a linking algorithm based on annotation rather than the naming of resources.

A wiki is a type of simple content management system which allows people to create and edit pages directly from a web browser. A wiki, essentially, is a CMS which partitions and distributes the problem of web content management out to individual contributors, allowing anyone to edit and improve a site. Wikipedia is by far the largest and most well known Wiki. Wikis are a significant example of a bottom-up information architecture. Anyone can add a page and that page is integrated into the overall collection by Wiki word style linking and by including itself in categories (conventionally, by naming them on the page).

Cunningham described WikiWikiWeb as "The simplest online database that could possibly work."<sup>2</sup> Like Markdown, its success has led to additional features, fragmentation, and growing complexity. Some commercial wikis are now complex content management systems. Indeed, it is somewhat difficult today to define the boundaries between Wikis, Blog platforms, and conventional CMSs.

If Wikis have a defining characteristic today it is probably the bottom-up architecture rather than the original novelty of in-browser editing which is now found across many different kinds of CMS. Cunningham designed Wikis to be collaborative platforms – places where people could collaborate with people they did not even know to create

---

<sup>1</sup><https://en.wikipedia.org/wiki/WikiWikiWeb>

<sup>2</sup><http://www.wiki.org/wiki.cgi?WhatIsWiki>

something new without the requirement for central direction or control. The idea was not only architecturally bottom-up but editorially bottom-up. Most Wiki products today, however, include features for exercising a degree of central control. Question and answer sites like Stack Exchange with their distributed and democratic control systems may be closer today to Cunningham's idea of a democratic creation space.

What Wikis illustrate for structured writing is that very simple markup innovations like the WikiWord can have revolutionary effects on how content is created and organized. Most Wikis today use words between double square brackets for WikiWords, rather than internal capitals, but the principle is the same. You can link to a thing merely by naming it.

Wiki words are also a case of subject domain annotation. Marking a phrase as a WikiWord says, "this is a significant subject". It does not provide type information like the subject domain annotation examples shown in this book, but merely denoting a phrase as significant says that it names some subject of importance that deserves a page of its own. This illustrates the point about bottom-up information architectures, that structured writing, even in very simple form, can create texts that are capable of self-organization, that can be assembled into meaningful collections without the imposition of any external structures. However, the wiki process leaves much of the complexity of content creation and management unhandled. The slack has to be taken up by human effort, which has worked well in the case of Wikipedia's army of volunteer contributors and editor, but is harder to reproduce on a corporate scale.

## reStructuredText

reStructuredText is a lightweight hybrid markup language most often associated with the Sphinx documentation framework which was developed for documenting the Python programming language. We looked at reStructuredText briefly as an example of a hybrid markup language in Chapter 37, *Markup*.

Similar to MarkDown, reStructuredText uses a plain text formatting approach to basic text structures. This part of the markup looks very natural because it uses characters and patterns that you might easily use to format a document if the only editor you had available were a plain text editor:

```
Hard Boiled Eggs
=====
A hard boiled egg is simple and nutritious.
Prep time, 15 minutes. Serves 6.

Ingredients

===== =====
Item Quantity
```

```
===== =====
eggs 12
water 2qt
===== =====

Preparation

1. Place eggs in pan and cover with water.
2. Bring water to a boil.
3. Remove from heat and cover for 12 minutes.
4. Place eggs in cold water to stop cooking.
5. Peel and serve.
```

But reStructuredText also has a feature called directives which is used to create markup with more complex semantics. Here, for example, is a directive for inserting an image:

```
.. image:: images/harcboiledegg.png
:height: 100
:width: 200
:scale: 50
:alt: A hard boiled egg.
```

For this, reStructuredText takes the same approach as XML, using characters in a way that they are almost never used in a normal document. This approach simplifies parsing, because there is seldom any question about whether a particular pattern is intended to be markup or text, but it also makes it less natural to read and to write. reStructuredText is therefore something of a syntactic hybrid as well as being a hybrid in the sense that it has both fixed concrete parts and extensible parts.

If you are looking for a lightweight document domain markup language of moderate complexity and a degree of extensibility, or if you are interested in Sphinx as an authoring and publishing system, reStructuredText is an option to consider.

## ASCIIDoc

ASCIIDoc is a lightweight markup language based on the structure of DocBook. It is intended for the same sort of document types for which you might choose DocBook, but allows you to use a lightweight syntax. In appearance it is very similar to MarkDown, as shown in this example from Wikipedia:

```
= My Article
J. Smith
```

<http://wikipedia.org>[Wikipedia] is an on-line encyclopaedia, available in English and many other languages.

== Software

You can install 'package-name' using the +gem+ command:

```
gem install package-name
```

== Hardware

Metals commonly used include:

- \* copper
- \* tin
- \* lead

However, while MarkDown was designed for simple Web pages, ASCIIDoc was designed for complex publishing projects with support for a much wider array of document domain structures such as tables, definition lists, and tables of contents.

If you are looking for a lightweight document domain markup language of medium complexity that is compatible with DocBook (meaning you are interested in creating books rather than web pages) then ASCIIDoc is something to consider.

## LaTeX

LaTeX is a document domain markup language used extensively in academia and scientific publishing. It is not based on XML syntax but on the syntax of TeX, a typesetting system developed by Donald Knuth in 1978.<sup>3</sup> Here is an example of LaTeX, from Wikipedia:

```
\documentclass[12pt]{article}
\usepackage{amsmath}
\title{\LaTeX}
\date{}
\begin{document}
\maketitle
\LaTeX{} is a document preparation system for
the \TeX{} typesetting program. It offers
```

---

<sup>3</sup><https://en.wikipedia.org/wiki/LaTeX>

programmable desktop publishing features and extensive facilities for automating most aspects of typesetting and desktop publishing, including numbering and cross-referencing, tables and figures, page layout, bibliographies, and much more. \LaTeX{} was originally written in 1984 by Leslie Lamport and has become the dominant method for using \TeX; few people write in plain \TeX{} anymore. The current version is \LaTeXe.

```
% This is a comment, not shown in final output.
% The following shows typesetting power of LaTeX:
\begin{align}
E_0 &= mc^2 \\
E &= \frac{mc^2}{\sqrt{1-\frac{v^2}{c^2}}}
\end{align}
\end{document}
```

Here is how that markup is rendered:<sup>4</sup>

It is the markup for the equation that shows why LaTeX is popular for academic and scientific publishing. While not exactly transparent, the markup is compact and functionally lucid for anyone with a little experience with it.

Wikipedia offers a comparison of various math markup formats which shows how big a difference syntax can make to the lucidity of markup language in some cases.

For the equation:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The LaTeX markup is:

```
x=\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
```

Whereas the XML-based MathML version looks like this:

```
<math mode="display" xmlns="http://www.w3.org/1998/Math/MathML">
<semantics>
```

---

<sup>4</sup>By The original uploader was Bakkedal at English Wikipedia - Own work, CC BY-SA 2.5, <https://commons.wikimedia.org/w/index.php?curid=30044147>

```
<mrow>
 <mi>x</mi>
 <mo>=</mo>
 <mfrac>
 <mrow>
 <mo form="prefix">−!-- - --></mo>
 <mi>b</mi>
 <mo>±!-- ± --></mo>
 <msqrt>
 <msup>
 <mi>b</mi>
 <mn>2</mn>
 </msup>
 <mo>−!-- - --></mo>
 <mn>4</mn>
 <mo>⁢!-- ⁢ --></mo>
 <mi>a</mi>
 <mo>⁢!-- ⁢ --></mo>
 <mi>c</mi>
 </msqrt>
 </mrow>
 <mrow>
 <mn>2</mn>
 <mo>⁢!-- ⁢ --></mo>
 <mi>a</mi>
 </mrow>
 </mfrac>
</mrow>
</semantics>
</math>
```

Clearly MathML was not designed with the idea that anyone would ever try to write it raw. It is intended to be the output of a graphical equation editor.<sup>5</sup> You might well choose to use a graphical equation editor to create LaTeX math markup as well, but it is certainly possible to write it and read it in raw LaTeX.

LaTeX is not as lightweight a language as Markdown. Its markup is almost entirely explicit (except for paragraphs, which are delineated by blank lines just in Markdown). But it is certainly lighter weight in its syntax compared to XML-based languages and has much greater functional lucidity. Sufficient functional lucidity to be able to write in raw markup rather than needing a graphical editor is the hallmark of a lightweight markup language. But LaTeX is barely out of the media domain in its structures, which limits its usefulness for structured writing.

---

<sup>5</sup>Interestingly, MathML comes in two different flavors. Presentation MathML is a media domain language describing how an equation is presented. Content MathML is a subject domain language describing what it means.

# Subject Domain Languages

So far we have looked at languages that are primarily document domain oriented. The document domain is an obvious choice for a public language since the use of common document types like books and articles is widespread. But there are a number of public subject domain languages as well. One example that we have looked at before (Chapter 20, *Merge*) is JavaDoc. Here is the example we looked at there:

```
/**
 * Validates a chess move.
 *
 * Use {@link #doMove(int theFromFile, int theFromRank,
 * int theToFile, int theToRank)} to move a piece.
 *
 * @param theFromFile file from which a piece is being moved
 * @param theFromRank rank from which a piece is being moved
 * @param theToFile file to which a piece is being moved
 * @param theToRank rank to which a piece is being moved
 * @return true if the move is valid, otherwise false
 */
boolean isValidMove(int theFromFile, int theFromRank,
int theToFile, int theToRank) {
 // ...body
}
```

Not only does JavaDoc have subject domain tags for parameters and return values, it effectively incorporates the Java code itself (all computer programs are a kind of structured text). Thus the JavaDoc processor will pull information from the function header itself to incorporate into the output.

There are a number of similar languages for documenting different programming languages, such as Doxygen which is used for multiple languages. Wikipedia maintains an extensive list at: [https://en.wikipedia.org/wiki/Comparison\\_of\\_documentation\\_generators](https://en.wikipedia.org/wiki/Comparison_of_documentation_generators).

It is difficult to find public subject-domain lightweight markup languages outside the realm of programming language and API documentation. This is probably because only programmers are likely to write their own parser in order to create a markup language. Most other people are going to choose an extensible language as a base, which today usually means XML. Part of my motivation for creating SAM is to provide a way to create subject domain languages with lightweight syntax.

---

# Chapter 40. Heavyweight markup languages

I am using the term “heavyweight” here as an obvious contrast to the commonly used “lightweight”, even though the term “heavyweight” is not used commonly of markup languages. Nonetheless, it fits. Both the abstract language XML and the concrete languages like DocBook and DITA are heavyweights in the sense that they have a lot of capability that comes at the expense of a large footprint.

Having said that, I should make the distinction between the heavyweight syntax of XML and the heavyweight semantics of a DITA or DocBook. It would certainly be possible to create the semantics of DITA or DocBook in a more lightweight syntax. And it is certainly possible to create very simple markup languages (with semantics much more lightweight than something like reStructuredtext, for instance) using XML. Despite this, there is a definite connection between heavyweight syntax and heavyweight semantics, perhaps because the more heavyweight languages have more need of the capabilities of a fully abstract syntax of XML and the processing tools that go with it.

I’m going to briefly survey some of the heavyweight languages. One thing to note about heavyweight languages is that they often contain structures from more than one domain. Their core is usually in the document domain, but they typically contain some media domain structures for things like tables that are hard to abstract from the media domain in a generic way. They also typically contain some subject domain structures, typically related to technology, since many heavyweight languages originated for documenting technical products. Finally, most contain some management domain structures, particularly for things like conditional text.

Why is the structured writing landscape dominated by a few very large and quite loosely constrained markup languages?

- Partly because big loosely constrained document domain markup languages are the smallest step into the document domain from the word processors and desktop publishing applications that most writers have used for years. They thus represent a relatively small conceptual change for writers. This also means that they do relatively little to partition and redirect complexity away from the writer. They exchange some formatting complexity for structural complexity, which may or may not be an overall win, but they do nothing to enhance rhetorical repeatability or drive information architecture, for instance.
- Partly because a lot of the adoption of structured writing is not motivated by a desire to lessen the amount of complexity that goes unhandled in the content system (with the aim of improving content quality) but by a desire to improve some aspect of content management, particularly content reuse. While it is possible to do these things without resorting to structured writing for a format, structured writing formats ease

the integration of the various parts. They also ease fears about having your content locked into the system of a single vendor.

- Partly it is because constraints are onerous if you don't get them right, and the benefits of getting them right are often under-appreciated, especially in content management applications where the consequences of a lack of constraints tends to show up years down the road (and it is all too easy to blame the problems that emerge on human failure rather than poor system design).

For all these reasons, it is worthwhile to look at where the big public languages fit in the structured writing picture.

For large systems like DocBook, DITA, and S1000D, there is not nearly enough space in this book to do them full justice or to fully characterize them in terms of the structured writing domains and algorithms described in this book. This chapter is therefore not to be taken as a buyers guide. Rather, this book as a whole is an attempt to provide a framework for thinking and talking about structured writing that will allow you to understand your requirements independently of any system, and then to evaluate, compare, and contrast systems in more or less neutral terms.

## DITA

There are two ways of looking at DITA. You can look at it as a complete structured writing system which can be used more or less out of the box. (Even packaged applications like Word or FrameMaker are not used completely out of the box for serious content creation: some customization of styles and output format is needed at least, and the same is true of DITA.)

Alternatively, you can look at what its name proclaims it to be: an information typing architecture. The acronym DITA stand for Darwin Information Typing Architecture, with the word “Darwin” representing DITA’s approach to the extensibility of markup: specialization.

With out-of-the-box DITA, you get a fixed set of topic types provided by the DITA specification and implemented in the DITA Open Toolkit and other tools. With DITA as an information typing architecture, you get the capability to create a unbounded number of information types. I will discuss DITA as an information typing architecture in Chapter 41, *Extensible and Constrainable Languages*. Here I will look at out-of-the-box DITA.

Out-of-the-box DITA comes in two main forms.

1. The DITA Open Toolkit. You can download the DITA Open Toolkit for free and use it to produce content. The formatting stylesheets that come with the toolkit are very basic, so you will likely want to do some customization of the output as a minimum.

2. Packaged DITA tools. There are a variety of tools that package DITA. Most of these are essentially content management systems of one degree of sophistication or another. These may add additional capabilities over what is supplied by the DITA Open Toolkit and may hide the underlying DITA structures to one extent or another. I don't intend to say anything about any of these tools here. The package they offer may differ from the DITA Open Toolkit and should be evaluated on its own merits.

The key features of out-of-the-box DITA that will determine how well it fits with your needs are its topic model and its focus on the reuse algorithm. The description of the document-domain/management-domain approach to reuse in Chapter 12, *Reuse* is based on the DITA model, which provides comprehensive support in those domains.

As described in Chapter 32, *Blocks, fragments, paragraphs, and phrases*, the DITA topic model is based on the concept of information typing, which is the idea that information can be usefully broken down into different abstract types, and that there is value in clearly separating the different types. (This is an approach to partitioning the complexity of information design.) One of the problems with this theory, and consequently with the application of DITA's topic model, is that it is not clear how big an information type is, and whether an information type constitutes a rhetorical block or just a semantic block. Specializing DITA may allow you to be more specific on this point, but if you are using out-of-the-box DITA you are probably using the basic concept, task, and reference topic types (though out-of-the-box DITA now includes a number of other topic types such as Machine industry task and Troubleshooting). There is no clear answer to whether these topic types constitute semantic blocks or rhetorical blocks. In practice they are used both ways.

The principal thing that sets out-of-the-box DITA apart from other approaches to structured writing is its map and topic architecture. In most other systems, the unit that the writer writes and the unit that the reader reads are the same. For very long works, there may be a mechanism for breaking up and assembling pieces. For instance, in DocBook, you can write a book using a book document type in which you can include various chapter document types to create a complete book out of multiple files. In this model, a chapter is a rhetorical block, though it may also be an element of a larger rhetorical block which is the whole book. But DITA partitions this document assembly function into a separate file type, the map. In a DocBook book document, there is a lot of book content in addition to the included chapters. Indeed, you could write the entire book in one book file if you wanted to. The content model of a DocBook document is described by a single schema and the content model of the chapters is simply part of the content model of the book. In other words, a DocBook book is a single document structure that just happens to be made up of individual files. A DITA map file, on the other hand, is an independent structure. It does not create a single logical document structure. It does not usually contain any actual content, and you can't write an entire book in a single map file. Instead, a map file is an instruction to a publishing tool chain about how to assemble a larger work out of component pieces.

This distinction is very important. In the DocBook model, there is a continuity of constraint between the book and its chapters. In DITA, the constraints on the map and the constraints on the topics in the map, are completely separate. This means that in DITA, the topic is the largest unit of content to which content constraints can be applied (at least in the conventional way). This represents a partitioning of the conformance, quality, and composition algorithms with leaves responsibility for larger units entirely with the writer.

Maps are structured like a tree so they can construct hierarchies an arbitrary number of layers deep. This means you have a choice about what parts of your structure you create using a map and what part you create inside a topic. If you have a list of four items, each of which needs two or three paragraphs of description, you can create one topic with the list of four items in it or you can create one topic for each item and then tie them together using a map. This is particularly important when we remember that the topic is the largest unit of content constraint in DITA. If we break the content down to this fine a level, we lose the ability to apply constraints to it. On the other hand, we are now able to reuse each of these items separately.

This presents something of a dilemma. We have already talked about structure writing as dividing content into blocks and made a distinction between semantic blocks and rhetorical blocks (Chapter 32, *Blocks, fragments, paragraphs, and phrases*). In the design of a markup language, rhetorical blocks are made up of semantic blocks which may be made up of smaller semantic blocks or structural blocks. This works fine for developing the structure of a rhetorical block, which is the work that will be presented whole to the reader. In that scenario, the unit that the writer writes is the rhetorical block. The semantic blocks are just elements of the model.

But things become more difficult when you attempt to do fine-grained reuse of content. Then you may want to write individual semantic blocks and combine them to produce rhetorical blocks. DITA will let you do this one of two ways. The first (which is frequently discouraged) is to nest one topic inside another. The second is to combine topics using a map, with the map representing the rhetorical block. However, DITA does not provide a high-level way to constrain the structure of a rhetorical block that is built this way.

If you want a constrained rhetorical block, you have to model it as a single DITA topic type. You can certainly do this by specializing from the base `topic` topic type, but in doing so you will probably move away from the “information typing” idea of keeping different types of information separate, as a full rhetorical topic often requires different types of information (as in the recipe example we have used so frequently).

This leads to the confusion about whether a DITA topic is a semantic block or a rhetorical block. For people who use out-of-the-box DITA this can be a problem because the default Web presentation of DITA places each topic on a separate page, which is not an appropriate presentation if your individual DITA topics are not rhetorical blocks. If your rhetorical block is made up of multiple DITA topics strung together by a map, and you want that rhetorical block to appear on a single page, you need to use a procedure called

chunking, which is not as straightforward as it should be. (Chunking is one of the things on the agenda to be fixed in DITA 2.0.<sup>1</sup>)

The idea that blocks are reusable is a very attractive one. But it is important to think through exactly what the reusable unit of content is. It is one thing to reuse rhetorical blocks whole (perhaps with some variations in the text). It is quite a different thing to reuse semantic blocks below the level of the rhetorical block, particularly if it is important to constrain the rhetorical block or to apply any of the other structured writing algorithms at the level of the rhetorical block.

Quality should be a serious concern with this model. If rhetorical blocks are being assembled out of smaller reusable units without proper attention to the rhetorical integrity or completeness and consistency of the result, quality can be seriously affected. If the author no longer sees, thinks, or works in the context of the rhetorical block, and if the structure of the rhetorical block is not constrained, content quality is very difficult to maintain. Complexity is being dumped on the reader because the cure for the reuse problem did not consider where the rhetorical complexity of composition would go.

DITA, as a technology, does not prevent you from working in whole rhetorical blocks, or from constraining your blocks in any way you want (using its information typing capabilities). But the block and map model (whether implemented by DITA or any other system) presents this inherent tension between creating smaller semantic blocks to optimize for reuse vs creating constrainable rhetorical blocks to optimize for content quality. And because it provides no facilities for imposing constraints above the topic level, it makes it difficult to partition and redirect this complexity away from the writer.

But merely doing reuse of content blocks does not require either kind of constraint. The constraints may improve quality and reliability of the system if used correctly and consistently, but the actual act of composing larger blocks out of smaller blocks does not require them. This has led many organizations to use DITA for its reuse capabilities without paying any particular attention to its constraint capabilities or its information typing roots. People taking this approach will sometime write their content in the base `topic` topic type rather than a more constrained specialization. Quite simply, they are not finding that the partitioning of information design provided by DITA's information typing to be useful and so are ignoring it. Of course, this means that they are laying the responsibility for composability on the writers, but this does not make them any worse off if DITA's information typing was not helping the writers, or DITA's approach did not fit the material.

The growing popularity of this approach to reuse has led to the development of alternatives to DITA that provide the same reuse-management capabilities but remove the constraint mechanisms. One example of this trend is Paligo, a reuse-focused component content management system that uses DocBook as its underlying content

---

<sup>1</sup><http://docs.oasis-open.org/dita/dita-1.3-why-three-editions/v1.0/cn01/dita-1.3-why-three-editions-v1.0-cn01.html#future-of-dita>

format, specifically for the purpose of minimizing constraints on the content.<sup>2</sup> Such systems can reduce the up-front complexity of component-based content-reuse, though possibly at the expense of costs down the road due the failure to apply constraints up front. In other words, more unhandled complexity falling through to readers.

## DocBook

DocBook is an extensive, largely document-domain language with a long history and an extensive body of processing tools and support. As we have noted, DocBook is not a tightly constrained language. Instead it is focused on providing very broad capability for describing document structures. In other words, it is mainly concerned with partitioning the document domain from the media domain in as comprehensive manner as possible. It makes no attempt to constrain the rhetoric of content in any way, and therefore makes no attempt to partition and distribute any part of the creative aspects of the content system.

Unlike DITA, DocBook does not ascribe to any information typing theory. It does not have an opinion about how content should be written or organized. It is very much about the structure of books, and leaves it to the author to decide what the rhetorical structure of the text should be. In other words, DocBook makes no attempt to constrain the rhetorical structure of a work, and in fact makes every attempt to avoid constraining it. The downside of this hands-off approach is that DocBook is a complex system, and the writer is forced to deal with this tool complexity without any of the benefit of other aspects of their task being partitioned and directed away.

Of course, this is not to say that the tool complexity of DocBook is any worse than that of similarly complex and capable publishing tools like FrameMaker. It may well be less so. DocBook is, nonetheless, can be quite challenging to learn and use. Because of this, writers often use simplified subsets of DocBook. (Where DITA is sometimes customized by the addition of elements, DocBook is often customized by their subtraction.) However, DocBook remains popular with many for its lack of constraint combined with its rich feature set.

Because of its lack of constraint, DocBook is not a particularly great fit with the idea of structured writing as a means to partition and redirect the complexity of the content system. However, it can play a very useful role in a structured writing tools chain as a language for the presentation algorithm. This is exactly how it is used in the production of this book. The book is written in SAM in a small, constrained language developed just for the purpose, which is then transformed by the presentation algorithm into DocBook, which then feeds the publisher's standard publishing tools. The DocBook created by this method matches the publisher's exact specifications as required to make the tools work correctly.

In using this approach I was partitioning the complexity of meeting the Publisher's DocBook requirements from my task of writing the book and providing myself with

---

<sup>2</sup><http://idratherbewriting.com/2016/08/01/paligo-the-story-xml-ccms-in-the-cloud/>

a much simpler authoring interface (both syntactically and semantically simpler). Of course, I was transferring that complexity to myself in another role (information architect or content engineer), since I had to write the algorithm that transformed my SAM source from my simple book authoring language to the publisher's preferred form of DocBook. But this was a big win for me, because writing that algorithm was a separate activity that did not impinge on my attention when I was engaged in the writing task. Another benefit was that because my little book writing language is highly constrained, conformance was much easier, and that meant that the DocBook produced by my algorithm was much more consistent and conformed much better to the publisher's requirements than if I had written in DocBook directly.

I wrote my previous book (*Every Page is Page One*) directly in DocBook (an experience that contributed to my decision to develop SAM) but it took a lot of revision to get the DocBook I wrote into the form that the publication process required. In other words, the publishing process has a set of constraints that are not enforced by DocBook itself, and have to be imposed by human oversight and editing when an author writes in DocBook. But in my highly constrained SAM-based markup language, all those constraints were factored out, which enabled me to translate it reliably into the DocBook that the publisher needed. And to work without having to remember any of these constraints.

## S1000D

S1000D is a specification developed in the aviation and defense industries specifically for the complex documentation tasks of those industries, and intended to support the development of the Interactive Electronic Technical Manuals (IETMs) that are typically required in that space. While it obviously has a fair amount of subject domain structures for the target domains, it also has media domain structures targeted at the production of IETMs and extensive management domain structures designed to support the common source database (CSDB), the content management architecture which is part of the S1000D specification. S1000D, in other words, is much more than a structured writing format. It is a specification for a complete document production system for a specific industrial sector. S1000D, in other words, represents a particular partitioning of the content system designed for a particular industry.

## HTML

HTML is widely used as an authoring format for content. For the most part this is a pure media domain usage: people writing for the web in its native format, often using a WYSIWYG HTML editor.

But HTML is still a document domain language, and efforts have been made over the years to factor out the media domain aspects of the language and leave the formatting to CSS stylesheets. This makes HTML a legitimate document domain markup language. In particular, people interested in using HTML this way often use XHTML, the version

of HTML that is a valid instance of XML. Being an instance of XML is important because it means you can write XHTML in an XML editor and process it with XML processing tools. This means that you can potentially publish content written in XHTML by processing it into other formats or by modifying its structure for use in different HTML-based media such as the Web and ebooks.

## Subject domain languages

There are hundreds of public subject domain languages written in XML. Most of these are more data oriented than content oriented, but you might be able to derive content from some of them using the extract and merge algorithms. Wikipedia maintains an extensive list at [https://en.wikipedia.org/wiki/List\\_of\\_markup\\_languages](https://en.wikipedia.org/wiki/List_of_markup_languages). However, most subject domain languages used for content are developed in house to suit a particular subject matter and reader base.

---

# Chapter 41. Extensible and Constraintable Languages

The languages we have looked at to this point are publicly specified and have existing tool chains. Some of them are more constrained than others, and some of them support different structured writing algorithms and different ways of partitioning and redirecting the complexity of the content system. Choosing one of them makes sense if the constraints they express and the algorithms they support are the ones that partition content complexity in a way that is right for your organization. If not, you will need to create your own structures to improve how complexity is partitioned and distributed in your organization.

There are essentially three options for this:

- Create your own language entirely from scratch, creating both the syntax and the semantics. (This is what John Gruber did when he created MarkDown.)
- Use an existing definition of markup syntax, such as XML or SAM, and create your own semantics by defining named structures using that syntax as described in Chapter 37, *Markup*.
- Take an existing markup language with extensible and/or constraintable semantics, such as DITA or DocBook and extend and/or constrain it to meet your needs.

Each of these approaches has its merits and its drawbacks. For instance, creating a new language entirely from scratch may enable it to achieve exceptional functional lucidity for a particular type of information; extending/constraining an existing language can save you a lot of tool development costs; while defining your own semantics based on an existing syntax may let you find the right balance between functional lucidity and development costs.

This chapter will look at the various kinds of extensible and constraintable markup languages.

## XML

The X in XML stands for eXtensible, but, as we noted in Chapter 37, *Markup* XML is an abstract language that does not define any document structures itself. Extension in XML, therefore, is extension from zero. Syntactically, everything is defined for you. Semantically you are starting from scratch.

You can define the structure of a new XML markup language using one of the several available schema languages. We will look at schema languages in Chapter 42, *Constraint Languages*, but the mechanics of defining a markup language in XML are out of scope for this book.

# DITA

DITA is somewhat unique among markup languages in that it was designed for extension from the beginning. In fact, it is something of a misnomer to call DITA a markup language. DITA actually calls itself an information typing architecture. What is an information typing architecture? DITA is really the only thing that calls itself by this name, so to a certain extent we have to derive the definition from the properties of this one example.

The conventional way to define a document type is using a schema language. Schema languages are simply languages for describing constraints on markup structures. (We will look at them in Chapter 42, *Constraint Languages*). So what does an information typing architecture provide over and above what a schema language provides?

There are plenty of precedents for this distinction. The programming world makes use of architectures and frameworks to abstract certain types of operation to a higher level. You could program these functions from scratch, but the architecture or framework is designed to save time and possibly avoid errors. In other words, an architecture partitions and distributes some aspects of a development project away from the local developer to the developer of the architecture and the algorithms the architecture provides. The usefulness of an architecture depends on whether the partitioning and distribution of the problem space that it provides is a good match for your problem space. In other words, does it partition and distribute the complexity of the system in substantially the same way you would have done it you were designing the system from scratch. (Of course, you might not have come up with all the good ideas in a good architecture yourself; the point is that once you have been exposed to those ideas and understood them, would you then partition and distribute complexity according to those ideas or not?) So the question to ask about DITA is, does its architecture partition and distribute content complexity the way that you would naturally wish it to be done to achieve the best outcome for your organization?

In DITA's case, the architecture consists of a set of predefined markup languages that are intended as a basis for extension through a mechanism known as specialization together with an extensive set of management domain markup and the specification of the behavior it should produce, as well as a facility (maps) for assembling information products, and a facility (subject schema) for managing metadata.

In other words, it predefines a range of structures, semantics, and operations that you might need in establishing an information architecture and then provides a way for you to build from there.

As with any other architecture, its usefulness depend on how well the predefined structures, semantics, and operations suit your needs, how easy the extension mechanism is to use, and how reliable the available implementations are.

It is common in the software world for there to be many competing architectures with different sweet spots – different ways of partitioning and distributing the complexity of the development process or the process the tool you are developing it to serve. Because an architecture is essentially a series of guesses about what a variety of systems may have in common, different architectures may be constructed very differently to cover different sets of commonalities among diverse projects and you may not see equivalent architectural features from one architecture to another.

There are not a lot of information typing architectures. The only other one I am aware of is the one I am developing myself, which is called SPFE. SPFE, however, is a very different kind of architecture from DITA. DITA has grown into a very large and complex architecture over the years, and it is out of scope for this book to attempt to describe its architecture in full.<sup>1</sup> All I shall attempt to do here is to do a basic mapping of some key features of the DITA architecture to the structured writing concepts explored in this book.

Inherent in the process of constructing an architecture is that you partition the field in certain ways. Architectures move functionality to a higher level by choosing some options and rejecting others. An XML schema is an information typing language. But a schema can define a markup language for any purpose at all, such as recording transfers between banks or storing the configuration options of an editor. Describing banking transactions or storing configuration options is not within the scope of the information typing that the DITA architecture was designed for. DITA therefore has a more restricted definition of “information typing”. The DITA specification defines “information typing” this way:

Information typing is the practice of identifying types of topics, such as concept, reference, and task, to clearly distinguish between different types of information.

—<http://docs.oasis-open.org/dita/dita/v1.3/csd01/part3-all-inclusive/archSpec/base/information-typing.html>

Unfortunately this definition is largely circular – information typing defines information types. But it does help establish a scale. Information typing is about defining topic types. The spec goes on to define the purpose of information typing:

Information typing is a practice designed to keep documentation focused and modular, thus making it clearer to readers, easier to search and navigate, and more suitable for reuse.

DITA information typing then, is not as general as structured writing. It is focused on information at a particular scale and on a subset of the structured writing algorithms. (That does not mean that it makes it impossible to work at other scales or implement

---

<sup>1</sup>Obviously I am not a fan of the DITA architecture or I would not be developing my own architecture in SPFE. For a much fuller and more sympathetic treatment of the DITA architecture, see *DITA for Practitioners Volume 1: Architecture and Technology* by Elliot Kimber, <http://xmlpress.net/publications/dita/practitioners-1/>

other algorithms, it just means that these are the areas that the architecture supports at a higher level.)

Out-of-the-box DITA is commonly associated with the idea that there are just three information types, task, concept, and reference. The DITA spec makes it clear that this is not the intention of DITA as an information typing architecture.

DITA currently defines a small set of well-established information types that reflects common practices in certain business domains, for example, technical communication and instruction and assessment. However, the set of possible information types is unbounded. Through the mechanism of specialization, new information types can be defined as specializations of the base topic type (`<topic>`) or as refinements of existing topics types, for example, `<concept>`, `<task>`, `<reference>`, or `<learningContent>`.

As I have noted many times, many of the structured writing algorithms partition complexity best with more specific markup, particularly markup in the subject domain. The ability to create an unbounded set of information types is therefore very relevant to getting the most out of structured writing.

Clearly, though, one does not need an information typing architecture to define an information type. You can, as John Gruber did with Markdown, sit down and sketch out a set of structures and a syntax to represent them, and then write a program to process them. With an abstract language like XML or SAM, you can create a new information type by defining a set of named elements and attributes using a schema language. How does using a higher level “information typing architecture” like DITA change this process? How does it partition the design problem differently?

First and foremost, it means that you don’t start from scratch. All topic types in DITA are derived from a base topic type called `topic` by a process called specialization.

What is specialization? We noted that XML is an abstract language, meaning that its syntax defines abstract structures that do not occur in documents: elements, attributes, etc. To create a markup language in XML, you define named elements and attributes for the structures you are creating. Thus in DocBook `para` is a type of element. `para` has what is called an “is-a” relationship to elements. This is a type of specialization. `para` “is-an” element, but it is a special type of element. An XML parser will process it generically as an element, reporting its name to the application layer. The application layer must then supply a rule that processes just this specialized `para` element (and not the also specialized but different `title` element).

DITA specialization follows the same principle, but moves it up a level. The base `topic` topic type is the abstract structure. More specific types like `knitting-pattern` or `ingredients-list` are specializations of `topic` (or of other topic types that are specializations of `topic`). A generic DITA processor can process them as a instance of `topic`, but it would require additional code to process them specifically as `knitting-`

pattern or ingredient-list. Each of these specialized types has an “is-a” relationship with the type it was specialized from. So knitting-pattern “is-a” topic.

But DITA specialization is different from simply creating new named elements in XML in a number of ways.

First, the base DITA topic type is not an abstraction like an XML element. You cannot create an XML element without inventing a name for it. The base DITA topic type, on the other hand, is a fully implemented topic type that you can use directly. You can, and people do, write directly in the base topic type without inventing anything new. We noted in Chapter 30, *Rhetorical Structure* that it is sometimes easy to treat what is intended as a meta model as a generic model. This is the case here. All topic types in DITA are derived by specialization from the generic `topic` type. They all have an “is-a” relationship to this generic type.

One consequence of this is that while the set of topic types you can create with DITA may be unbounded, it is not universal. The generic topic type has specific characteristics and if a specialized topic has an is-a relationship to the generic topic, the structures in the specialized topics have a corresponding is-a relationship to structures in the generic type. Thus there can be information types that cannot reasonably be said to have an is-a relationship to a DITA generic topic. That is, there can be information types such that processing them using the code of the type they are specialized from would produce no meaningful result. For example, an information type that factored out most of the text that would appear in the published version would not process meaningfully as a generic topic because the factored-out text would not be restored.

To specialize a topic type, you specialize the root element and any child elements or attributes that you need to define your new topic type. Each specialized element or attribute should have an is-a relationship to the element it specializes. Thus a procedure element might be a specialization of an ordered list element and its step elements might be specializations of a list item element. (You can see that in this case, processing a procedure as an ordered list would produce meaningful output, but that you might also want to specialize the output of steps in a procedure, perhaps by prefixing each step with “Step 1:” rather than just “1.”.)

The second way in which specialization differs from giving names to abstract elements is that specialization is recursive. That is, suppose you have a topic type `animal-description`, which is a specialization of `topic`. You want to impose additional constraints on the description of different types of animal, so you create more specialized types `fish-description` and `mammal-description` which are specializations of `animal-description` (and would be processed like an `animal-description` if no other processing were specified for them). Then you might decide to impose still more constraints on the description of different kinds of mammals, so you create a type, `horse-description` that is a specialization of `mammal-description`. This type will be processed as a `mammal-description`.

if no specific processing is provided for horse-description; as animal-description if no specific mammal-description processing is provided; and as topic if no specific animal-description processing is provided.

The third way in which information typing in DITA differs from doing it from scratch is that DITA information types share a common approach to processing and to information architecture. In particular, they inherit a common set of management domain structures and their associated management semantics. It is possible to ignore all of these things, but if you do so, the value of using DITA for information typing is reduced because you then have to invent your own ways of doing these things that are contrary to the partitioning provided by the architecture.

As a generality, the fewer pieces of an architecture you use, the less value there is to basing your work on that architecture, both because you have more work to do, and because you take less advantage of the infrastructure or tools and expertise surrounding that architecture, and create a system that is less understandable to people versed in the architecture. All this betrays a poor fit between the system partitioning you are creating and the partitioning that the architecture provides. All architectures come with overheads and if you don't use their features, you still have to live with their overheads, which adds cost and complexity to your system. Thus while you can use DITA and depart from the default DITA way of doing things, the value of using DITA diminishes the further you depart from the DITA way. The same would be true of any other information typing architecture.

## Specializing between domains

We have talked a lot about the benefits of moving content from the document domain to the subject domain as a means to factor out constraints in the document domain and as a way to enable multiple structured writing algorithms and improve functional lucidity.

DITA's base types clearly belong to the document domain. Clearly you can use specialization to create more specialized document domain structures. But can you use specialization to create subject-domain structures that factor out aspects of the document domain?

In formal terms the answer is no. Subject domain information does not have an is-a relationship to document domain information precisely because it is the document domain structures that you factor out when you move to the subject domain.

Take the list of ingredients in a recipe. In the document domain, they could be presented as a list or as a table. In subject domain terms they are actually more of a table (database sense) than a list. That is, a set of records with a defined semantic structure:

```
ingredients:: ingredient, quantity, unit
eggs, 3, each
salt, 1, tsp
butter, .5, cup
```

How do you create this record structure by specializing document domain elements? What is the best starting point to specialize from? A table is the most obvious candidate because it is structured like a set of records. However, the more common presentation of an ingredient list is as a list. But there is no structure in a list that supports dividing a list item into three named fields. The ingredient structure above is neither a list nor a table; it is a record set, a data structure whose contents could be presented in many different ways in the document domain, but is not a specialization of any of them.

We can make a distinction between two main types of subject domain languages, the rhetorical subject domain, where the focus is on how to present information on a particular subject, and the data-oriented subject domain, where the focus is on capturing the various pieces of information that are required on a subject while factoring out the presentation. Both present problems for specialization.

One of the elementary things that the rhetorical subject domain does is to factor out the titles of various sections of a topic, replacing the generic structure of sections and titles with specific names blocks whose titles can be supplied consistently by the presentation algorithm. These named sections do have a kind of is-a relationship to generic sections, in that they are sections. But this is not a true is-a relationship because a generic section requires a title, and a rhetorical block cannot have one. It therefore does not have all the characteristics of the thing it is specialized from, and therefore it is not a section. It is a named block of text that, in the document domain, would be expressed as a section, but that is not the same things as falling back to being a section. Processing such a block as a section would result in the text following on from that of the previous section with no title, thus lacking the defining document and media domain characteristic of a section.

For the data-oriented subject domain, which, let us remember, can include material recorded in databases and source code, there is no necessary relationship between the structure of content in the source and the presentation of content in the document domain. Information that is in separate fields in the subject domain may need to be combined to form sentences by the presentation algorithm before it is even readable as a piece of content. Such structures clearly have no is-a relationship to a DITA generic topic.

More generally, as we have seen, the subject domain changes the algorithm for each function so inheritance of the code is of little or no value. Subject domain algorithms work differently. And since all subject domain algorithms work on the same structures, rather than there being separate structures for each algorithm, as in the document and management domains, there is little scope for inheritance of processing code from the document domain to the subject domain. Rather, the document domain is semantically downstream from the subject domain. Subject domain algorithms essentially create

document domain structures as part of the publication process, as we saw in Chapter 27, *Publishing*.

## DocBook

DocBook is not really extensible in the same sense as the other languages mentioned here, but it still deserves a mention. DocBook does not provide an extension mechanism like DITA's specialization. What it does provide is a deliberately modular construction that makes it easy to create new schemas that include elements from DocBook. DocBook takes full advantage of the extensibility features built into XML schema languages.

Does the fact that DocBook does not invent its own extension mechanism mean that it is not as extensible as DITA? No. By relying on XML's own extensibility features, which are both more comprehensive and lower level than DITA's specialization mechanism, DocBook is as extensible as it is possible for any XML vocabulary to be.

Where it differs from DITA is that there is no fall-back processing. Extensions DocBook are not DocBook. They are new languages that incorporate DocBook structures. The extensions cannot be processed by standard DocBook tool chains, though the incorporated DocBook structures obviously can. DITA's specialization mechanism means that a specialized topic will always pass through the DITA publication process, though whether it will be presented in a useful or comprehensible way very much depends on how well the is-a relationship between specialization and base was maintained. If you would rather ensure that topics always pass through the publication process, even if the results are gibberish, DITA will support that. If you want to ensure that errors are raised if any structure is not recognized by the publishing tool chain (thus avoiding accidental gibberish) then DocBook's extension mechanism will give you that.

Another aspect of DocBook customization deserves to be mentioned here even though it is not strictly speaking extension. DocBook has a huge tag set and it is quite conceivable that if you want a small constrained document domain markup languages that you can create one by sub-setting DocBook. DocBook provides for just about every document structure out there, so if you are building a document domain language, chances are the pieces you need are in there.

The great advantage of creating a new language as a subset of DocBook is that the result is also a valid DocBook document and can therefore be published by the DocBook tool chain. You will not have to write any algorithms at all if you take this approach. Creating a subset of DocBook can therefore allow you to impose more constraints and improve functional lucidity significantly compared to standard DocBook without having to write any processing code at all.

Technically speaking, any XML-based markup language is extensible in the same way that DocBook is. However, DocBook's structure, and the implementation of its schemas, was designed deliberately to support both extension and sub-setting of DocBook, something which is not true for many markup languages.

# RestructuredText

RestructuredText has a number of blocks for things like paragraphs, titles, and lists that are defined with a concrete syntax. It defines other blocks using directives:

```
.. image:: images/biohazard.png
:height: 100
:width: 200
:scale: 50
:alt: alternate text
```

It is extensible by adding new directives to the language. However, there is no schema language for RestructuredText. To create a new directive, you have to create the code that processes it.

There is an important distinction to be made between languages that are extensible by schema and those that are extensible by writing code to process the extension. If a language is extended by writing processing code for the extension, the only way to know if the input is valid is by processing it. If it raises a processing error, it is invalid.

If you have only one processor for a language, you can treat that processor as normative. That is, the definition of a correct file is any file that can be successfully processed by the normative processor. The language, in other words, is defined by the processor. But if you have multiple processors, how do you determine who is at fault when one of those processor fails to process a given input file? Is the processor incorrect or the source file?

A schema creates a language definition that is independent of any processor. (In other words, it partitions and redirects the complexity of validation in language design.) It is the schema that is normative, not any of the processors. If the source file is valid per the schema, the processor is at fault if it does not process that file correctly. If the source file is not valid per the schema, the blame lies with the source file.

In the case of RestructuredText, the capacity of the processor to be extended in this way is built into the processor architecture. It is not like you have to hack around in the code to add your extensions. There is a specific and well documented way to do it. But while RestructuredText allows you to extend it by adding new directives, it does not have a constraint mechanism. There is no mechanism (other than by hacking into the code) to restrict the use either of new directives or the existing directives and structures.

# TeX

TeX (pronounced “Tek”) is a typesetting system invented by Donald Knuth in 1978. As a typesetting language it is a concrete media domain language. But Knuth also included a

macro language in TeX which allows users to define new commands in terms of existing commands. (I say commands because that is the term used in TeX. Markup in the media domain tends to be much more imperative than markup in the subject domain, which is entirely descriptive, so “commands” is an appropriate name for TeX’s tags.) This macro language has been used to extend TeX, most notably in the form of LaTeX, a document-domain language that we looked at in Chapter 40, *Heavyweight markup languages*.

As we noted with RestructuredText, extension of a language is not the same thing as constraint. Introducing new commands does not create a constraint mechanism.

## SAM

While lightweight languages provide great functional lucidity, they suffer from limited extensibility (which generally requires writing code) and a general lack of constraint mechanisms. I believe that a fully extensible, fully constrainable lightweight markup language would be a valuable addition to the structured writing toolkit. This is why I have developed SAM, the markup language used for most of the examples in this book and for writing the book itself.

As described in Chapter 37, *Markup*, SAM is a hybrid markup language which combines implicit syntax similar to MarkDown with an explicit syntax for defining abstract structures called blocks, recordsets, and annotations, and with specific concrete markup for common features such as insertions, citations, and variable definitions.

SAM, like XML, is for defining specific markup languages. However, all languages defined in SAM share a small common base set of text structures for which SAM provides concrete syntax. This allows SAM to combine lightweight syntax for the most common text structures with the ability to define specific constrained markup languages for particular purposes, particularly subject domain languages. In other words, SAM represents a different partitioning of the markup design process from both the common lightweight languages and from XML.

SAM is designed to be extensible and constrainable through a schema language (this is not complete at time of writing, but hopefully will be available by the time you read this). The intent is that the schema language should be able not only to define and constrain new block structures, but to constrain the use of the concrete structures as well, and to constrain the values of fields using patterns.

SAM is not designed to be nearly as general as XML in its applications. As a result, its syntax is simple and more functionally lucid and its schema language should also be simpler and make it much easier for writers to develop their own SAM-based markup languages.

I use SAM for the majority of the examples in this book because SAM is designed to make structure clear and that is all I have needed to do in most examples. All the examples

could be expressed in XML as well. Using XML would just have made them harder to follow. Naturally, to write in SAM you would need to know more about the rules of the language, but you should be able to read a typical SAM document and understand its structure with little or no instruction.

This is similar, but not identical, to the aim of mainstream concrete and hybrid languages such as Markdown and Restructured Text, which is to have the source file be readable as a document. In other words, they strive to make the document structure clear from the markup. They are document domain languages, and they strive to make sure that the markup expresses the document structure they create in a way that is readable. SAM has the same goal, except that SAM was designed primarily for creating subject domain languages. As such, it is designed to make the subject domain structure of the document clear to the reader.

A SAM document may not look as much like a finished document as a Markdown or reStructuredText document. For example, it does not use underlines to visually denote different levels of header. Instead, it focuses on creating a hierarchy of named blocks and fields. In doing so, it uses the kind of markup people commonly use to create named blocks of text and to express a hierarchical relationship between them. Blocks are introduced with a name followed by a colon, and hierarchy is expressed through indentation.

examples: Basic SAM structures

example: Paragraphs

The is a sample paragraph. It is inside the {block}(structure) called `example`. It contains two {annotations}(structure), including this one. It ends with a blank line.

This is another paragraph.

example: Lists

Then there is a list:

1. First item.
2. Second item.
3. Third item.

example: Block quote

Next is a block quote with a {citation}(structure).

" " " [ Mother Goose ]  
Humpty Dumpty sat on a wall.

SAM is an open source project. A description of the language and a set of associated tools are available from <https://github.com/mbakeranalecta/sam>.

## SPFE

SPFE is another project of mine. It is designed to be a framework for implementing structured writing algorithms and its structure follows the model I laid out in Chapter 27, *Publishing*. It is tempting to compare it to DITA as an information typing architecture, but as I commented before, architectures are not necessarily parallel to each other and often differ in their emphasis. SPFE takes a different approach to the partitioning and distribution of content complexity, with a major emphasis on directing content management and information architecture complexity away from writers. Individual writers working in a SPFE system should have to know little or nothing about how SPFE works, as long as they follow the constraints of the markup language they are using.

SPFE is principally designed for subject domain markup. As such, it does not start with a generic document domain topic type like DITA. SPFE does not require any particular schema, though it does require that schemas meet certain constraints.

But SPFE does not leave it entirely to you to develop schemas from scratch. Instead, it supports building schemas from pre-built components. The pre-built components include a collection of semantic blocks and the default processing code for each stage of the publishing algorithm. SPFE also allows you to define your own reusable structured components with processing code. This is, essentially, extensibility through composition, rather than extensibility through specialization (as in DITA) or extensibility through processor extension (and in reStructuredText). Constraints are supported through normal schema mechanisms and by selecting the minimal required structural components for the individual case.

By strictly segregating the presentation and formatting layers, SPFE reduces the effort required to process custom markup formats. Custom format are processed to a common document-domain markup language which it then processed to all required media-domain output formats. The SPFE Open Tool Kit includes a basic document domain language for this purpose, but you can also use DocBook or DITA in this role, allowing you to take advantage of their existing publishing capabilities. This also allows you to install SPFE as an authoring layer on top of an existing DITA or DocBook tool chain.

To create a subject-domain markup language in SPFE, therefore, all you have to define for yourself are the key subject-domain fields and blocks that are essential to your business. All the other elements you need, such as paragraphs, lists, tables, and common annotations, you can include from the pre-built components, along with their default processing code.

Among its default processing steps, the SPFE process includes the subject-based linking algorithms described in Chapter 15, *Linking* and the subject-based composition and architecture algorithms described in Chapter 13, *Composition* and Chapter 21, *Information Architecture*, including bottom-up information architecture. The conformance and audit algorithms are well-supported as well.

While it has support for reuse, SPFE is not as focused on content reuse or content management as DITA. It deliberately limits some of the forms of reuse that tend to produce unmanageable complexity. While it can produce books and top-down information architectures, its main focus is hypertext and bottom-up information architectures. SPFE does not define or require maps as an assembly mechanisms, though you could implement maps in SPFE if you wanted them. SPFE's processing model is modeled on a software build architecture and it is designed to work well with a version control system system as a repository rather than a content management system. One of its key design objectives is that writers should have to know little or nothing about how SPFE works.

Both SAM and XML are supported as markup syntax for SPFE, and you can freely mix and match SAM and XML content.

SPFE is an open source project available from <http://spfeopentoolkit.org>.

---

# Chapter 42. Constraint Languages

Structured writing is about applying constraints to content and recording the constraints that the content follows, both to constrain what writers write and to constrain how algorithms interpret the content. This requires some way to express constraints in a formal and machine readable way. Schema languages partition this problem and redirect it to a common validation algorithm expressed by a standard piece of software that everyone can use. Schema languages are, quite simply, languages for expressing constraints.

For a concrete markup languages like MarkDown, the content constraints are established in the code of the MarkDown processor. They are validated when the processor parses the MarkDown file. (In practice, though, MarkDown does no meaningful validation. Anything it does not recognize as markup, it simply outputs as text.)

For abstract markup languages like XML you define structures yourself. Basic XML syntax is validated by the parser, but the definition of constraints is the business of a schema language. The validation of those constraints is the business of a piece of software called a “validator”.

A schema language is a structured language for defining structured languages. The schema for a markup languages says what structures are allowed and in what order and relationship. A given document either conforms to those constraints or it does not.

Here is an example of a schema in a schema language called RelaxNG, which is one of several schema languages available for defining XML-based markup languages:<sup>1</sup>

```
<element name="book" xmlns="http://relaxng.org/ns/structure/1.0">
 <oneOrMore>
 <element name="page">
 <text/>
 </element>
 </oneOrMore>
</element>
```

This example defines two elements and three constraints. The first element is called book and the second is called page. The constraints are:

- The page element must occur inside the book element. (Because the page element is defined inside the book element structure.)
- There must be at least one page element inside the book element, and there can be more. (Because the page element is defined inside an RNG oneOrMore element.)

---

<sup>1</sup>[https://en.wikipedia.org/wiki/RELAX\\_NG#XML\\_syntax](https://en.wikipedia.org/wiki/RELAX_NG#XML_syntax)

- Text can occur inside the page element, but not directly inside the book element. (Because the RNG text element occurs inside the definition of the page element, but not as a direct child of the book element definition.)

Thus if an author wrote:

```
<book>Moby Dick
 <page>Call me Ishmael. Some years ago- never mind how long
 precisely- having little or no money in my purse, and nothing
 particular to interest me on shore, I thought I would sail
 about a little and see the watery part of the world.</page>
</book>
```

the validator would report an error because the words “Moby Dick” are directly inside the book element and text is not allowed in that position.

There are several different schema languages for XML each of which is capable of expressing and enforcing different sets of constraints. It is not unusual to combine different schema languages to more completely constrain a markup language. In particular, it is not uncommon to use a schema language called Schematron in concert with other schema languages like RNG or XSD. While most schema languages work by modeling the structure of a document, as in the RNG example above, Schematron works by making assertions about the structure in a language called XPath. A Schematron schema would make a lousy guide for authoring, but it can test assertions, and therefore enforce constraints, that no other schema language can.

Here is a simple Schematron example. It defines one of the constraints listed above, namely that a book element must contain at least one page element.

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
queryBinding="xslt2">
 <pattern>
 <title>Book constraint</title>
 <rule context="book">
 <assert test="page">A book must contain at
 least one page.</assert>
 </rule>
 </pattern>
</schema>
```

The rule says that in the context of the element book the assertion that there is an element page must be true. If it is not, the message within the assert element will be displayed.

In the RNG schema example, the schema is essentially a template that models the hierarchy of the document. The constraints are consequences of the structure of that

hierarchy. (The `book` element cannot contain text because there is no place for text in that part of the hierarchy.) Only documents that conform to the template are valid.

In the Schematron example, the schema is a series of constraint statements. The hierarchy of the document is a consequence of meeting all the constraints. Any document is valid as long as it conforms to all the stated constraints.

There are essentially two ways to describe constraints. One is to start from the basis that nothing is allowed unless there is a specific rule allows it. The other is to start from the basis that everything is allowed unless there is some specific rule that forbids it. Schematron is based on the latter doctrine. It says anything is valid as long as it passes a given set of tests. Other schema languages take the former approach. If you validate an XML document against an empty XSD schema, for instance, the validator will say it is invalid because it can't find the definition of the root element. In essence, therefore, their constraints are described as a set of permissions. The writer is then constrained to stay within the boundaries of what is permitted.

Starting on the basis that everything is allowed is not generally a good model for structured writing. You have to write algorithms that can handle everything that is included in a document, so you pretty much have to start with a definitive list of what is allowed and in what combinations. This is why Schematron is seldom used as a standalone schema language. But it can be very useful for qualifying the use of certain structures permitted by other in other schema languages, in particular the kinds of constraints that cannot be expressed in the main schema language.

The wider point here are that there are many occasions in the course of writing and publishing content in which to express and enforce constraints. Some constraints can only be expressed during the publishing process, for instance. An example would be the resolution of a key reference. We can only tell if there is a value or resource corresponding to the key when the synthesis algorithm is complete and all the elements of the published content set are assembled and resolved.

Constraint, therefore, is not something that exists at a single point in the writing and publishing process. It is something you need to consider across the entire publishing process design. In particular, when errors occur at any point in the publishing process, you should trace them back to the point where the constraint that could have prevented them was violated, and ask how that constraint could have been better implemented or enforced.

One of the most basic design principles of any process is that errors should be caught as early in the process as possible, and that any error-prone process should be designed to make errors less likely. Factoring out constraints by moving content to a different domain, as we have noted several times, is an excellent way to reduce errors in structured writing, and to ensure that errors are caught earlier in the process.

---

## **Part V. Design**

---

## Table of Contents

43. System design .....	367
Identifying complexity .....	368
Current pain points .....	368
Unrealized possibilities .....	369
What's working now .....	370
Complexity that impacts others .....	370
Process overheads .....	371
Your style guide .....	372
Change management issues .....	372
What's coming down the road .....	373
Partition and direct complexity .....	374
Focus on complexity, not effort .....	376
Indivisible complexity .....	377
Subtract current tool complexity .....	377
Avoid tool filters .....	377
Select domains .....	378
Select tools .....	383
Count the costs and savings .....	384

---

# Chapter 43. System design

Unless you are writing directly on paper, chiseling into stone, or drawing letters by hand in a paint program, you are using structured writing techniques to create content. You are using tools that have been designed to partition and distribute some part of the complexity of content creation in some particular way, and you are observing a discipline, a set of constraints, imposed by that partitioning. It is not a matter of whether your system is structured or not, but how and for what purpose it is structured, and whether that purpose is being achieved.

The question you need to address, therefore, is whether your current partitioning is ensuring that all the complexity in your content creation process is being handled by a person or process with the skills, resources, and time required, or is complexity going unhandled in your system and getting dumped on the reader?

If not all of your complexity is being appropriately handled, if complexity is falling through to your readers in the form of quality problems, then the question becomes, how do you change your approach to structured writing to handle complexity better?

It is important not to approach this problem piecemeal. Complexity cannot be destroyed, only redirected, and every tool you adopt introduces new complexity which must also be partitioned and redirected appropriately. Attacking one piece of complexity in isolation usually results in complexity being directed away from the area you attack, but that complexity goes somewhere, and if you don't think about where it goes, and how it will be handled there, you can easily end up with more unhandled complexity in your system than you started with.

The very fact that structured writing has the ability to move the complexity around the system, imposing it on one role at the expense of another, can lead to piecemeal rather than comprehensive solutions as different groups or different interests try to attack their part of the problem. For instance, there has been a long history of IT departments choosing content management systems on the basis that they were easy for the IT department to install and administer, only to have them be deeply unpopular with users because they pushed complexity out to writers and other users. On the other hand, some groups of writers want to write in uber-simple formats like Markdown, despite the difficulties that its limited structure and capabilities create for an overall publishing process. Everyone tries to simplify their own lives, heedless of the expense to others.

Here is the inescapable fact: complexity is irreducible. It can be moved but not destroyed. The only way to remove complexity from a process is to stop doing something. Actually, the first question you should ask in any assessment of your process, is what things you are doing that you should simply stop doing. What is not creating any value? But let's assume that you have done that. Now that you have stopped doing things that you did not need to do, simplifying one function, will always move complexity somewhere else

in the system. Moving complexity somewhere else in the system is fine if it is moved somewhere where it can be handled by an appropriate person or algorithm. The point is not to eliminate complexity, but to make sure it is handled correctly. The enemy is not complexity itself, it is unhandled complexity.

How do you design a content system that ensures that all complexity (or as much of it as is reasonably possible) is handled by a person or algorithm that has the time, skills, and resources to handle it? You start by understanding the sources of complexity in your content system.

## Identifying complexity

Where is the complexity in your content system? It is different for every organization. While the basic functions of content development are the same for everyone, the amount of complexity they generate differs greatly depending on your needs and circumstance. If you translate into 35 languages for example, translation generates more complexity in your content system than in the system of an organization that translates to one language, or to none. If you have a complex product line in an industry with a complex vocabulary, terminology control creates far more complexity in your system than for someone with a simple consumer product. In order to know where you need to partition and redirect complexity, you need to identify where the biggest sources of complexity are in your content system. Here are some places to look:

## Current pain points

Inconsistency, duplication, delay, error, and failure are escape valves for unhandled complexity. The complexity underlying these problems can be hard to see, even when it is causing pain. Familiarity with your current process may make it difficult to see that things could be different. Seeing that there is another possible way to do things often provides the insight into the hidden complexity of the current process. Hopefully the chapters in this book on the structured writing algorithms and the different ways that they are handled will help you see your current processes in a different light, and therefore help you to identify the complexity in them, and assess whether it is being adequately handled.

In order to find the true source of the complexity, however, you may have to work your way upstream from the place where the problem is manifest. Ultimately, all unhandled complexity finds its way to the reader, so quality problems are usually the place that complexity will show up if it is not being handled properly. The other place that you may find the impact of complexity is in processes that seem too expensive or in staff that seem overburdened or burnt out.

Let's say that you observe a quality problem: readers are getting lost in the content. They find an initial page, but they need more information on the concepts it mentions. But

when they search, they end up back at the same page. What is the root cause of this problem? It could be one of (at least) three things:

1. The information on the page is incomplete. The information should be there, but it's not.
2. The content they need is not in the content set anywhere because no one realized it was needed.
3. The content they need is in the content set but they can't get to it because there is no link to it from the page they are on.

If you determine that the first issue is to blame, then the complexity that is going unhandled is that of determining the reader's needs on a particular subject and making sure that it is always present in topics on that subject. You have a repeatability problem.

If the second issue is the problem, then it is the complexity of content planning that is not being handled. You may need to improve your audit capability.

If the third issue is the problem, then it is the complexity of managing and expressing the complex relationships between subjects that is not being handled. Perhaps the authors are not aware of the relationships, in which case you may have a rhetorical structure problem. Or perhaps they are aware but the cost of creating and maintaining links is too high, so they are skimping on linking. In this case you may need a different approach to linking.

Be careful when you do this analysis not to fall into two common pitfalls:

- Don't stop your analysis too soon, and don't be blinded by the limits of your current process. Trace the problem back to the actual source of complexity, independent of any tool or process considerations.
- Don't stop with the first big chunk of complexity you find and rush off to buy a tool to address it. You need to identify all the source of complexity in your system and come up with a comprehensive plan to manage all of them, otherwise you will just move complexity around, while adding tool complexity to the mix. You can easily end up worse off than you were before. Even your biggest hot-button issue may demand a very different solution once you have figured out the total complexity picture in your system and the overall best way to partition and direct that complexity.

## Unrealized possibilities

Chances are that there are things you don't do today because they are not possible with your current tools, or any of the tools you are used to. It is easy to overlook these unrealized possibilities altogether because everyone else is dealing with the same tool limitations, meaning you may not see these possibilities realized elsewhere. But don't let the designs you can conceive be limited by the designs your current tools can execute.

Those limitation just point to a design complexity that you don't currently have the means to handle.

A comprehensive bottom-up linking strategy is a good example of unrealized possibilities. If you are working in the document domain, and using document domain or management domain linking techniques, the then creation, management, and updating of links is expensive and complex to execute. In reaction to that complexity, you minimize the links you create, or create them in sub-optimal places. That complexity manifests itself as unrealized potential.

But if you switch to subject-domain linking based on subject annotations, you partition and distribute the complexity of linking away from authors and towards an algorithm, radically reducing the barriers to a comprehensive linking strategy and allowing the possibility of that design to be realized. (And note that you don't have to go to a full subject domain system to use this approach to linking. It works just as well with subject annotations in otherwise document-domain information types.)

## What's working now

Some parts of your current system are working well, successfully partitioning and directing complexity to the right people or processes. You still need to inventory this complexity to make sure it is still handled well in your new process. (But don't assume that because you are satisfied with this part of your current process that it is actually working optimally – you may just be settling for what you are used to.)

For instance, if you are generating reference information from source code using a tool like Doxygen, that is a complex task that is currently being handled by an algorithm. Make sure you add that to your inventory so that when you design your new system you don't end up with that complexity no longer being handled properly.

At the same time, look around the edges of your current success stories to see if there is any unrealized potential there. Suppose your generated reference is being published through a different tool chain so it looks different and is not linked from or to the rest of your content. This is integration complexity that is going unhandled and falling through to the reader in the form of less usable content.

You may end up completely changing the way you handle the complexity you are successfully handling now in order to better distribute complexity across the whole system. But don't loose sight of the complexity just because you are currently handling it well.

## Complexity that impacts others

More difficult to identify are points where the impact of complexity is falling on someone else. These are the pains you are not feeling because someone else is suffering the

consequences. Often this is the reader, but it could be others in your organization – support, sales, field engineering, etc.

For instance, if you are asking developers to contribute content to your documentation set, but you are working in a complex document domain or media domain tool that the developers don't have the bandwidth to learn, you could be dumping a lot of authoring complexity on them.

You may get push back from the developers saying that if they are going to contribute, they are only willing to do so in MarkDown. This is another case of the deflecting complexity onto someone else, since MarkDown may not give you the structure and metadata you need to integrate the content with your publishing system.

Here you have a functional lucidity problem. The complexity of integrating content into an overall content set is not being sufficiently partitioned and directed away from the occasional contributor. Chances are there is also a rhetorical problem here: developers not knowing what information it actually needed, and not being given any rhetorical guidance. The solution may be to provide them with a specific subject domain format that precisely specifies the content required. The developers may be willing to use and learn this format (if its functional lucidity is good) precisely because it makes their lives easier by redirecting rhetorical complexity away from them.

## Process overheads

Communication overhead is a huge source of complexity. Desktop publishing (DTP) was revolutionary in its day mainly for eliminating communication overhead between writers, designers, and typesetters by combining the three jobs into one. There is a huge benefit to developing interfaces that reduce the need for communication between collaborators. If the writer is thinking visually then giving them the tools to execute their visual ideas removes the need for communicating with a designer and a layout artist.

There are other approaches to handling this communication overhead. The document domain approach is to try to get the writer to think in terms of abstract document structures rather than formatting (separating content from formatting). Tactically, this is the exact opposite of how DTP partitioned things, but strategically it has the exact same objective – to eliminate the communication overhead between writers, designers, and production people, and its attendant costs. The document domain approach further speeds up the process by handing the formatting to an algorithm, but introduces more abstraction into your system and transfers complexity to the person who writes the formatting algorithm.

The DTP approach dumped the formatting and production complexity on the Author, who now became designer and layout artist as well as writer. The document domain approach distributes the design complexity to the coder of the formatting algorithm and the execution of that design to the algorithm itself. No complexity has been removed, just

redirected. And new complexity has been introduced because the author now has to learn the correct document domain structures to use. This complexity can be redirected again by moving to the subject domain, where the author write in a simpler, more concrete format where these choices are easier to make.

Any part of your process that is cumbersome, any part where information is being lost or work is having to be done twice, or where things slow down or bottlenecks occur is a place were complexity is hiding.

## Your style guide

Traditionally, a style guide has been the dumping ground for complexity. Every new rule and requirement created in response to a content problem gets written into an ever growing style guide. This complexity is not actually being handled, it is merely being dumped on the writer. But the style guide quickly grows beyond the capacity of any writer to remember or follow it in every word they write. The rules are not followed and the complexity falls through to the reader.

This is not to say that there is no need for a style guide. There are style issues that cannot be effectively factored out and these the writer will have to deal with, and will need guidance to deal with consistently. But the writer's ability to conform to these stylistic constraints will be directly proportional to their number. Using structured writing techniques to redirect as many style issues as possible away from the author (and thus the style guide) increase the conformance to those issues that cannot be partitioned or directed away.

Taking your style guide and simply going though it looking for rules that you could factor out by moving content to a different domain is a great way to inventory the complexities in your system.

## Change management issues

The growth in unhandled complexity will often be interpreted (especially by the advocates of a system you have just installed) as a change management problem or a training problem. But while change management and training are both necessary any time you change how you do things, the most likely cause of problems after the installation of a new system is design problems, specifically a system design that redirects complexity away from itself without consideration for how it is going to be handled.

Change management is a scapegoat for process complexity that no one will own. It is very common to blame the failure of structured writing systems on unwillingness of writers to change. The cure, the backers of the system assure us, is more training and more generalization of the change. But what is usually going on in these cases is that the system design has dumped new and unmanageable complexity on whatever group

is rebelling – almost always the writers, either the full time writers or the occasional contributors.

## What's coming down the road

The world is in the middle of a transition from paper to hypertext.<sup>1</sup> This involves three types of complexity. First, the complexity of architecting information and publishing to paper. Second, the complexity of architecting information and publishing it to hypertext. Third, the complexity of transitioning your content processes from exclusively focusing on paper and paper equivalents (such as PDFs) to either publishing to both or only to hypertext.

Almost every organization currently has to produce both, but most are not doing both well. As we noted in Chapter 11, *Single Sourcing* and Chapter 21, *Information Architecture*, there are major differences in how you write, organize, present, and format content for paper and for hypertext, and differential single sourcing requires different tools and techniques than are found in most single sourcing tools today. Most tools essentially allow you to design for one media and then do a more or less best effort attempt to transfer that design to the other media.

This means that in practice, whether you acknowledge it or not, you have a primary media and a secondary media. And while there are differential single sourcing techniques that can improve your ability to address each media separately, there are limits to what you can do to address both media without rewriting content to fit a different information architecture. In short, unless you are planning to go Web only (and some organizations are Web only for a large part of their content) then you do have to choose a primary media.

If your current primary media is paper, and if your future primary media should be hypertext (which it should in many cases today) then you need to consider the complexities of your chosen primary media as well as the complexities of the the differential single sourcing you will need to do to address your secondary media appropriately.

Shifting to a new primary or exclusive media can also remove some complexity from your system. Reuse, for instance, is a much bigger issue in the paper world than in hypertext, where you can link to common material rather than including it inline. Single sourcing, and differential single sourcing in particular, are larger issue in the transition period than they will be if and when you stop producing paper formats. Linking and the techniques of a bottom-up information architecture, on the other hand, belong to the hypertext world and are likely to grow in importance as you turn more of your focus to the web and online media.

---

<sup>1</sup>I say hypertext here rather than web because the web is both a delivery vehicle and a information architecture. You can publish to the web by sticking a PDF on a server. This is not hypertext. On the other hand, there are hypertext media that are not on the web, such as hypertext CD-ROMs or help systems. In terms of creating complexity in the content system, it is hypertext, not the Web itself, that makes a difference. Hypertext requires a different approach to rhetoric and information architecture and involves major differences in the linking and publishing algorithms compared to paper.

If you choose your tools and your structures primarily on the basis of the needs of the paper world or the transition period, you are likely to find yourself going through a similar upheaval in just a few years when the transition to hypertext catches up with you.

## Partition and direct complexity

Once you know where your major sources of complexity are, and have a good idea of what that complexity costs you in terms of process and quality, it is time to decide how you want to partition and divide that complexity, and to choose the structured writing techniques that will achieve that partitioning.

In some cases, we reduce the complexity that falls on each contributor by gathering a particular complex operation from the many and distributing it to one uniquely qualified or equipped person. For example, when we separate content from formatting we take formatting responsibility away from all writers and distribute it to a single designer who writes the formatting algorithm. In other cases, we do so by taking a complex operation currently being performed by a single person and distribute it out to many contributors. For example, linking is typically the sole responsibility of the writer writing the piece that contains the links. But the subject-domain approach to linking, which uses subject annotation rather than link markup, partitions and distributes the linking task three ways. The writer identifies significant subjects in their text. Other writers index or structure their content so that its subject matter can be identified clearly by algorithms, and the information architect or content engineer writes the linking algorithm (or may implement some other way of handling the deflection point in the content). In another example, the bottom up approach to taxonomy management partitions and distributes the task of terminology discovery out to the writers while distributing terminology management to a taxonomist.

As is no doubt clear by now, I regard the heart of this process to be partitioning and directing complexity away from writers. Writers are the principle source of value in a content process. Writing is an activity that requires full attention, and so any complexity that the writer has to deal with while writing is diminishing the available attention they have for writing itself, which will always result in compromises to content quality. Achieving functional lucidity for writers should be the first priority of your content management system design.

Because attention is a limited resource, there is value in partitioning and distributing tasks even when you perform all of the tasks yourself. Partitioning the tasks allows you to give your full attention to each individual task and thus perform better than you could if you tried to divide your attention between them while performing both at once.

But functional lucidity is not a fixed or universal thing. It depends very much on the writer, their background, and their skill set. It also depends, to a certain extent, on the nature of the writing task. Functional lucidity for creating entires in a reference is

different for functional lucidity for creating a slide deck or an essay on the fundamental architecture of a product. Familiarity and frequent practice of an activity reduces the amount of attention it requires to do it well, so a full time professional technical writer may be able to effectively manage far more content management tasks and may need less rhetorical guidance than an occasional contributor. On the other hand, a full time writer might need more rhetorical guidance than a subject matter expert for creating highly technical material, because they don't understand the technology or the reader's task as well. No matter how much complexity your writers can handle, however, they will always benefit from a system that maximizes functional lucidity, and partitions and directs away any form of complexity that is not germane to their ability to write the content they are supposed to produce.

That said, there are times when you should direct complexity toward writers. As we saw when we looked at terminology management, writers are the ones on the front line who best understand how terminology is being used and what subject your terminology needs to cover. A terminologist, however skilled, has less access to that information on a daily basis and could easily miss some of the subtleties that need to be expressed, or the local terms that communicate those subtleties most effectively to a particular audience. Subject annotation partitions discovery towards the writer (without actually adding any complexity, if they are already doing subject annotation for other reasons) and supports the distribution of conformance checking and decision making towards the taxonomist.

A good principle, therefore, is to always partition and direct complexity toward expertise. If the expertise is distributed, direct it outwards. If it is centralized, direct it inwards. Bottom up taxonomy, for instance, partitions language choices towards writers while allowing discovery and conformance to flow back to taxonomists.

In many cases, you want to direct complexity to algorithms. This means that you need to clarify the exact rules that the algorithm is to follow, and sometimes that means partitioning the complexity carefully so that you carve out the processes that follow consistent rules from those that require individual human attention.

The use of subject-domain annotations to drive linking is a good example of this. The task of finding a resource to link to for every significant subject mentioned in a content set is a complex and tedious one, and is it made much worse in an environment in which content is changing all the time and has to be continuously published. But the basic rules for link discovery are actually quite consistent. Where subject X is mentioned, create a link to the best resource or resources on subject X. The problem is one of identification. How do you identify the subject that is worth linking to, and how do you identify the best resources on that subject?

Where links are formed by hand, these are complex, time consuming tasks. But the use of subject annotation, combined with the indexing of topics by their subject, provides algorithms with the information they need to do the discover accurately. Actually, there is a wrinkle to this. Without terminology control for how you talk about significant

subjects, simply naming the subjects might not be enough for an algorithm. Too many subjects and topics might be misidentified due to inconsistent naming or ambiguous names. By adding type information to subject annotations (such as identifying “Rio Bravo” as a reference to a movie) we remove the ambiguity and lay the groundwork for appropriate terminology management. This is a complicated piece of complexity management and partitioning, but the result is that we remove a lot of complexity from authors while greatly improving linking, change management, and terminology management.

## Focus on complexity, not effort

Complexity is by no means the same thing as effort. Structured writing systems that are well designed to support appropriate algorithms can reduce overall effort considerably, while significantly improving quality. But where they place complexity matters. Even if a task requires less effort, adding complexity to it changes how the person assigned to that task works, and how they need to be qualified and trained. It is important to appreciate how the distribution of complexity and effort in the system you choose affects the dynamics and composition of your team.

In the end, the question of where complexity is distributed in your system is at least as important, if not more so, than the question of how much effort is avoided. The wrong distribution of complexity can not only undermine quality, it can also undermine the attempt to reduce effort. Complexity in the wrong place not only undermines the productivity of those saddled with it, it also undermines the reliability of every other algorithm, thus undermining the attempt to reduce effort and cost in those algorithms.

Distributing complexity away from writers is key because when structured writing systems distribute complexity toward writers, they don’t merely add a new and complex task that must be learned, they impose that complexity directly on the activity of writing itself. More complexity than the writer can handle will not only effect the quality of the content they produce, it will also affect the correctness of the structures that they create. The algorithms that reduce effort rely on accurate content structures. If overloading writers results in poor content structures, then the algorithms will not work properly, and that will result in more effort to fix things.

A focus on reliable handling of complexity, therefore, will likely result in a greater and more reliable reduction of effort than a direct assault on effort itself. Indeed, it is sometimes worth investing in activities that seem like additional effort, such as creating structures and algorithms to improve repeatability and conformance, or creating specific subject domain structures for a new subject to be documented, rather than knocking the content out in a generic document domain format. Avoiding this up front effort may seem like a win, but in many cases the amount of down-stream effort that will be created by less reliable content and algorithms may be far greater.

## Indivisible complexity

There are complex tasks which by their very nature have to be done by one mind. It is important to respect this indivisible complexity when planning your content system. Splitting functions that belong together, such as writing semantic blocks and combining them into sound readable rhetorical blocks, can cause severe quality problems. Remember that the partitioning of complexity always requires that you create a structure that transmits all of the information required to handle the information you are passing to the next partition. If too much information is required, then you are not reducing the complexity of the task in the first partition, and if information cannot be easily expressed in a standard way, chances are you are letting complexity fall through the cracks.

## Subtract current tool complexity

As we have noted, all tools introduce complexity into the content system. We choose them because they allow us to partition and distribute complexity better, including their own complexity. But when you are inventorying the complexity of your system, the complexity introduced by your current tools should not be part of the inventory. Those tools may be going away are a result of your system redesign, so their complexity is not part of the inventory of the inherent complexity in your content system.

## Avoid tool filters

When designing any process it is important not to see things through the filter of your current tools. Every tool reflects the tool-designer's view of how some or all of the complexity of the content system should be partitioned and directed. And since the partitioning and directing of the complexity of a system is the definition of process, tools are encapsulations of process. When you buy a tool, you buy the process it encapsulates, and long practice with the tools can shape how you view process. After a while, it is hard to imagine a process in any other terms.

Thus all too often when we spec a new tool, we essentially end up specking our old tool with some particular improvement we think will make our lives better. But many cases, the improvement we are seeking is not compatible with the current tool. (If it were, the vendor would probably have included it in their ongoing attempts to drive upgrade sales through new features.) If your old tool won't cut it, chances are you need a different process from the one incorporated in that tool, and you are going to need to break through your tool filter to envision a new process. This is why this book has focused on algorithms and structures, not tools and systems: to help you overcome the tool filter in your process design decisions.

Over the years I have seen many requirements documents for proposed structured writing systems that essentially said that the proposed system must work exactly like Microsoft

Word. This is not surprising when the people writing the requirements have used nothing but Word to create content for years. The tools you know shape how you work and what you think of as possible. As Henry Ford is supposed to have said of the Model T, “If I asked customers what they wanted, they would have said faster horses.” Even when we are dissatisfied with our current tools, we tend to want the same basic tool only more so. This is why so many structured writing tool vendors literally advertise that their editor looks and feels “just like Microsoft Word”. (Not to mention those vendors who create tools that modify Word itself.)

But Microsoft Word is a tool that sits on the boundary between the media and document domains. Using Word itself, or something that looks like Word, is usually an attempt to move its use slightly more into the document domain, but as we have seen, the WYSIWYG authoring interface invites a slide back into the media domain by hiding the structure that is supposed to be created and showing only the formatting that is supposed to have been factored out in adopting the document domain.

It is not surprising, then, that the structured writing tools that have been popular in the market to date have been predominantly document domain tools, and have tended, like DocBook, to be very loosely constrained. (It is much easier to write an XML document in a WYSIWYG editor if the underlying structures are minimally constrained, since it lets you insert whatever bit of formatting you want anywhere you want, just as in Word.

Even with tools like DITA, which, while it is still fundamentally a document domain system, is more constrained, and capable of being constrained further, tend to be used in its generic out of the box form and with a Word-like WYSIWYG interface.

Thus even when a decision-making process is based on business requirements rather than specific tools, it is often tacitly driven by existing tool sets and ways of doing things, because those existing tools and processes shape our view of what the business requirements actually are. We don’t ask for a better way to get from DesMoins to Albuquerque, we ask for a faster horse that eats few oats. This is why it is important to forget about your current tools and their associated processes and to focus on the sources of complexity in your system and how you can partition and distribute that complexity effectively.

## Select domains

Once you have a good idea of where the complexity lies in your content system and how you would like to partition and direct that complexity, it is time to decide which domains you want to work in. This is not as simple as picking one of the three and using it for everything. There are some pieces of content for which no meaningful subject domain markup makes sense – content that has no repeatable pattern of either subject matter or rhetoric. For this you will need generic markup in the document or media domain. Some content will need to be laid out by hand with an artists eye for design. That can only happen in the media domain. In practice, your content system is likely to include a mix

of subject domain, document domain, and media domain content, with some measure of the management domain thrown in where needed. In fact, major public languages like DITA and DocBook contain structures from all four domains.

Also, the places where complexity is hurting your content is not necessarily the same for all content types. The complexities that attend the maintenance of a reference work are likely to be different from those that attend the creation of a full color print add in five languages. Different types of content requires different partitioning and direction of complexity, and so require different structures from different domains.

Ideally, the choice of domains would be simple. Given the complexities you have identified for partitioning and redirection, choose the domain that accomplished the desired partitioning. But in practice it is more complicated than that because structures and the algorithms that process them are themselves sources of complexity. For example, it may seem like DITA addresses the complexities of your content reuse problem. But it also introduces complexities into the authoring process, in the form of complex markup and management domain intrusions. It also introduces significant content management complexity, both because it produces so many small artifacts, and because writers need a way to find reusable content. This creates a retrieval problem, which creates terminology management complexity. Handing these new complexities requires new tools, typically a DITA-aware structured editor and a DITA-aware component content management system, as well as new roles and extensive training. And even with those things in place, there is still a lot of conceptual and management complexity that writers have to deal with, and you still have issues with rhetorical conformance because there is no way to constrain rhetorical blocks larger than a DITA topic.

All of that additional complexity, and the cost of the tools, may be worth it if you can realize big enough gains from reuse without falling into its quality traps. For many organizations, it is the additional savings from reduced translation costs that swing the needle to the positive side, rather than reuse benefits alone. But some organizations have reported that they simply don't realize the amount of reuse that would justify the expense and complexity of their DITA systems.

DITA provides every document/management domain reuse algorithm in the book, but at a high cost in additional complexity. You might find you are better off with reuse systems that don't impose the kind of information typing constraints that are fundamental to DITA or simple document domain systems that provide a less comprehensive suite of reuse features but inject less complexity into your system. Or they might be better off with subject domain reuse tactics, which are less comprehensive than DITAs document/management domain tools provide, but actually remove complexity from the writer rather than adding it, improving both functional lucidity and conformance. It is usually better to optimize for the overall management of complexity across the system as a whole rather than to optimize one function at the expense of others.

Another source of complexity in the choice of domains is the amount of development you will have to do in house in order to implement them. Media domain tools like Frame

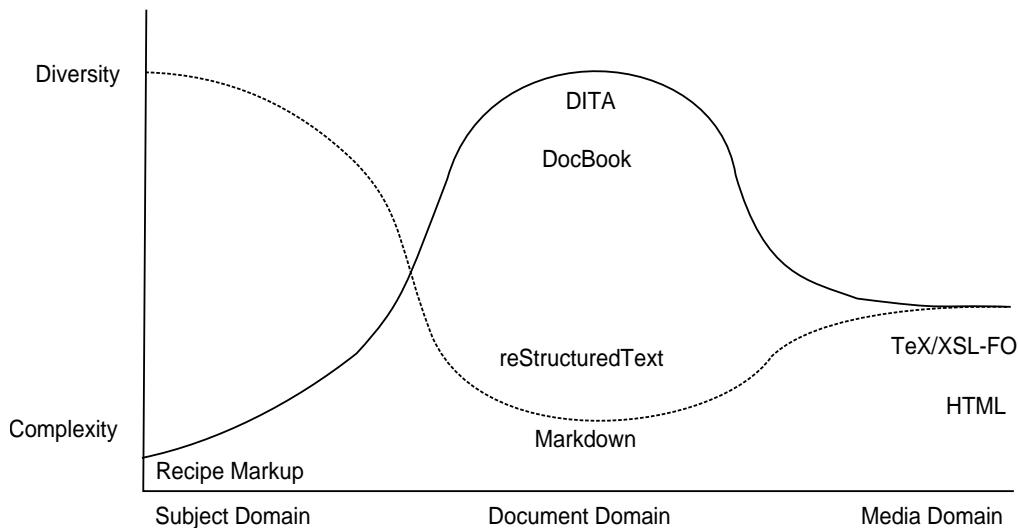
Maker and public document domain tools like DocBook and off-the-shelf DITA come with ready-made structures and algorithms. To implement a subject-domain strategy that is specific to your own content, you will have to develop some structures and algorithms yourself. This obviously adds a level of complexity to your process.

I should stress that even with media domain and public document domain tools, you will almost certainly have to do some structure and algorithm development, probably on an ongoing basis. Developing and maintaining FrameMaker style sheets is structure and algorithm development, and many organization end up using tools like FrameScript to transfer mundane formatting or management tasks from writers to algorithms. If you want your DITA or DocBook output to match your brand, you are going to have to rewrite the formatting algorithms to create the look you want. And if you do any kind of DITA specialization or DocBook customization, you are going to be in full structure and algorithm development mode, requiring all the same skills that would be needed to develop your own subject domain structures and algorithms.

As we have seen throughout this book, the subject domain provides the most comprehensive set of structures and algorithms for addressing a wide range of structured writing algorithms. In particular, it does the best job of partitioning and directing complexity away from writers, which potentially open your system up to a wider range of contributors. It also provides for higher levels of conformance, which in turn means more reliable algorithms. And with the subject domain, most algorithms work on the same set of structures, as opposed to the document and management domains, in which each algorithms requires new structures. Finally, while there are more of them, subject domain structures tend to be small with few permutations of structure, meaning that their algorithms have less complexity to handle. All this helps contain development complexity.

Nonetheless, an organization that is looking to address the complexities of creating a bottom-up information architecture, for instance, might identify the subject domain as the best way to build such an architecture, but conclude that, for them, writing and maintaining the necessary structures and algorithms was too much complexity to introduce, and might decide to go with a Wiki or a Markdown-based solution. This would throw more of the responsibility for linking and for repeatability onto writers, but that might be the right balance to strike in how complexity is partitioned and directed in that organization. (Scale can play a big role in this decision. At a large scale, managing linking and repeatability by hand becomes onerous, while developing and maintaining subject domain structures is amortized over a larger body of content.)

We can usefully map the various content management domains in terms of two properties, complexity and distribution. Complexity measure the size of a language and its level of abstraction. Distribution measures the number of different languages required to represent a set of content.



The big public document domain markup languages like DITA and DocBook are by far the most complex. Complexity is not merely a matter of the number of different elements in their schemas. It is also a matter of the number of different combination or permutations of those elements that are allowed. If, as is common, there are dozens of different elements you can insert at many points in their document structures, than writers have to understand those permutations and their consequences and information architects and content engineers have to anticipate all of the possible permutations in the algorithms they write. Because of their size and their loose structure, there is very little you cannot express in these languages, but they are also a huge source of complexity that is often difficult to partition and redirect successfully. Simpler document domain languages are less complex but also less capable. There are document design that simply cannot be executed in MarkDown, for instance.

However, the upside of the document domain is that it is very homogeneous. You can express just about any document design with DocBook or DITA. This does not mean that they support an efficient partitioning and redirecting of content system complexity, it simply means that, other than very layout-specific designs that can only be executed in the media domain, you can produce just about anything in either one of these languages. This is obviously appealing in the sense that it requires only a single tool set and a single set of training. However complicated the tools and the training may be, and however much this may limit effective participation of occasional contributors, it is at least only one thing to worry about.

The media domain, by contrast, is both moderately complex and moderately diverse. At the core of its diversity is the paper/hypertext divide. But there are multiple forms of paper (including virtual ones like e-books) and multiple forms of online media. (The initial motivation for most structured writing, separating content from formatting, is to publish to more than one of these forms.)

The move away from the media domain accelerated as we entered the transition phase between paper and the hypertext. Most organizations prior to the beginning of this

shift delivered only to paper and therefore could work in the media domain without problems. (Though, obviously, without the assistance of algorithms in handling many other functions in the content system.) But while there is no sign that paper and paper-like e-media are going away entirely, we have now reached the point where more and more content is being created and delivered only on the Web. Thus the use of media domain tools, and of simple document domain tools like MarkDown that are specific to the presentation needs of one media, are gaining in popularity. Again, the use of the media domain cuts you off from the assistance of algorithms in all other parts of the content process, but it is increasingly viable thanks to so much of information delivery once again being single-media.

The subject domain is far simpler than the media domain or the document domain. Our recipe markup is far simpler than even Markdown. But the subject domain is far more diverse. There are hundreds of different subjects, each with their own unique structures. And presenting the same subject to different audiences may also require a different subject domain structure. Obviously you don't need all the subject domain structures in the world. You only need enough to cover your subject matter for your audience, but that will still be more structures than if you have chosen to write in the document domain or the media domain.

On the other hand, most subject domain structures are very simple. Being simple does not necessarily mean that they have a small number of elements, though that is almost always the case. It also means that there are few permutations of those elements, which makes it very much easier to validate and process these documents. Because subject structures are concrete rather than abstract, and use the language of the subject matter, they tend to be clear and intuitive to writers. Writers need little or no training to use them. And they can often be expressed in lightweight syntax, removing the need for expensive editors.

However, their diversity is itself a source of complexity. It means that you can't buy structures and algorithms off the shelf. You have to create and maintain them for yourself. This does not mean maintaining a complete custom publishing tools chain. You only have to manage the processing from subject domain to document domain. You can make any public document domain language your document domain layer, and use the existing tool chain for that language for the rest of your publishing systems. (By extension, this means that if you already have a document domain system in place, you can add a subject domain layer on top of it without changing the rest of the system.)

Creating and maintaining subject domain structures and algorithms is relatively simple, because the structures themselves are simple and well constrained with few permutations. It is also made significantly simpler because most structured writing algorithms are supported by the same subject domain structures, unlike the document/management domain, where each algorithm requires different structures. And because it has good functional lucidity, the subject domain tends to produce more reliable structures, so there are fewer error conditions to worry about. With a good architecture that partitions the publishing process well, there should be few side effects to worry about.

However, the need to add new structures and algorithms will occur from time to time and will have to be met fairly rapidly when it occurs. This means that you need to keep this information architecture/content engineering capability available to your organization. This may not be as onerous a requirement as it sounds, because the truth is that you will find that there are similar requirements for a large document domain system as well, but is it definitely a source of complexity in a subject domain system.

Ultimately, then, there is no one right domain that everyone should be working in, and certainly no one right tool that everyone should be using for content creation. Rather than are three (plus one) domains of structured writing each of which offers different capabilities for partitioning and directing the irreducible complexity of the content system. Which you choose should ultimately depend on the nature and type of complexity you are dealing with and the resource available to you to address that complexity. None of the structured writing domains are destinations, none are desirable in themselves, they are merely means to an end, and that end is a content system in which all of the complexity of content creation is handled by a person or algorithm with the appropriate skills, time, and resources to handle it.

## Select tools

Once you have decided how you want to distribute complexity in your content system, you choose the languages, systems, and tools that allow you to implement that partitioning most efficiently. This may be a recursive process, since tools introduce complexity of their own which must be appropriately partitioned and distributed to make sure the tool does not introduce unhandled complexity into the system. As you begin to map tools into the system, pause to consider how the complexity that tool introduces will be handled. This may require changing other procedures or introducing other tools. If the tools introduce too much downstream complexity, or the complexity it introduces is hard to handle, you might need to consider a different tool, or even a different strategy for distributing complexity across your content system.

It is a simple fact of life that as you move content from the media domain towards the document and subject domains, the number of available off-the-shelf tools become fewer and the need to configure or extend those tools to get the result you need becomes greater. The reason for this is simple. The further you move your content towards the subject domain, the more you are relying on specific context-dependent algorithms to move it back to the media domain for publishing.

This is, after all, the point of the exercise. It is by moving functions from people to algorithms that we make sure that all of the complexity of the content system is handled appropriately and efficiently. The idea is to produce better content in less time and at less cost, and that is accomplished by handing parts of the work over to algorithms. And the further you go along the continuum from media domain to subject domain, the more particular the algorithms become to you, your organization, your audience, and your

subject matter. Thus you have to become more engaged with the design of algorithms and the structures they require.

The need to take more responsibility for structures and algorithms is not a downside of adopting more structure, therefore. It is what you are aiming for: the transfer of effort and complexity from humans to algorithms, which means the transfer of complexity to those who write algorithms. The need to employ people to write and maintain those algorithms is not a drawback. Rather, it is the desire to transfer parts of your content system complexity to these people that drives your adoption of structured writing practices.

This is not something that should be taken lightly. It is the point of the exercise, and therefore something that needs to be taken seriously and approached deliberately. The development of this capability within your organization is actually far more important than your choice of tools or even domains. Understanding the capability you need to add to the organization comes back again to how you decide to partition the complexity of your system. Remember, the goal is to partition and direct complexity so that every part of that complexity is handled by a person or algorithm with the skills, time, and resources to handle it. Once you decide what complexity you want to partition and direct away from your writers, you know what you are expecting your information architects and content engineers to handle. Quite simply, then, you are looking for people with the skills to handle that complexity, and you need to give them the time and resources they need apply their skills to the complexity you have assigned to them. Since different organizations will face different levels of complexity and will partition and distribute it differently, the definition of those roles, and their qualifications, will differ in different organizations.

## Count the costs and savings

Most of what is written about structured writing focuses on cost savings, with the biggest arguments in its favor being focused on saving from content reuse and translation. I have chosen instead to focus this book on the management of complexity. The focus on cost reduction, while it makes for an easy sell to those who must ultimately fund any structured writing project, tends to lead to a focus on one particular cost in the system, and the systems introduced to reduce that cost sometimes introduce complexity into the system that eats up all of the anticipated savings while damaging the quality of the output.

A focus on cost reduction, too, carries with it an implicitly admission that you can't think of ways to enhance the value of what you do – increasing value trumps reducing costs, and it has more upside. A focus on complexity, on the other hand, addresses both cost and quality at the same time. Unhandled complexity, or complexity handled by a person or process without the skills or resources to handle it properly, not only reduces quality, it also costs money. You may not be able to produce a neat (and misleading) spreadsheet that equates content reused with dollars saved, but a focus on comprehensive management of complexity can yield cost saving in all kinds of places, while also enhancing value.

The economics of this decision are clearly complex. You may decide that the cost of creating and maintaining the most appropriate algorithms and structures is not worth the cost or quality improvements they promise. But hopefully that decision can be made with a full appreciation of the benefits that those algorithms are capable of delivering. But whatever you decide, make sure you understand how complexity is being distributed in the systems that you implement. Make sure that the people you are distributing complexity to have the skills, time, and resources to handle it, and be conscious of the effect it will have on their productivity and the reliability of their work.

Structured writing is a tool for managing the complexity of your content system and making sure all of that complexity gets handled by people with the right skills and resources deal with it, so that it does not fall through to the reader. Getting there may require introducing new skills into your team. Don't regard bringing in those skills as a downside of structured writing. Instead, look at the introduction of those skills are the method you are using to better handle the complexity of content creation. Look at structured writing simply are the technique you use to partition and transfer the complexity around your system so that everyone on the team can do their jobs better.

---

# **Index**

---

# **Colophon**

## **About the Author**

Mark Baker is a twenty-five-year veteran of the technical communication industry, with particular experience developing task-oriented, topic-based content, and in designing and implementing structured authoring systems. He is also a frequent speaker on matters related to technical communications and structured authoring, and contributes to several publications in the field. Mark is currently President and Principal Consultant for Analecta Communications, Inc. [<http://analecta.com/>] in Ottawa, Canada.

Mark's blog, Every Page is Page One [<http://everypageispageone.com>] is focused on the idea that, in the context of the Web, Every Page is Page One, that the future of technical communication lies on the Web, and that to be successful on the Web, technical communicators cannot simply publish traditional books or help systems, they must create content that is native to the Web.

## **About XML Press**

XML Press (<http://xmlpress.net>) was founded in 2008 to publish content that helps technical communicators be more effective. Our publications support managers, social media practitioners, technical communicators, and content strategists and the engineers who support their efforts.

Our publications are available through most retailers, and discounted pricing is available for volume purchases for business, educational, or promotional use. For more information, send email to [orders@xmlpress.net](mailto:orders@xmlpress.net) [<mailto:orders@xmlpress.net>] or call us at (970) 231-3624.

---

# **Appendix A. Copyright and Legal Notices**

Structured Writing

Mark Baker

Printed in the United States of America.

Copyright © 2017 Mark Baker

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means without the prior written permission of the copyright holder, except for the inclusion of brief quotations in a review.

## **Credits**

Cover Image:

Cover Background:

Foreword:

## **Disclaimer**

The information in this book is provided on an “as is” basis, without warranty. While every effort has been taken by the author and XML Press in the preparation of this book, the author and XML Press shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

This book contains links to third-party web sites that are not under the control of the author or XML Press. The author and XML Press are not responsible for the content of any linked site. Inclusion of a link in this book does not imply that the author or XML Press endorses or accepts any responsibility for the content of that third-party site.

## **Trademarks**

XML Press and the XML Press logo are trademarks of XML Press.

All terms mentioned in this book that are known to be trademarks or service marks have been capitalized as appropriate. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.