Structured Writing

Theory and Practice

Structured Writing: Theory and Practice

Table of Contents

List of Figures

Chapter 1. Foreword

This is a book about structured writing. It proposes a theory of structured writing, by which I mean largely a breakdown of the ways in which structured writing is useful for getting quality writing done efficiently.

I feel the need right of the bat to make a distinction between a theory of structured writing and a theory of markup design. Most forms of structured writing involve the use of markup languages (though markup languages are not the only way to express the structure of content). Most of the study of markup languages, however, has not focused on their role in facilitating the writing process. Markup design theory largely concerns itself with the representation of the structure and/or meaning of texts. It intended use it to make the texts available for further use of study. From a theoretical standpoint, it is concerned with whether and how markup truly reflects important aspects of the content of the text. In other words, markup is seen principally as an means of annotating the text.

Structured writing, on the other hand, is an attempt to mange and control what is written. As such, its first concern is to constrain the text, to control what can and cannot be written and how it can be expressed. This sometimes means factoring out part of the text itself.

This does not mean that structured writing is not interested in annotating content. Annotation can be extremely useful for improving the quality of content. On the Web, annotation can also be important for making content findable and usable by downstream processes. Annotation may therefore be both a tactic and a goal of structured writing. But for structured writing, the amount and type of annotation and/or constraint used is a practical matter. It is not about achieving an abstractly “correct” markup, but about creating better content more efficiently to serve specific purposes.

The biggest consequence of this is that structured writing approaches markup as a tool for authoring. The ease of authoring becomes a key design goal. The basic tools of any craft are a key to quality. Structured content aims to make content better. Markup that is hard to use to to understand does not make content better because it distract the author from the writing task. It may even disqualify the most appropriate author from writing the content at all, if they are not willing to learn the system. Markup that annotates without constraining similarly does nothing to improve content quality because it provides no guidance and does nothing to detect mistakes or to make content easier to validate and audit.

Without the ability to constrain, and therefore to validate, there is also nothing helping to make sure that the annotation of the content is correct.

Many existing markup languages show some balance of constraint and annotation in their design. This book aims to provide a systematic way of thinking about structured writing as a system of constraints on the writing process. It will treat markup design as a discipline supporting the expression of such constraints. Markup, for this purpose, is not an end, but a means to an end, which is the creation and delivery of high quality content.

This does not mean that it is unconcerned with markup as annotation, since content delivered as annotated markup is an increasingly important part of publishing on the Web. Highly annotated content may therefore be an important output of a modern structured writing process. This book, however, is not concerned with the theory behind what annotation is desirable in published content, but with how we can constrain the writing process to produce that annotation with the greatest consistency and quality.

I have called this book “Structured Writing” for a reason, even though the term “structure content” is more commonly used today. This is deliberate. The focus of this book is on writing, on the point at which, and the process by which, ideas get written down.

There is a tendency when we think in terms of structured content to think about the stored resource, the corporate content asset, and how structured enhances the value of that asset by allowing it to be used in multiple ways to generate more revenue. This all a very good thing, but when we think in these terms we often end up creating content structures for the sake of repositories and processing algorithms. We create structures that preserve the value of the content asset for the corporation, but we do so without thinking about how the author is going to actually write content into those formats.

Failure to think about this issue can reduce the quality of your content by dividing the author’s attention during the authoring process. Equally, it fails to take advantage of the capacity of structured writing techniques to actively improve the quality of content. Thus the system designed to preserve the value of content fails as a system to create valuable content.

As this book will show, these are not incompatible aims. By focusing on structure writing as a tools for content creation, and for content quality enhancement, we can supply content that meets all of the downstream value extraction needs as well. However, the focus in this book is on the writing process itself, and how we can use structured writing techniques to make authoring easier while improving content quality.

Some readers may detect something of a split personality in this book. On the one hand, I a have very deliberately tried to cover the field of structured writing in an even handed way, and to show both the benefits and costs of the various structured writing techniques. At the same time, I have definite opinions on many topics in structured writing and I cannot pretend that these never influence the things I choose to talk about or the way I characterize them. There is definitely an element of advocacy in this work. I will, from time to time, declare my prejudices as a warning to the reader that the opinions of others on a subject are apt to diverge from my own. I may not do it as often as I should.

Much of what is written about structured writing and structured writing systems comes from people who have a vested interest in one particular system or another, either because they are associated with a vendor or consultant whose business is built on a particular model, or because they have training or experience in a particular model and want those skills to continue to be relevant. That does not invalidate what they say, but the reader should be aware the there may be a bigger picture.

Chapter 2. What is structured writing?

I am not a fan of sweeping definitions of big terms. Definitions should clarify meaning, but attempts to define terms like “content” and “content strategy” seem only to provoke arguments. The definitions proposed often seem calculated not so much to clarify meaning as to claim territory. Still the phrase “structured writing” seem to demand something in the way of definition, not because its meanings are obscure, but because they are so varied. This is a product of diversity of interests, not lack of definition. My definition is a declaration of my interests; no more.

Let’s start with the broadest possible get-your-arms-around-the-whole-thing definition:

> Structured writing is the act of creating content that obeys one or more constraints.

What is a constraint? Any rule that shapes, defines, or limits the content. Examples:

- A second level heading can only be used under a first level heading.
- A recipe must list each ingredient and the quantity used.
- An API reference must specify a return type for a function.
- A list must contain at least one item.
- A person’s name must include a salutation.
- All birds must be identified using their formal names in the Linnaean taxonomy.
- A semicolon is used to join two independent clauses.
- Start every sentence with a capital letter.

You probably see the problem here. By this definition, all writing, except perhaps the scrawls of a two-year old, is structured. And this is, in fact, the case. All content has structure. Without structure, it would not have meaning.

So our get-your-arms-around-the-whole-thing definition, while correct, is not very helpful. We need to constrain it some way.

Let’s begin by asking why it is useful to impose constraints on writing. Constraints make texts useful for a specific purpose. In order to know if a text is useful to us, we have to know what constraints it follows. When we do a Google search, for instance, we are expressing a constraint. We are saying, only show me content that is related to this phrase.

This constraint is somewhat imprecise, which is why search results are not always accurate. Most of the pages that Google searches do not explicitly say which constraints they meet in a universally agreed manner, so the search engine has to try to extrapolate what constraints the page follows and what constraints the reader’s search string was meant to express. Given that imprecision on both ends, it is amazing how well Google works.

Still, the search process would be much easier if content explicitly stated what constraints it met. This is what the semantic Web initiative is trying to achieve. This is also where the idea of metadata comes in. Metadata is a record of the constraints that a particular piece of content meets. Metadata is not going to solve the global search problem, for reasons we will explore later, but it does give us a clue about how to constrain our definition of structured writing.

> Structured writing is the act of creating content that obeys one or more explicitly recorded constraints.

By this definition, unstructured writing is writing that does not tell you what its structure is -- what constraints it obeys. This does not mean it lacks structure in the wider sense. It simply means that its structure is not made explicit. For our purposes, structured writing is writing that keeps a record of the constraints it obeys -- or at least some of them.

That “or at least some of them” is important, because it is pretty difficult to record all of the constraints that a text obeys. There are few circumstances, for instance, under which you are likely to record that a text obeys the “join independent clauses with a semicolon” constraint. Nor would you need to, since this is a universal constraint that can be easily recognized in context.

This raises the question, why record the constraints at all? Why not just follow them as you write and assume that the reader will recognize them in context?

Part of the answer lies in that fact that many constraints are not universal and cannot easily be recognized in context, or that the reader can read faster and understand better if the constraints are made explicit. Explicit constraints can take the form of labels and subheads, for instance. Thus the sections of a recipe may be labeled “Ingredients” and “Preparation”. The very fact that you call it a recipe is a record of one of the constraints it follows.

The other part of the answer, though, lies in how we manage content from creation to publication. In order to create content, we have to know what constraints it is supposed to meet. In order to manage the content, we need to know what constraints it does meet.

Structured writing, then, plays a role in both content creation and content management. As a term, “structured writing” is part of a cluster of terms that includes “structured authoring,” “structured content,” and, more recently, “intelligent content.” Certain communities prefer one term over the other. Tool vendors, for instance, almost always use “structured authoring.” Many content strategists have recently adopted “intelligent content”.

These terms all have the same center, if not always the same edges. “Intelligent Content,” in particular, seems to imply a particular application of structured writing, rather than being simply a synonym (though if it grows in popularity, the term’s implications will inevitably grow looser). And clearly there is a difference in emphasis between “authoring” (a verb), “content” (a noun), and “writing” (both verb and noun). I am choosing to use “structured writing” for this book because it bridges three concerns - the act of composition (writing as an act) and the act of management (writing -- or content -- as an asset to be managed) and consumption (writing as a product to be consumed).

On the management front, suppose we manage a collection of content that includes recipes and we want to select all the recipes that are good matches for a Pinot Noir. We could attempt to do this Google style by searching for “recipe Pinot Noir”. This would probably get us a lot of correct results, but probably some incorrect ones as well. Some other types of content might contain the words “recipe” and “Pinot Noir” (such as this essay on the definition of structured writing). Some recipes might use Pinot Noir as an ingredient rather than a wine match. Some recipes might be missed because they do not include the word “recipe” (most don’t).

We could get better results if we could do a query on two explicit fields. Something like:

```
RETURN topic WHERE type='recipe' AND wine-match='pinot noir'
```

But this can only work if our stored content contains explicit metadata that records these particular constraints, type=’recipe’ AND wine-match=’pinot noir’.

If we did not create and store that metadata with the content when we wrote it, we will not be able to respond to this query. Supporting this type of query means that we have to explicitly create support for it in our content.

This means that a piece of structured content is structured for a particular purpose that you thought of at the time you created it. The content is structured for that purpose or set of purposes you thought of, but is unstructured for other purposes. Just as a hat can be the right size for Tom and the wrong size for Harry, a piece of content can be structured for Mary and unstructured for Jane. It all depends on context.

So let’s further constrain our definition:

> Structured writing is the act of creating content that obeys one or more explicitly recorded constraints that serve a defined purpose.

This does not mean that content that has been structured for one purpose may not also turn out to be structured for another purpose. This frequently turns out to be the case. On the other hand, it means that you cannot ever be sure that the structure you apply to your content today will apply to future purposes that you do not yet understand. In this sense, the idea that structured writing can “future proof” your content is misleading. It can only guarantee future uses you foresee today.

And what this means, in any given context, is that when someone says, “we are going to switch to structured writing”, what they really mean is that we are going to add an additional bit of structure, an additional set of constraints, that we did not apply before. Any piece of recorded content has been structured for some specific purpose or purposes. We are simply talking about adding more structure for additional specific purposes.

In this respect, saying that we are going to start structured writing is like saying we are going to start eating healthy or adopt an active lifestyle. Assuming that your current diet was not pure poison, and that you have not been completely stationary to this point, you mean that you are going to eat a diet that is more healthy and live a lifestyle that is more active than it was before. In the same way, adopting structured writing really means being more structured than you were before.

Another use of constraints in content management is to guide authors to make sure that they create content that meets requirements (requirements is another word for constraints). Rather than authors deciding ad hoc on the constraints their content will follow and recording them as they go, it is often useful to establish a set of constraints that authors must follow while writing and to establish them up front before the content is written. So:

> Structured writing is the act of creating content that obeys one or more predefined and explicitly recorded constraints that serve a defined purpose.

There is nothing new about this: templates and style guides are examples of predefined constraints that authors are required to obey. However, this is not a book about how to write a style guide, so I need to constrain the definition further:

> Structured writing is the act of creating content that obeys one or more predefined and explicitly recorded constraints that serve a defined purpose, in a format readable by machines.

So far, nothing in the definition has specified that the explicitly recorded constraints have to be machine readable (though the query example above implies it). There is a lot of content out there that uses a consistent layout and headings across multiple documents to explicitly record the fact that they obey a set of constraints for a defined purpose. An API reference, a tax form, and a tide table are all examples of this type of content. All these formats are entitled to call themselves structured, but their structures may only be intended to be read by people, not machines.

Using a format readable by machines (such as XML) can add several powerful capabilities to structured writing. Making the structure of a text machine readable allows us to enlist the help of machines in making the content better, and also to hand many management and production tasks over to machines so that writers can focus more on content.

Of course, any piece of content created on a computer is stored in a format that is readable by machines. In most cases, however, such formats only record those constraints as are required by the software itself for its own purposes. But since those constraints are predefined, explicitly recorded, and serve a defined purpose, any old Word doc technically still meets my definition. To make structured writing distinct and worth talking about, we need to constrain the definition once again:

> Structured writing is the act of creating content that obeys one or more predefined and explicitly recorded constraints that serve a defined purpose, in a format readable by machines, with the goal of making the content better.

There several other ways I could have gone with this final constraint. I could have added “with the goal of separating the editing function from the publishing function to realize greater control over publishing options.”

I could have said, “with the goal of making content interchangeable between systems and organizations.”

I could have said, “with the goal of owning the content storage format so that I truly own my content and it possibilities.”

Those are all legitimate aspirations, and all things that people have done successfully. But they are not quite as interesting, nor do they have quite the same potential for good, as the goal of making the content itself better. They are publishing and content management aims, and while those aims can definitely contribute to making content better, they are only contributions. They are not the whole story of structured writing and what we can accomplish with it.

This book is about how we can use structured writing to make ourselves better writers and produce better content. That involves the use of machines as tools to help us write better -- just as many other professions use machines to make them more proficient. But the point is to be better writers.

One other thing that this book is not about: it is not about the use of markup for the academic study or representation of texts. This is a matter of significant concern in the humanities, where the Text Encoding Initiative[http://www.tei-c.org] (TEI) plays an important role in determining standards for representing texts for study. But TEI is not about the writing process. It is not about using structure writing to improve content quality. For some purposes, the TEI format might become an output format for a structured writing system, but other than that, it is outside the scope of this book.

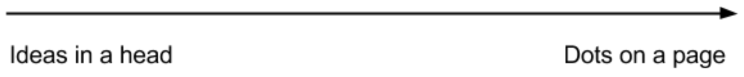
Part I. The Structured Writing Domains

Table of Contents

Chapter 3. The Three Domains of Content Structure

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 of ink or pixel on a page or screen).



The writing and publishing process is all about how we get from ideas in a head to dots on a page.

In the simplest case, an author writes down their ideas with pen and paper. The entire translation from ideas to dots takes place in the author's head. The content is recorded directly in the final physical form.

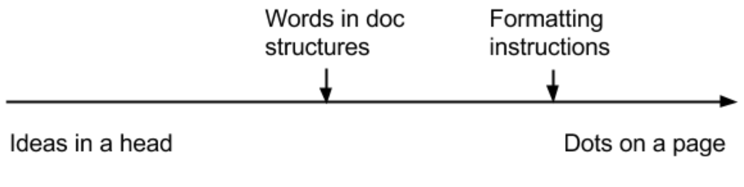
Recording content directly in physical form is rare these days. In most cases, the content is recorded earlier in the process, and software takes it from the place it is recorded to the final rendering on paper or screen.

Word processing and desktop publishing software, and various approaches to structured writing all establish a point in this process from ideas to dots where the content will be recorded by the author, and then provide algorithms to complete the movement from that point to dots on a page. The differences between them lie in how and where in the process the content is recorded, and the kinds of algorithms used to take it the rest of the way.

When you use a word processor, you record content using abstract document structures -- lists, tables, paragraphs. Separately, you can define styles which are then applied to the content before it is sent to a printer to be rendered into ink on paper.

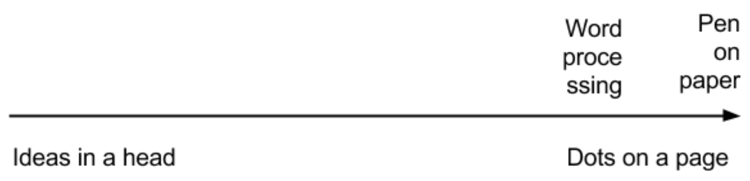
What is notable about this process is that the content and the styles are defined separately. The author creates two different kinds of data: content and styling instructions. These two streams are then fed to the software to drive the final rendering.

This combination of multiple streams of data and instructions is a common feature of any system that records content before the dots stage. All content must eventually be displayed as dots and way you capture content that does not specify the position of dots, requires instructions -- algorithms -- for turning the recorded format into the exact locations of dots on paper or screen.



In a modern word processor, the software can take content and style instructions and combine them on the fly to create a WYSIWYG display, creating the illusion that the author is producing the final rendered output directly. As we will see, the earlier we record the content, the more difficult this becomes.

What we commonly call "structured writing" is simply a process of moving the recording of the content still earlier in the process from ideas to dots. What this implies, of course, is that WYSIWYG word processing is not the opposite of structured writing. It is actually just a point on the structured writing continuum: a point quite close to dots on a page.



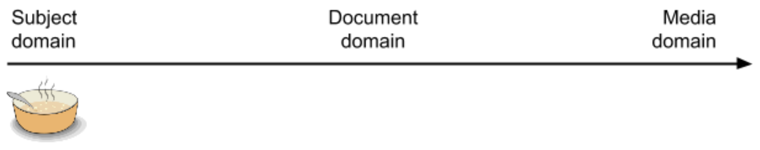
Other approaches to structured content simply move the moment at which content is recorded closer to the ideas-in-a-head end of the scale.

Of course, the text itself always conveys the author's ideas, at least as well as the author is capable of expressing them. What is at issue here is what part software plays in the process by which the ideas in the author's head make the journey to dots on a page. That role may as simple as allowing different dots to be rendered on different surfaces from the same source files, or the software may play an active role in shaping and organizing the text.

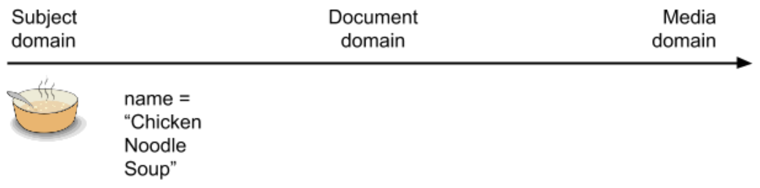
The three domains

We can divide this journey from ideas to dots into three domains: the subject domain, the document domain, and the media domain.

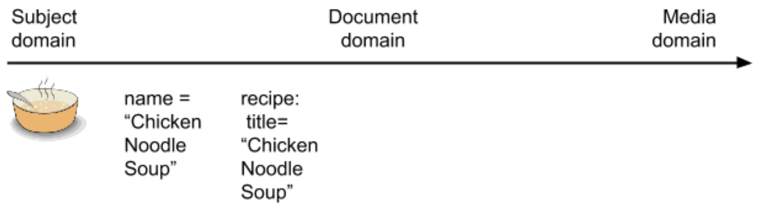
Let's suppose that an author is writing a recipe for chicken noodle soup. They start out with the idea of a soup with chicken and noodles. This is an idea about the subject matter and not yet any form of content.



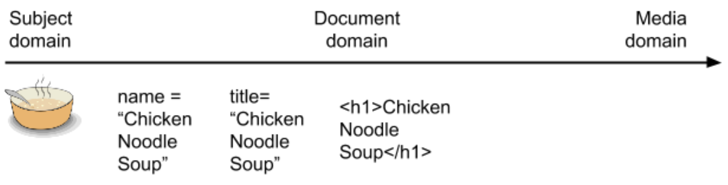
They then decide to give the dish the name "Chicken Noodle Soup." Unlike the soup itself, the name is content. However, it is not yet part of a document. It is a piece of data in the subject domain.



Then the author decides to write down the recipe for Chicken Noodle Soup. They will probably use the name of the dish as the title of the recipe. A title is an object in the document domain. Documents have titles. This is a particular kind of document, however -- a recipe. While a recipe is a document type, it is a document type with a strong relationship to the subject domain -- the subject of dishes and their preparation.



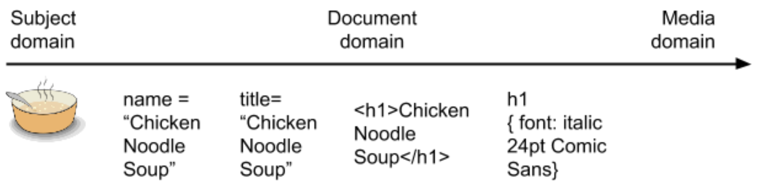
As the process continues, some format for publishing the recipe is chosen. This might be HTML. HTML is a markup language in the document domain. That is, it contains markup for typical document structures such as title, paragraphs, and lists. To express the title of the recipe in HTML, we turn the declaration of the title of the recipe into a declaration about an HTML heading level:


Subject domain
Document domain
Media domain
name = "Chicken Noodle Soup"
title= "Chicken Noodle Soup"
<h1>Chicken Noodle Soup</h1>

In doing this, we have greatly weakened the association with the subject domain. At this stage we know that "Chicken Noodle Soup" is the title of a web page, not that it is the title of a recipe, and not that it is the name of a dish. However, we have also made it more specific what type of document this is: a web page.

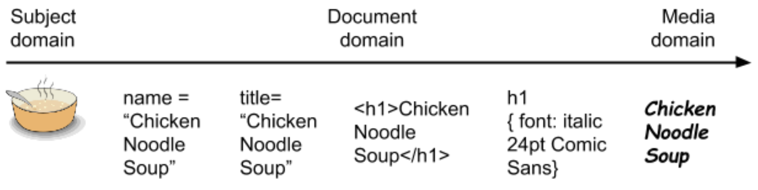
In deciding that it is a web page, we have also started to bring in something of the media domain. HTML is by no means exclusive to a single media. It can be use to create web pages, help platforms, mobile applications, and even printed pages. At the same time, it includes a number of presumptions about the media domain in which the document will be displayed.

If you follow modern practice, however, your HTML should not include any specific information about how the document will be rendered -- what fonts will be used, how big the margins will be, etc. We move the content further into the the media domain by creating a CSS stylesheet that specifies these matters. (This is one of those cases of two converging streams of information that I mentioned at the beginning.)


Subject domain
Document domain
Media domain
name = "Chicken Noodle Soup"
title= "Chicken Noodle Soup"
<h1>Chicken Noodle Soup</h1>
h1 { font: italic 24pt Comic Sans}

Adding the CSS moves the entire package further into the media domain. Notice, however, that in this case, all of the document domain information in the HTML format has been retained. The movement from the subject domain through the document domain into the media domain does not have to mean that information from an earlier domain has to be thrown away as you add information from the later domains. For instance, HTML5 microformats allow you to retain subject domain information all the way into the user's browser. Traditional publishing techniques tended to throw away subject domain information as the process advanced. Today, that information is more likely to be retained as long as possible. (One of the implications of the term "intelligent content".)

When the resulting page is loaded into a browser, dots are painted on the screen in the appropriate shape.


Subject domain
Document domain
Media domain
name = "Chicken Noodle Soup"
title= "Chicken Noodle Soup"
<h1>Chicken Noodle Soup</h1>
h1 { font: italic 24pt Comic Sans}
Chicken Noodle Soup

This process involves translating the document and media domain information in the HTML and CSS into the graphics primitives of the platform on which the content is displayed -- basically lighting up dots on a screen or printing dots on paper. This final step will destroy all document and subject domain information, but since the browser retains the HTML source, any information that has reached that point is available to code running in the browser.

Declarative vs. Imperative

As we move from the subject domain through the document domain and into the media domain, we will find that we are moving from the declarative to the imperative. A declarative statement simply says what something is. An imperative statement gives an instruction. Ideas in the subject domain are purely declarative. Dots are created by purely imperative software instructions.

As we proceed from the subject domain to the media domain, therefore, we are continually converting declaratives into imperatives. For example, the CSS rule:

```
h1 { font: italic 24pt Comic Sans}
```

turns the more declarative HTML <H1> tag, which simply declares that a piece of content is a level one heading, into a specific set of instructions about fonts.

As we proceed from ideas to dots, the content contains fewer declarations about the subject matter and more of a declaration about a document, and then a little less about the structure of a document and a little more of an instruction to a rendering process. By the time we reach the HTML stage, we no longer have any declaration that associates the words with soup (unless we use microformats). There is only an document structure (this is a major heading, this is a list), which then becomes more concrete at each of the following stages until it is just instructions for placing dots on a surface.

When we look at formats used to record content at each of these stages, we will find that in the subject domain the format is entirely declarative. In the document domain we typically find a mix of declarative and imperative elements. In the media domain, we have only imperatives (though some imperatives may be declarative in form).

The implication is that as we move the point of recording earlier in the process, we are actually doing the opposite: we are removing imperatives and substituting declaratives. We are turning instructions about how a document should look into declarations about how the document is organized or what the content is about.

One of the classic descriptions of structure writing is “separating content from formatting.” This is how we separate content from formatting, by replacing instructions about how to format the content into declarations about the structure or subject matter of the content. If you have wondered exactly how content gets separated from formatting, and what that looks like in practice, this illustration should clarify how any movement away from the media domain towards the document domain or from the document domain towards the media domain is a case of further separating content from formatting. And it should emphasize that the process can look very different depending on where you start from and where you are going.

Why is it valuable to move the point of content creation back from the media domain into the document domain or the subject domain, from the declarative to the imperative? There are several reasons:

Declaratives give you choices. A declaration about the structure of a document or the subject of a sentence does not force you to display it in a particular way. An imperative, on the other hand, is an instruction that has to be obeyed. Moving from the declarative to the imperative is a process of making choices. Moving from the imperative to the declarative is a process of postponing choices.

Declaratives can be constrained. We defined structured writing as the act of creating content with certain constraints. Declaratives can be constrained much more easily and with much greater precision than imperatives. Declaratives can also be audited and validated with greater precision than declaratives.

Declaratives are easier to write. For an author to create declaratives, they need only a know the subject matter and the format of the declarations. For an author to create imperatives requires that they understand the language and effect of the instructions. In the case of embedded algorithms, it requires that the user understands the language of those algorithms.

We will look at these advantages, and the many different content processing algorithms you can enable with a more declarative content structure, in later chapters.

Chapter 4. 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 this only really means “not structured in a way that is currently useful to me.” The fact is that all content has structure, and we can actually find all the patterns and techniques of all forms of structured writing in the media domain. This makes it a good place to study the fundamentals of structured writing.

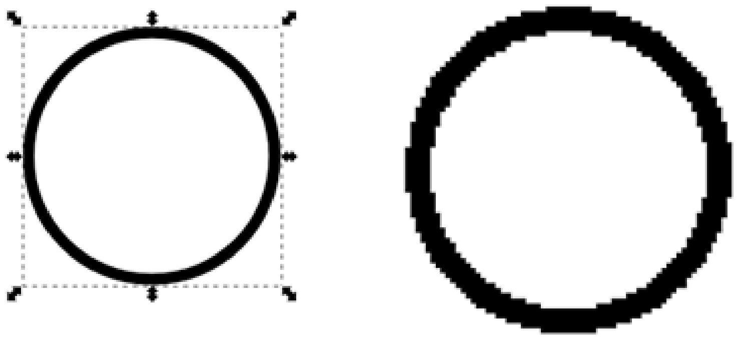
At its most basic, a hand guiding the pen over paper or chisel over stone is working in the media domain through direct physical interaction with the media. This is not structured writing, in the sense we mean it hear. No computable structures are involved because no computer is involved.

We don’t do a lot of physical writing these days. Mostly, we interact with computers which then drive machines that interact with the physical media in the form of screens, paper, or engraving surfaces. The closest you can get to pen and paper in the computer world is to use a paint program, 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. You can then print those dots to paper. 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. This allows you to type letters on the keyboard. However, those letters are still recorded as a matrix of dots, not as characters, so you can’t go back and edit your text as text, only as dots. (This type of program is technically known as a raster graphics application, where the word “raster” means an array of pixels or matrix of dots.)

To work more efficiently, 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, which records characters as characters, but a text editor does not keep any formatting information (unless you create markup -- but that would be getting ahead of ourselves). For most publishing purposes, plain text is inadequate. We need to maintain the ability to format the document.

A vector graphic program creates graphics as a collection of objects ("vector" meaning the mathematical representation of a shape or line). For example, it allows us to create a circle as a shape, described mathematically in computer memory, rather than as a matrix 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. 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.



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 (Paint), right.

In a typical 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 or photo editing program. This is an instance of what in structured writing circles 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.

To give you some idea of how a circle might be represented as an object, here is a sample piece of Java code for creating a circle:

```
Circle circle = new Circle();
circle.setCenterX(100.0f);
circle.setCenterY(100.0f);
circle.setRadius(50.0f);
```

This code creates a new circle object and specifies three values that you would need to draw a basic circle, an X and Y coordinate for the center of the circle, and the radius. (The object actually needs more fields, like line width, color, and fill, but I have simplified the example for size and clarity.)

When a vector graphics file is displayed or printed, a rendering algorithm turns those objects into a matrix of dots on the current media, using more dots for higher resolution media, and perhaps substituting gray dots for colors when printing on a black and white device.

One way that this can be done is using PostScript, which works by moving a virtual pen over a virtual surface, recording dots as it goes. So the description of a circle object in terms of location and radius is turned into a set of instructions for moving that virtual pen:

```
100 100 50 0 360 arc closepath
stroke
```

This says, “draw an arc whose center is at coordinates 100, 100, and whose radius is 50, starting at degree 0 and proceeding to degree 360”. This is the algorithm guiding a virtual pen over virtual paper, just as the author we first described moved a real pen over real paper.

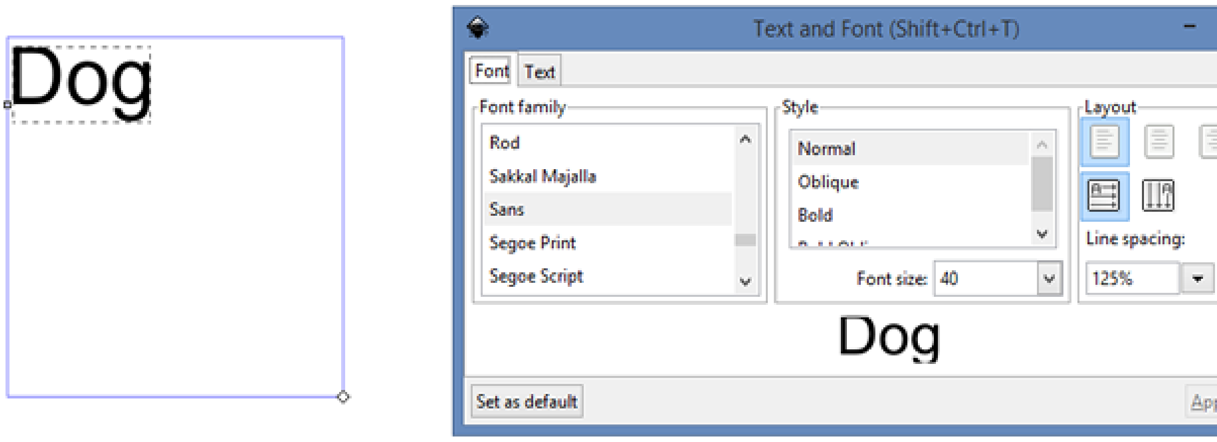
If representing the circle as an object creates an abstraction of “circle” in the media domain, this bit of PostScript code takes that abstraction (a circle as a set of coordinates) and (in concert with the underlying graphics driver) makes it concrete again (dots on a media).

The point here is not to understand the code, but to see that by creating a more abstract form of the content, we can make it easier to work with, and that we can use algorithms to render it back into concrete form. All of structured writing comes down to finding better ways to represent content for a particular purpose (more efficient, more versatile, more verifiable, more precisely constrained) and ways to transform those representations, eventually, into dots on a screen or paper so that the content can be read.

This is a pattern we will see as we look at structured writing across the three domains. Rather than storing the image of the document to be printed, the computer stores an abstraction of the document which consists of raw text combined with (or, in some cases, replaced by) additional pieces of information which are commonly called metadata. This metadata can then be used to manipulate the content in whatever way you need, and also to drive algorithms to convert it back into a concrete media domain representation that can be displayed to readers.

Adding metadata to text creates structures that the computer can create, store, and manipulate. It is useful to think of structures as objects, just as we think of the representation of a circle in a vector graphics program as an object. And indeed, a vector graphics program will let you create a text object.

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



Font size: 40

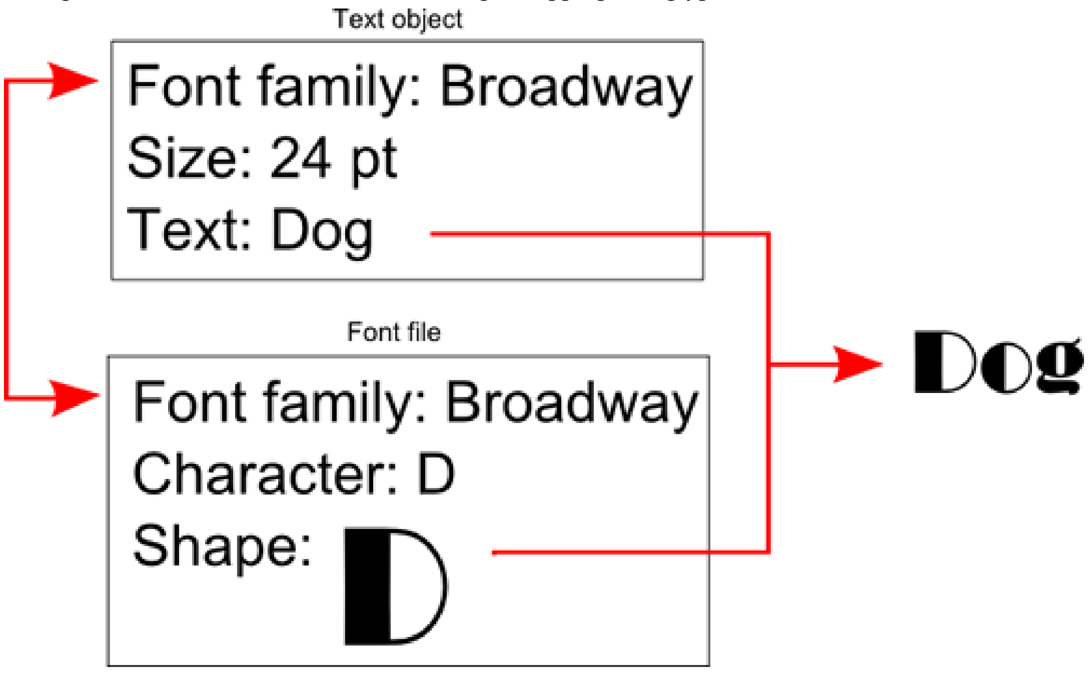
Line spacing: 125%

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

Abstracting out font information

A vector graphics program displays text in a chosen font. If your change the value of the text object's font attribute, it will immediately redraw the text in the new font. 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. Whereas the representation of the text in the paint program included the shape of the letters, in the vector graphics program it does not. That information has been factored out.

The shape of the characters (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.



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

This is a pattern we will see over and over again in structured writing. 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, like the font, and move it to a separate file. By moving out information that is constant for a given application (the shape of a capital F is the same for all capital F's for text in a given font), we simplify the format of the information we are preparing and keep the downstream presentation more consistent.

Adding more structure to content means adding more metadata to it. But if we just kept adding metadata, it would very quickly overwhelm the content. So whenever we can we carve some metadata off into separate files and create rules for pulling it back in when it is time to publish the content.

Whether it is, "the capital F will always be this shape" or "the list of ingredients will always have the ingredient name aligned left and the quantity aligned right", filtering out these invariant rules into a separate file is a key part of structuring content.

This means that 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 separate out these invariant properties into separate structures, and where doing so will contribute to the core goal of structured writing, which is to make content better.

Using constraints to improve efficiency

Writing a document in a vector graphics program is certainly better than trying to write it in a paint program, but it is hardly ideal for writing long documents. This is why most of us use tools that are designed specifically for writing documents.

A vector graphics program knows nothing about the document domain. It works purely in the media domain, and pretty much lets you put shapes and text boxes anywhere you like. This give you enormous freedom, but it also makes you do a lot of extra work if what you want to create is a typical document that is basically one long text flow with some headings and graphics thrown in.

Word processors and desktop publishing programs make it easier to create documents by introducing some document domain constraints, as well as some higher-level tools for managing large text flows. 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. Common features like tables are supported as objects than can exist in text flows. Text flows are allowed to cross page boundaries. New pages are created automatically as text expands.

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 decision about positioning of elements, which makes creating documents faster and more consistent. As we said, structured writing is about making content that obeys constraints, and these basic document domain constraints are the next step in that journey.

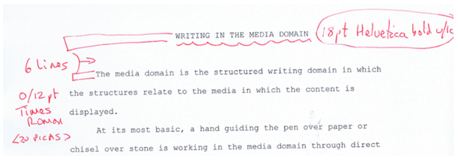
These constraints are not without their negative consequences. You always give up something when you impose a constraint. There are certain page layout effects that are difficult or impossible to achieve in Word or FrameMaker because you have given up some of the liberty of a vector graphics program. (You also gave up some liberty in moving from raster to vector graphics, which is why photo editing, which requires adjusting individual pixels, is still done in raster rather then vector format.) There is always a trade off of some sort when your make these moves. When you impose constraints on your content you must understand the trade off you are making and what the costs are on both sides of the ledger.

But while they make authoring easier by introducing document domain constraints into the program, both word processors and especially desktop publishing programs are still very much located in the media domain. While there are some basic document domain objects being created under the surface, the author sees and interacts with the media domain representation of those objects on their screen. And the way that these programs allow writers to distinguishing one block of text from another is almost always be applying formatting styles. The document domain objects they provide are just enough to give the author something to hang media domain styles on.

Enabling the media domain

Providing the ability for the author to work in the media domain was at the heart of the desktop publishing revolution. For centuries, scribes worked directly in the media domain, using pen and ink to inscribe words and pictures on paper or velum. With the printing press, however, 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 would be created later by 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. The M in XML stands for Markup. (But modern structured writing does not always have such a clear cut distinction between text and markup as this usage implies, as we shall see.)


18pt Helvetica bold u/lc
6 lines
0/12pt Times Roman
<20 picas>

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.

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

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. (Here is that same pattern again -- two steams of objects, merged by an algorithm to produce a new stream of objects.)

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 ???.

Theoretically, those inline styles could be added by a document designer after the author had written the original HTML, but separating those tasks was not the intent of the desktop publishing movement. The intent was to combine the job of designer and writer into one, effectively moving the author from working in the document domain to working in the media domain.

The inefficiency of applying styles to every element soon became apparent. Stylesheet mechanisms were devised to separate the media domain elements of a document from the document domain elements.

Using styles rather than direct formatting does not mean that you have moved from working in the media domain to working in the document domain. It just means that you have moved from working in an unstructured way in the media domain to working in a structured way in the media domain.

The set of document domain objects that you are working with is so small that it does not even distinguish between a heading and a list item. Both are just paragraphs with different styles applied. What distinguishes them is not what type of object they are, but what styles are applied to them. (The use of the word “paragraph” in these programs does not have much to do with its meaning in the study of composition. “Block” would be a better word.)

Actually, it is not quite so clear cut as that. While most DTP and word processing programs create lists as paragraphs with styles applied, rather than separate list objects, they do provide interface controls of creating lists as if they were objects. In this respect, they are similar to paint programs that let you draw shapes as objects that are resizable until you release your mouse and then become just dots. You can create a list as an object in these programs, but the result is just a set of styled paragraphs, not a permanent list object.

This sort-of-is/sort-of-isn’t-an-object approach is a pretty accurate match for how many authors think about what they are doing when they use these programs. Sometimes they are thinking about creating document domain objects and sometimes simply about formatting.

They don’t of course, make the analytical distinction between media domain and document domain, but they are clearly sometimes thinking “make a paragraph” and sometimes “insert a blank line”. Sometimes they think about creating a numbered list. Sometimes they just type a number at the beginning of a line and expect the application to format it appropriately.

Being too strictly object oriented in how a document is created could slow the writer down by forcing them to think in terms that are not natural to them. Thus these programs -- Word much more so than Frame -- let the author do it whatever way seems natural to them.

But while being fuzzy about this can make everyday writing tasks less mentally demanding, it can also have unwelcome consequences when you tackle longer documents or more varied or complex publishing scenarios. Word’s difficulties with maintaining list numbering are well documented. Frame does a better job of this kind of thing, but has a steeper learning curve because its greater reliability in handling these things comes at the expense of forcing the writer to think more about what kind of object they are creating.

Word processing and desktop publishing, therefore, tend to roughly straddle the boundary between the document and media domains, with authors thinking partially in document domain terms and partially in media domain terms, and the applications creating and storing objects and structures from both domains. In many cases, there is both a document domain way and a media domain way to do something in these applications, which simplifies the authoring thought process but produces files that are often unreliable for downstream processing, or that require significant cleanup if, for instance, some general change of style or publication to a new media is required.

The only way to solve these problems cleanly is to move writers, and the files they create, out of the media domain entirely. The next stop on that path is the document domain.

Chapter 5. Writing in the Document Domain

As we saw in our examination of the media 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 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. For that we need to move to the document domain.

But it is not simply a matter of document domain constraints being found in the document domain and media domain constraints in the media 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 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 would then be defined with more spacing above.

The problem is that an author can forget to use the first-item-of-list style. Or they could add a new first item above it 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, therefore, is not actually to enforce it on the author, but to factor it out altogether.

To factor out the spacing-above-lists constraint, we remove the need for the author to specify the style at all. You can't do this in a typical media domain application, because the only way to create a list is to apply list formatting to a set of paragraphs. To factor out the formatting step, we need another way for the author to specify that a list is a list.

To do this, we create a list object -- not a styled paragraph, but an object that is specifically a list. Lists belong to the document domain because they are a common rhetorical tool that is not specific to any one subject area (subject domain) and can be formatted in a wide variety of ways (media domain). Once we have a list object, we can create rules -- in a separate file -- about how lists are formatted.

Structurally, a list object looks something like this:

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

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

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

Now the correct spacing above lists is not something the author has to think about. In fact, it is not something they can manipulate even if they want to. Authors just create document domain list objects. Media domain list formatting rules have been factored out of the author's world. The media domain constraint about spacing above a list will now be followed automatically and reliably by algorithms.

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 objects in the document domain rather than applying 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 objects 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 abstract out a set of style metadata 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 structures.

The problem is, extending the document domain is not as easy as extending the media domain. For one thing, document domain structures are more abstract than media domain styles, which makes them harder to think about. For another, content in the document domain has to be processed by algorithms before it can be published in the media domain, which means that when you create new document domain structures, you need to create the corresponding algorithms as well, which requires a skill set somewhat more complex than defining styles sheets.

This added difficulty in extending the document domain means that alternatives have been developed to manage the need for different document domain structures. There are four basic approaches:

1. Languages like markdown, restructuredText, and HTML provide a default set of structures, but no way to add your own. Either they provide enough document domain structures for your needs or they don't. Since these systems are not extensible, some people will create variants of them to meet

specific needs (that is, extend them by modifying the core language itself). Thus there are a number of variants of Markdown designed for particular purposes.

2. Metalanguages like XML provide a way to define your own structures, but no default set to start with. If you start from scratch in XML, you will need to define all the document structures you need for yourself. The upside of this is that you can constrain those languages very closely to exactly meet your needs. The downside is that you then have to write and maintain the algorithms as well.

3. Systems like DITA, DocBook, and SPFE provide a default set of structures, and a way to add your own if you need to. DocBook and DITA both provide a large set of standard structures, whereas SPFE provides a library of simple structures that you can combine to meet specific needs.

4. Moving content to the subject domain can factor out much of the variation in document domain structures. In many cases, you can use systems like SPFE, DocBook, and DITA to extend into the subject domain as well. There are also a large number of purpose build subject domain languages. The next chapter will look at how you can factor out document domain structures by moving to the subject domain.

Which of these approaches you choose will depend on what you need to do -- which constraints you need to observe and express in your content.

Now, let’s get back to the discussion of lists. If we need more than one type of list object in the document domain, we either need to extend our document domain language with new lists types, or choose an existing document domain language that already has all the list types we need. But 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. To keep the author in the document domain, we need to create document domain objects that contain the metadata we need to make those decisions at the formatting stage.

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

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 factored out of the document domain structures.

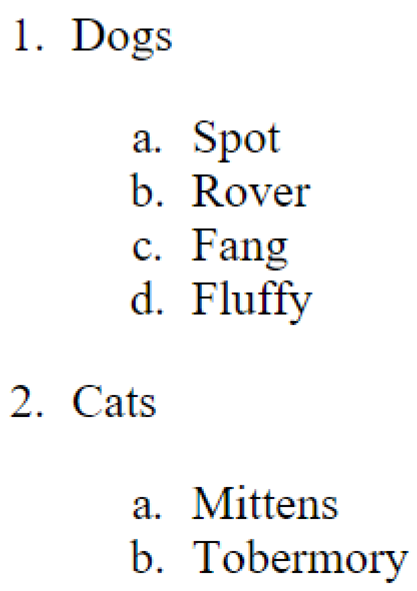
Does the need for separate ordered and unordered list objects imply that we will need a separate document domain list structure for every possible way a list could be formatted in the media domain? No. In fact, that would really just be working in the media domain by proxy. When we work in the document 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.

For example, consider nested lists. While nested lists are formatted differently, we don’t need a separate nested list document domain object. 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 they are both ordered lists. In the media domain, one will be formatted with Arabic numerals and the other with letters.

Figure 5.1. One document domain object, two media domain styles



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 objects 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. 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 structured, but does not always get it right.)

In the document domain, document structures are almost always implemented hierarchically. List items are *inside* lists. Nested list are *inside* list items. Sections are *inside* chapters. Subsections are *inside* 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.

For example, HTML, though a document domain language, is relatively flat in structure. It has six different heading styles H1 through H6. Docbook, by contrast, 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, 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 much more difficult that simply starting a new paragraph in Word or Frame.

These considerations are another reason why there is more than one document domain language available in the world, and why extensibility is important. A single document domain language that captured all the document domain structures that anyone might want would be very large and very complex.

Worse, a universal document domain language would not express the individual and specialized document domain constraints that individual organizations need to manage the critical parts of their content creation and management processes. Much of the virtue of going to the document domain lies in the ability to impose such constraints, which means that the world has and needs many document domain languages. And in the class of document domain systems that are designed to be extensible (for example: Docbook, SPFE, DITA) we will also find that they are designed to allow you to add additional constraints as well.

Writing an algorithm to transform a large unconstrained document domain language into the media domain would also be a daunting task, since it would need to have a rule to format every single combination of document domain structures that could occur in that language. With a large number of elements and few constraints on how they can be nested, the number of combinations would grow exponentially.

In fact, you will find in practice that some large document domain languages, even thought they are constrained in many ways, can permit some combinations of structures for which their common formatting scripts do not have full support, meaning that you often need to check your outputs and possibly fix your document domain markup to get it to format correctly. This is not necessarily the fault of the language. Technically, it is the fault of the scripts. But a language that allows for a lot of edge cases combinations rather invites the creation of scripts that don't cover all those rare or unexpected cases due to the expense of creating, testing, and maintaining all the necessary code.

Constraining structure

There are other reasons for working in the document domain beside abstracting out formatting rules. One of the main ones is to constrain how documents are structured. For example, lets 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, for instance.

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, but 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.

```
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 an author to add a graphic without a figure, title, and caption. A document that lacked these elements would be rejected by the schema and reported either by the editor or by the processing software.

(The figure number would be generated automatically, of course, just like the numbers in an ordered list. That constraint has been factored out rather than enforced.)

This raises an important aspect of document domain languages, and one of the reasons that they tend to have hierarchical rather than flat structures: constraining document structures.

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 constructed. For a start, there will be procedure objects, which will have step objects nested within them. Step will only be allowed to appear inside procedures. 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 to 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.

Backsliding into the media domain

HTML was originally designed for sharing scientific papers. It was not designed to strictly control the organization and presentation of scientific papers -- it was designed more to accommodate requirements than to constrain them -- but it does have features that betray its origins. One example is definition lists. Definition lists are a common feature of scientific papers, which need to precisely define how they will use key terms. But as HTML has come to be adopted as generic language of web pages, the definition list (dl) structure has come to be treated as a generic labeled list structure, used for all kinds of things other than definition lists.

This highlights one of the challenges of structured writing, which is to make sure that structures do not get used for purposes other than what they were intended for. This often happens when people look for an easy way to create an effect in the media domain. If the writer wants a piece of text formatted in a particular way, but the only document domain structure that formats that way is intended for something different, it is easy for them to use the structure incorrectly to get the formatting effect they are after.

But this means that the document domain structure that is being misused no longer expresses the constraints it was designed to express. This means you loose functionality. For instance, you can't find all the definitions in a set of HTML documents by looking for all the dl elements. You will get all kinds of things that are not definitions. You might also miss a lot of definitions that were not created using the dl structure.

This highlights something that is clearly true about HTML, and that can potentially affect any document domain language. It can slip back into the media domain if people start using it for the media domain effects produced the default transformation algorithms rather than adhering to its document domain structures. 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.

Reversing the backsliding

One way to get back to writing in the document domain has emerged recently in the form of a new syntax called MarkDown. The idea behind MarkDown it to represent the major document domain structures of HTML using the kind of formatting people use in text-only email messages. This approach removes the difficulties associated with typing raw HTML. Here's a sample of Markdown (courtesy of Wikipedia):

```
Heading
=======

Sub-heading
-----------

### Another deeper heading

Paragraphs are separated
by a blank line.

Leave 2 spaces at the end of a line to do a
line break

Text attributes *italic*, **bold**,
`monospace`, ~~strikethrough~~ .

A [link](http://example.com).
[28]

Shopping list:

  * apples
  * oranges
  * pears

Numbered list:

  1. apples
  2. oranges
  3. pears

The rain---not the reign---in
Spain.
```

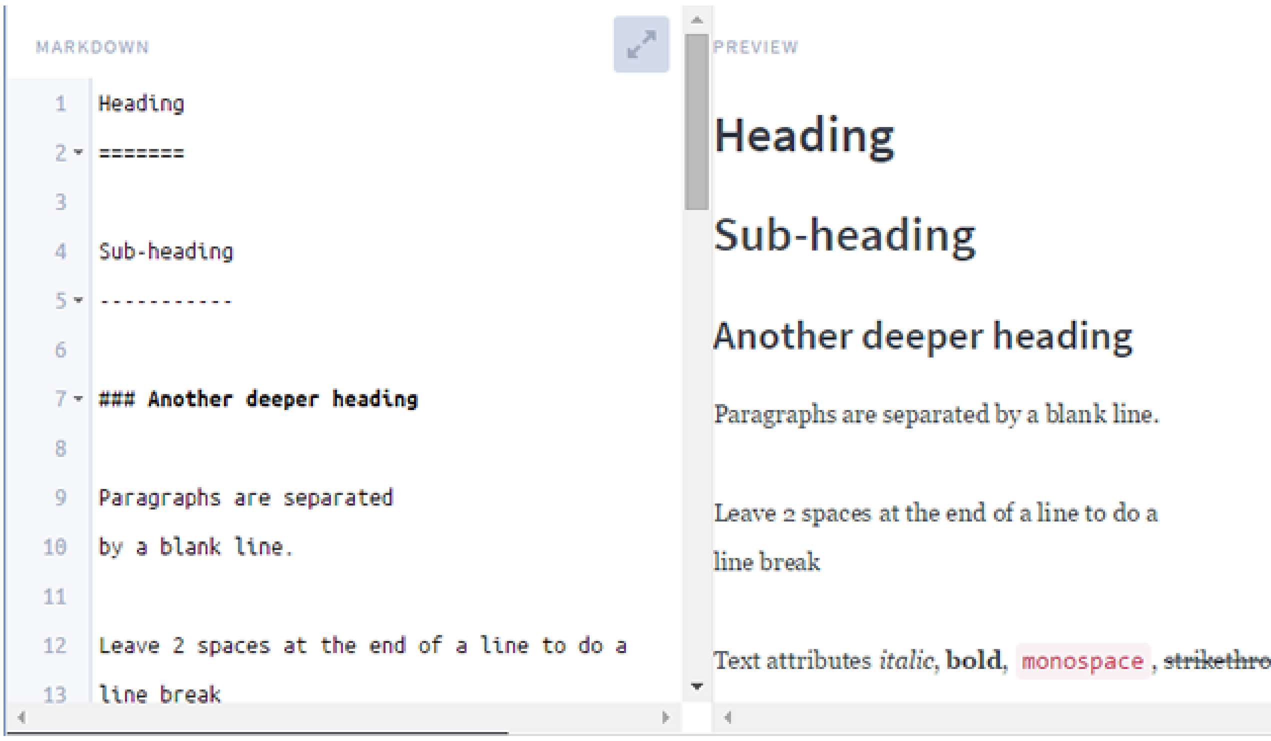
It translates into the following HTML (again, courtesy of Wikipedia):

```
<h1>Heading</h1>

<h2>Sub-heading</h2>

<h3>Another deeper heading</h3>

<p>Paragraphs are separated
by a blank line.</p>

<p>Leave 2 spaces at the end of a line to do a<br />
line break</p>

<p>Text attributes <em>italic</em>, <strong>bold</strong>,
<code>monospace</code>, <s>strikethrough</s>.</p>

<p>A <a href="http://example.com">link</a>.</p>

<p>Shopping list:</p>

<ul>
<li>apples</li>
<li>oranges</li>
<li>pears</li>
</ul>

<p>Numbered list:</p>

<ol>
<li>apples</li>
<li>oranges</li>
<li>pears</li>
</ol>

<p>The rain&mdash;not the
reign&mdash;in Spain.</p>
```

Markdown was not designed to be a pure document domain language. It was simply designed to let you write HTML quickly in a text editor. But the effect of using Markdown is that it gets you out of working in a WYSIWYG view and lets you see the structure of the document you are creating.

Many markdown editors use a split screen view that shows the formatted version in one pane as the writer writes markdown syntax in the other. But even here, the writer is still working in the document domain because they still see the structure in the view they are working on. A Markdown editor is never going to produce the kind of messy HTML that a pure WYSIWYG HTML editor can produce.

Figure 5.2. Markdown Editor



```
Heading
=======

Sub-heading
-----------

### Another deeper heading

Paragraphs are separated
by a blank line.

Leave 2 spaces at the end of a line to do a
line break
```

Heading

Sub-heading

Another deeper heading

Paragraphs are separated by a blank line.

Leave 2 spaces at the end of a line to do a
line break

Text attributes *italic*, **bold**, `monospace`, ~~strikethro~~

A detail of the Dillinger Markdown editor showing the Markdown and browser views side by side.

Something else interesting is at work here. A list in markdown is just a sequence of paragraphs that start with asterisk characters. On the face of it, this is just like a document domain editor creating lists by styling paragraphs. But if you look at the resulting HTML, you will see that it creates a proper list wrapper element around the list. The markdown interpreter is inferring the hierarchical structure of the document domain from the essentially flat Markdown syntax.

The author is working in something that looks and feels quite like the media domain, thought they have no actual styles and cannot change the formatting at all. But they use abstract formatting notation (underlines for headings, asterisks for unordered lists) to create document domain objects. The beauty of this is that the document domain constraints are preserved, while the author can work in a simple format that is easy to type, and reasonably easy to read.

This is an important reminder that XML and its applications are not the only route to structured writing. In fact, there are many other ways to create structured texts that obey the appropriate constraints for a particular use case. We will look at some of them in later chapters.

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 paragraph, lists, and titles. But these structures alone are not enough to hang meaningful and useful document domain constraints on, 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. 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 defined that arguably belong to the document or subject domains, such as 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 media domain styles don't really cover all the rules for creating bibliographies that your institution 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 explicit in the content in a machine-readable way. And if the constraints are not machine readable, 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 your want to control how bibliographic information is presented, and enable algorithms to find and extract data from bibliographic entries, you are going to need a document domain structure for a bibliography. That means you are either going to have to extend your document domain language to include on, or use a document domain language that already includes one.

One such language is DocBook. Here's an example:

```
<biblioentry id="bib.xsltrec">
  <abbrev id="bib.xsltrec.abbrev">REC-XSLT</abbrev>   <editor><firstname>James</
  <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 in a simpler notation that it easier for humans to read. (I've used this notation for earlier examples, and I'll talk more about it later.):

```
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 entires 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. Not only that, you could write an algorithm to extract bibliography information from a document by looking for biblioentry structures and selecting 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 author name structures.

This is another important way in which we can cut down the number of document domain structures we need. If we capture the individual pieces of information that make up a bibliography entry, we only need one bibliography entry structure even if we want to present bibliography entries differently in different publications (organize them differently, that is, as opposed to formatting them differently).

(What we are seeing here is a bit of a foretaste of the subject domain, for while bibliographies are a common document feature, regardless of the subject matter of the document, a bibliography itself is always about the same subject: books. So what is really going on when we model a bibliography entry this way is that we are abstracting out a document domain constraint by moving the content to a subject domain structure.)

Specialized document types

So far we have looked at moving individual elements of a document such as lists, graphics, and bibliographies into the document domain to introduce constraints on how they are structured and how they are formatted. But within the document domain, there are many distinct types of documents, each of which has its unique patterns and requirements. We can create multiple document types in the document domain for these different document types.

Some public markup languages support more than one document type. For instance, DocBook supports book and article, DITA supports concept, task, and reference document types, and SPFE provides a range of more specific document types for different purposes. And as each of these systems is extensible, there is the opportunity to add more types to meet your needs.

Some of these different document types are squarely in the document domain. For example, manual, quick-reference card, article, web-page, picture-book, novel, and catalog are all distinct document types that are distinguished by the kind of reading task they are used for, regardless of the subject matter. Thus a quick reference card can be a quick reference to any subject, a manual can be a manual for any product or service.

Many more document types are specific to certain subjects. A recipe is specific to the preparation of food. A telephone directory is specific to finding telephone numbers. A knitting pattern is specific to creating knitted fabrics.

Once we get into document types that are specific to a particular subject, however, we are starting to get into the subject domain.

Chapter 6. Writing in the Subject Domain

In our examination of the document domain we saw that while there are document types that are independent of any particular subject, such as manual, article, or report, there are also many document types that are specific to particular subjects. For instance, a recipe is a document type specific to preparing individual dishes.

You can write a recipe in the document domain. For instance, you could write it reStructuredText, 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.
```

However, there are specific constraints on the format of a recipe that this approach neither follows or records. To impose and record these constraints, you need a recipe document type.

A recipe follows a well known pattern. It has an introduction, a list of ingredients, and a set of preparation steps. The simplest form of a recipe document type might look something like this:

```
recipe: Hard Boiled Egg
    introduction:
        A hard boiled egg is simple and nutritious. Prep time, 15 minutes. Serv
    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 simple structure expresses the basic constraint of the well known recipe pattern and records that the author has followed it. Because these structures are specific to the subject matter, rather than specifying document structures or formatting, this markup is in the subject domain.

One of the common patterns of structured writing is the factoring 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 have titles been factored out here. Since we have structures specifically for `ingredients` and `preparation` we can factor out the actual titles and have the presentation algorithm add them in the transformation to the document domain.

If your organization publishes a lot of recipes, you probably have a lot more constraints on the content 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
```

Enforcing and recording these additional fields not only makes sure your recipes consistently meet your corporate constraints, they also offer 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.

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 an algorithm to construct the sentences “Prep time, 15 minutes. Serves 6.” from the `prep-time` and `serves` field values in the presentation algorithm.

The specification of constraints could get more detailed still. For instance, 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:

```
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. The following markup accomplishes the same thing, relying in part on the ability of an algorithm to break “2qt” into quantity and unit fields without the author having to do it explicitly:

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

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.

A key difference between subject and document domains

At this point we can see that with subject domain markup, we have a lot of choices about how documents are constructed. This highlights an important difference between document domain and subject domain markup. A document domain markup language specifies the content and order of a document. We expect that the document domain markup will specify exactly what content is to appear on the page and in what order. This is necessary in the document domain because the document domain does not record any information about the specific subject matter of individual pieces of text. We can’t write an algorithm to publish certain pieces of a document domain file, because the markup does not record which are which. (More on this in the next chapter.)

Once we introduce subject domain markup, however, that changes. With subject domain markup in place, we can write algorithms that select certain pieces of the content to display or not display. This means that the recorded content no longer specifies exactly what content is to appear on the eventual rendered page and in what order. Rather it is a collection of identifiable pieces of content that you can select from or reorder for publication.

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 says, display this text only in this publication. (This makes it management metadata, which we will talk about later.)

By contrast, this is how this might be handled 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. Nor does it contain the subheadings what would introduce either of them in the appropriate publication. This has a number of interesting consequences:

- There is an additional step required to publish this content. Any actual publication requires a specification of what content will appear where, so we need to create a document domain representation of this content for each publication we want to publish it in. Selecting the appropriate content for each document is the task of what I will call the synthesis algorithm.

- The document domain versions that we create for each publication don't need the conditional markup that is required when we write in the document domain. We apply conditions externally in the synthesis algorithm and create different document domain files for each publication.

- The author does not have to know anything about the mix of publications in which this recipe will appear. They just write a recipe and supply all the pieces of information they are asked for.

- If we add a new publication to our stable called *Family Dining* we can write a new synthesis rule to create recipes with both an alcoholic and non-alcoholic beverage suggestion. If our content was recorded in the document domain, we would have to go back through all the recipes and add new conditional markup to describe the content to appear in *Family Dining*. Having our content in the subject domain could thus save us major costs, and could make new forms of publication easier and more economically attractive. For instance, if we wanted to suppress some content for mobile publication, we could do so, without editing any of our existing content.

- The subject domain markup is specific to recipes. Only recipes have wine-match and beverage-match fields. By contrast, the document domain markup is much more general. It could be used to drive any publishing process that publishes the same content to multiple publications. This reduces the cost of developing the markup and the algorithms to process it. But it also means that authors have to know a lot more about how the publishing system works, and what is appropriate content for each publication. This reduces your pool of authors and makes authoring more difficult ant therefore more expensive. Finally, it means that it is much more expensive to maintain the content if you change publications.

As the above discussion illustrates, writing in the document domain and writing in the subject domain have very different cost and opportunity profiles. Writing in the document domain is usually cheaper to start with, but runs into maintenance cost and opportunity costs when your needs change. Writing in the subject domain incurs more up-front costs, but can save you money and create more opportunities down the road.

In the document domain chapter, we also noted that document types are not universal. Some work better in one media, and some in another. While content recorded in the document domain can be formatted for any media, it is not necessarily structured perfectly for every media. Different media have different properties, such as behavior and navigation, which demand a different approach to document structure and even the way content is written. For hypertext environments, where readers are likely to arrive at a page by search and navigate onwards by links, a page is a hub, not a leaf, and this can make a big difference to both writing style, document structure, and the subject matter you include inline (vs. linking to it).

If we want to create different document structures for different media, recording our content in the subject domain gives us that flexibility.

Using subjects to establish context

In the discussion of 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.

Suppose we have a markup language for recipes:

```
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
```

If we want to have a separate formatting for the list of steps in a recipe procedure, we can do so without having to create a separate recipe-preparation-list type. The list of steps here is in an ordinary ordered-list structure, but that structure is the child of a `preparation` element which is the child of a `recipe` element. We can write a rule that creates special formatting just for ordered lists that are the children of `preparation` elements that are children of `recipe` elements.

Future proofing

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 possible 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 your 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 for content, 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. Thus the subject domain markup has future proofed your content against this change of layout. The document domain reStructuredText markup specified the use of a table, which is not a truth about the subject matter, but a decision about layout 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. 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.

Simplicity and Clarity

One of the biggest benefits of subject domain markup for authors is a much higher degree of simplicity and clarity compared with a typical document domain language.

While a general document domain language like DocBook needs to have structures for a wide range of document structures, a recipe markup language such as we have developed in this chapter, has only a few simple elements. Better still, there are very few permutations of those elements. Because subject domain languages do not specify document order, we don’t need to allow for many possible document orderings in the language, thus reducing the permutations we have to allow for and deal with. The synthesis algorithm can take the named structures of the subject domain markup and order them in any way you like.

Because subject domain structure describe the subject matter they contain, they are also much clearer to authors, 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 authors 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 author 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. A general document domain language can be used to write all kinds of documents. It will not enforce or record nearly as many constraints, or enable nearly as many options for validation or publishing, and it won't be nearly as clear and simple for authors to use. But neither will it require you to create subject domain languages for each of the subjects you write about. At first glance, that may seem like a slam dunk case for sticking with the document domain, as the idea of inventing subject domain languages and the synthesis and presentation algorithms to go with them may seem daunting. But as we will see, the decision is not so clear cut, as sticking with he document domain comes with a lot of complexity, and sometimes custom development, that may not be apparent at first.

Using available subject domain languages

However, moving to the subject domain does not necessarily mean having to develop all your subject domain structures and their accompanying synthesis and presentation algorithms yourself. There are existing subject domain languages already in existence in many fields. For instance, there are at least three recipe markup languages out there, REML, RecipeML, and CookML.

Some of the most commonly used subject domain languages are those for documenting software APIs. These include Sphinx (Python), JavaDoc (Java), and Doxygen (multiple languages). We will look at some of these in more detail later.

Wikipedia contains a long list of XML markup languages[https://en.wikipedia.org/wiki/List_of_XML_markup_languages] (though note that not all subject domain languages use XML). Many of these are subject domain languages. Does this mean that if a subject domain language already exists for your subject matter that you should use it rather than developing your own? Not necessarily. It is pretty much a universal rule of markup languages (and a lot of other things) that the more needs they try to serve, the more complex they become. If you take a look at REML and CookML you will see that both are more complex than the recipe markup language we developed here.

The upside of these existing languages is that they may have algorithms and systems associated with them that you can take advantage of. However, they are often more complex that you need, sometimes less clear, and often don't enforce or record all the constraints you want for your business.

You don't have to adopt these languages directly, however, to take advantage of the systems that use them. Since all recipe languages describe the same subject matter, it is easy to transform content from one to another where they have equivalent structures. This allows you to create your own language to maximize simplicity and clarity for your authors, and to enforce and record all the constraints that are important to you, and still take advantage of existing functionality with a simple transformation.

The same is true of existing document domain languages. If you have your own subject domain language, you do not have to create an entire publishing chain for it. You only need to create an algorithm to transform it to an existing document domain language that has the publishing features you need.

Limits of Subject Domain Markup

While all content is specific to its subject matter, not all content breaks down into such easily identifiable fields as a recipe. A generic essay document format fits equally well for an essay on radishes as an essay on asteroids. Subject-specific structures are much easier to discern for reference works. The format of a telephone directory, an API reference, or a parts list is always specific to the subject matter. In fact, we find that the format of an API reference for one programming language can be different from the format of another language because of differences between the languages.

On the other hand, there are many document types that today are typically written as unstructured text. The content could be separated into distinct fields, as we separated prep time and number served into distinct fields in our recipe example, but is instead presented ad hoc as part of general text. Developing and applying subject domain templates the pull out this content into distinct and accessible fields can serve to make the content more consistent, more accessible to the reader, and more adaptable for various publishing scenarios. We will look at some examples in later chapters.

This only scratches the surface of what you can do when you record your content in the subject domain. In future chapters, we will look at the major algorithms of structured content, the benefits the provide, and how the work with content in each of the domains.

Chapter 7. 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.

When you manage unstructured content, the content file contains just the content. Any management metadata related to the content is contained in the repository, whether that be a simple file systems or a more complex asset management system. (People may also keep separate logs of management metadata, in spreadsheets, for example.)

With structured content, this separation of the content from the management metadata is not always maintained. Sometimes management metadata finds its way into the content file itself. Why? Because structured writing makes it possible to address, and therefore manage, individual content structures within a file. Sometimes you can do that management based on media, document, or subject domain metadata. Sometimes you need (or choose) to insert management metadata into the content file.

Example: Including boilerplate content

For example, lets say you have a standard warning statement that you are required to include in a document wherever you have a dangerous procedure. Structured writing is all about 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)
    step: Plant dynamite.
    step: Insert detonator.
    step: Run away.
    step: Press the big red button.
```

In the markup above (yes, I will explain it eventually) the `>>>` is an insert command. It inserts the content of the file located at `files/shared/admonitions/danger`.

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 insertion 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. This is one of the hallmarks of management domain markup. It is imperative and it is addressed to a particular software system.

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.

An alternative approach in the subject domain

There is another way to handle this situation, this time using the subject domain. As we saw in the previous chapter, factoring out invariant text is a feature of the subject domain. 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). 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 rather than copying and pasting. It leaves it entirely up to the author to remember and enforce the rule about dangerous procedures.

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 or not a procedure is dangerous:

```
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. This retains 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 a procedure has `is-it-dangerous` 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 our tagging language for procedures. Now the writer is forced to answer the question “yes” or “no” for every procedure they write.

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 were it is located. They are recording a fact, not giving an instruction.

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, your are going to limit your pool of authors to a few highly trained individuals, and even then they are still likely to miss some instances. The cost of ensuring full compliance with all these rules without subject domain markup to enforce the constraints could be quite high.

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 elements for describing user interface elements.

In some cases, such languages can mix subject domain elements 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
```

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 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.

Ad hoc management decisions

So far we have contrasted management domain and subject domain approaches to handling invariant rules. Sometimes the management decisions are being made ad hoc by writers as they write, not based on any invariant properties of the subject matter or document structure, and if the decisions affect only part of the content in a file, then the only way to record those decisions so that the publishing algorithms can act on them is to include management domain metadata in the content.

An example of this is content reuse. The safety warning example was a case of an invariant rule for including a fixed piece of content, which was, therefore, being reused. But there are other situations in which the same text, or substantially the same text, may appear in different places for ad hoc reasons, or for reasons where any rule would apply to so few cases that it would not be worth defining and modeling.

The obvious and traditional way of handling such cases is either to write the text again (if you are not aware that other instances exist, of where they are) or to cut and paste the text. The downside of this is that the text now exists in multiple places, which creates management headaches if you ever need to change it. It also costs more to research and write the same content over and over again.

If you do ad hoc reuse, by using some form of management domain include instruction, you partially solve these management problem because you can change the content in one place and it will automatically appear in its changed form the next time any of the documents that reuse it are republished. The content does not have to be researched, written, or edited multiple times.

I say “partially solves”, because when reuse is not based on an invariant rule (and even when it is, if that rule is not encapsulated in subject domain markup), then it is up to the author to discover if a piece of content they are about to write already exists, locate it, and use it. The more potentially reusable content you have, the more difficult this search becomes, because there is more to search through. And unless the author remembers what all the pieces of reusable content are, they constantly have to ask themselves if the content they are about to write has already been written. This means they will end up searching for reusable content even when there isn’t any, but that search also takes time. Mistakes and omissions are inevitable, which means that when changes occur, you still have to search for duplicate instances of the content, and the time saved by not researching and writing the content again can be chewed up in all the processes around reusing content.

Successful reuse strategies, therefore, require a high level of discipline and management, whether or not that is provided by subject domain markup.

The management domain is used for other things besides reuse. We will look at some of these later in the book when we consider the various algorithms and structures found in structured writing.

The management domain and content management

So far I have talked about the management domain as in intrusion into structured writing. But it is worth looking at where that intrusion comes from. Management domain structures in structured writing are in intrusion from the content management process.

So far in this book I have talked about structured writing as a way to make content better. In other words, I have talked about it having a fundamentally rhetorical purpose: it is there to improve the text. And since we now live in a world of dynamic content delivery options, it is there to help us build better rhetorical structures that might be difficult to build and manage by hand.

The world of content management, on the other hand, exists mainly to manage the vast collection of content that many organizations own today, and the process by which it is created. Content management is a significant business problem quite apart from the rhetorical properties of the content. Many organizations adopt structured writing not for any rhetorical purpose, but as an enabler of content management processes.

Many content management processes involve finding, identifying, combining and publishing content from diverse sources. These processes often require that content meets certain constraints, which means that structured writing -- the application of constraints to writing -- is a useful content management tool.

Tools and processes that are designed primarily to tackle the content management problem, rather than the rhetorical problem, almost invariably use document domain structures with a significant injection of the management domain, though in these cases is might be more accurate to say that the document domain structures are being used as an extension of the management domain.

The focus of this book is the use of structured writing as a tool for rhetoric -- for making content better. As such, it will talk a lot more about the subject domain than would a text focused on content management. Nonetheless, as shown briefly here, the subject domain can have powerful content management features as well. I will look more at this in later chapters.

Part II. Algorithms

Table of Contents

Chapter 8. Quality in Structured Writing

When I talk to programmers about what I do, they often ask my 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 comprehension[http://www.technologyreview.com/featuredstory/540001/teaching-machines-to-understand-us/] and speech recognition[http://www.technologyreview.com/news/544651/baidus-deep-learning-system-rivals-people-at-speech-recognition/]).

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 language to match, to teach computers to understand relationships that humans would express in ordinary language. Content management systems have implemented metadata schemes, often involving elaborate taxonomies, in an attempt to make content more findable when regular text-based searches don't work as well as they would like.

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 and understand it with the same level of sophistication (or if the machine's understanding is actually more sophisticated than the human's) 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 computers that could read and understand like humans do. Semantic technologies are about dumbing it 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 would be daunting, for instance. 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 semantics to the machine, and which you choose depends on what specific functions you want to perform.

But if the machine can 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 are rapidly learning to read human text better than most humans, 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. They leave out stuff that needs to be said, or they say them in a way that is hard to understand.

We all suffer from a malady called The Curse of Knowledge 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.[http://everypageispageone.com/2015/08/04/the-economy-of-language-or-why-we-argue-about-words/] Laboriously stating the obvious is as much a writing fault as omitting the necessary. Yet what is obvious to one reader is necessary to another. The curse of knowledge is that as soon as something becomes obvious to us, we can no longer imagine it being necessary 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 computers is that is forces us to be very careful in how we create and structure data for machines to act on. We quickly hit on the phrase "garbage in, garbage out", because the machines we were talking to were 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 at 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 it 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 they are difficult to remember and to audit, meaning there is little feedback for an author who strays. Also, the long form content typical of the paper era did not lend itself to obvious auditable pattern. The short form content more prevalent in the Web era more naturally falls into repeatable and auditable patterns which we can express through structured writing.

Structure and quality

Structured writing provides a way to both guide and audit content for quality. While you don't need computers to define a structure for content, paper-based processes have always had to be build around the publishing process, and thus largely stayed in the media domain. But most of the valuable structure that guides and audits writers writing about a specific subject for a specific audience lies in the subject domain. Without computers capable of processing subject domain markup into publishable media domain markup, the ability to apply structured writing to the problem of quality was limited.

My reply to the people who ask me whether structured writing is relevant, therefore, is "garbage in, garbage out". Structured writing is not about making content readable by machines, it is about making content better. Making content readable by machines is something we do so that we can use the machine to help us make the content better.

Structure, art, and science

To many writers, this 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 all of Western music is based.

Computer programming is widely regarded as an art[http://ruben.verborgh.org/blog/2013/02/21/programming-is-an-art/] 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 literary 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 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.

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, 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 to to facilitate the operation of a content management system or make it easier to reuse content. In some cases, these systems actively interfere with the author's art and directly hinder the production of high quality content.

In the opening chapter, I noted that structure exists to serve a particular purpose:

> [A] piece of structured content is structured for a particular purpose that you thought of at the time you created it. The content is structured for that purpose or set of purposes you thought of, but is unstructured for other purposes. Just as a hat can be the right size for Tom and the wrong size for Harry, a piece of content can be structured for Mary and unstructured for Jane. It all depends on context.

Writers who have had bad experiences with structured content have usually been faced with structures that were not designed for the writer's purpose. But such content is not merely unstructured for these author's purposes, it is actually contra-structured. It has an enforced structured 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 think they are nearly 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 Paradox[http://www.theatlantic.com/business/archive/2016/01/automation-paradox/424437/], automation does not decimate white collar jobs the way we have been told to 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.

Chapter 9. Algorithms: Separating Content from Formatting

If there is one phrase that most people associate with structured writing, it is “separating content from formatting”. We structure content so that we can process it with algorithms. An algorithm is a repeatable method for doing something. If we want our processing algorithms to be repeatable, we need the structure of the content to be consistent. That means that the separation of content from formatting has to be done consistently and repeatably. That makes it the first algorithm of structured writing.

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 represents 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 this or other 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, not the content:

```
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.

So, it turns out that 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 content.

```
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 separating content from formatting, because the content actually still contains formatting information in the form of a style 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.

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}[tab]Sand
{style: list-item}[tab]Eggs
{style: list-item}[tab]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 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}[tab]Sand
{style: unordered-list-item}[tab]Eggs
{style: unordered-list-item}[tab]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 separation. 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

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 introduce 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 CSS[https://css-tricks.com/numbering-in-style/].)

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` element 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 the difference based on which type of list it belongs to. The element name “list-item” describes it role in the document (within its context 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 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 level 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 is it perfectly possible and legal in there 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.

Furthermore, writers understand that the higher the number of the heading element (`H1`-`H6`) the larger the font will appear in the output. They don’t know which font it will be or how big it will be, but larger/smaller is still a formatting distinction. Formatting has not been completely separated from content.

By contrast, in DocBook, we have a `section` element. 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. By creating a `section`(structure (DocBook) element, DocBook 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 element 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.

(Now it must be said that not everyone necessarily holds to the view that heading 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 involved separating out some of the content as well. Or rather, that it involved separating out some of the characters. In other words, the distinction between 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
```

But what if you have hundreds of addresses, all with the same labels. In this case, are the labels really content, or are they format? 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
```

Now, of course, we have moved our content into the subject domain. Notice that in our previous attempts at separation we separated out the formatting of a list and replaced it with an abstract form of a list. This certainly separated out some of the formatting, but it still left us with a list. And deciding to list items in a document is still a formatting decision.

Moving content from the media domain to the document domain actually separates out the style information, but still leaves a lot of general formatting decisions, such as the decision to use lists or tables, firmly attached to the content. Separating style from content is useful in a number of ways, but it does not achieve a complete separation for content from formatting.

In the subject domain example, however, we moved our content away from being a labeled list and turned it into an subject-specific record. We could how have an algorithm turn that record into several document structures. This algorithm, which I will call 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 entirely separated from formatting, we also gain the ability to query and reorganize the content in various interesting and useful ways (which we will explore in further 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. Algorithms: Processing Structured Text

If structured writing is about separating content from formatting, then it is also about reuniting content and formatting in order to publish content to a specific device. To do this, we process the structured text using algorithms.

Since part of the point of structured writing it to be able to publish in different combinations to different media, so we will need different algorithms to create different combinations and target different media. Understanding the basics of these algorithms is important to understanding structured writing even if you don’t intend to code the algorithms yourself.

Two into one: reversing the factoring out of invariants

Moving content from the media domain to the document domain and 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.

Thus 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 cases 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 the structures.

We are not concerned at this level 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 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 what we need an algorithm to do.

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 only real difference is that we have factored out the details of how the structures are recognized. (Yes, code and pseudocode are examples of structured writing at work!)

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 style sheets, you understand a good deal of how structured writing algorithms work.

One important thing to notice is that when you create a style sheet 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 style sheet 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 the rules apply to 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 (which kind of sits on the boundary between the media domain and the document domain). 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 in parentheses immediately after it. So the annotation says that the words “Rio Bravo” refer to a movie. (I really will explain this markup eventually.) 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 style sheet. (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 can 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)</title>
```

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 structure 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 in 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 of unordered lists. Because `unordered-list` is just a container and does not produce any formatted output, its rule has nothing to do:

```
match unordered-list
    continue

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

Applying these 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 algorithm (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. 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. (The 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.) We will look at algorithms that use the query-based approach in later chapters.

Chapter 11. The Single Sourcing Algorithm

Single sourcing was one of the earliest motivations for structured writing. However, the 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. To make life easier, I will distinguish three main meanings of “single sourcing” as follows:

Single sourcing	Producing the same document in different media.
Content reuse	Using the same content to create different documents.
Single source of truth	Ensuring that each piece of information is recorded only once.

In this article we will look at single sourcing as defined above.

Basic single sourcing

The basic single sourcing algorithm is straightforward and we have covered most of it already in the discussion of basic content processing.

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, using the syntax that I have been using throughout this book. (The block of text set of by blank lines is implicitly a paragraph -- a structure named `p`.)

```
page:
    title: Hard Boiled Eggs

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

    section: Ingredients
        ul:
            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. (Though there may be complications with something called namespaces, which we will discuss later.)

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

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

    <h1>Hard Boiled Eggs</h1>

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

    <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 problem, such as where to break each line of text, 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. It also creates issues with references. For instance, a reference to content on a particular page cannot be known until the pages are paginated by the algorithm.

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). 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. (I call this the encoding algorithm.)

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>'
```

Other typesetting systems you can use for print output include TeX and later versions of CSS.

Differential single sourcing

Basic single sourcing outputs the same document 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
```

In the markup language I am using here (and will eventually explain) 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 URLs so that reader can type them 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 ou
</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 content written as it is, though, there is no easy way to produce this preferred form for paper. While the content is in the document domain, 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 representation 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 formal way that makes the clarification available to algorithms. Because the algorithm itself has access to the clarification, it can produce either kind of clarifying content for the reader by producing either document domain representation.

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 the chapter on linking????.

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 now 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 online 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.

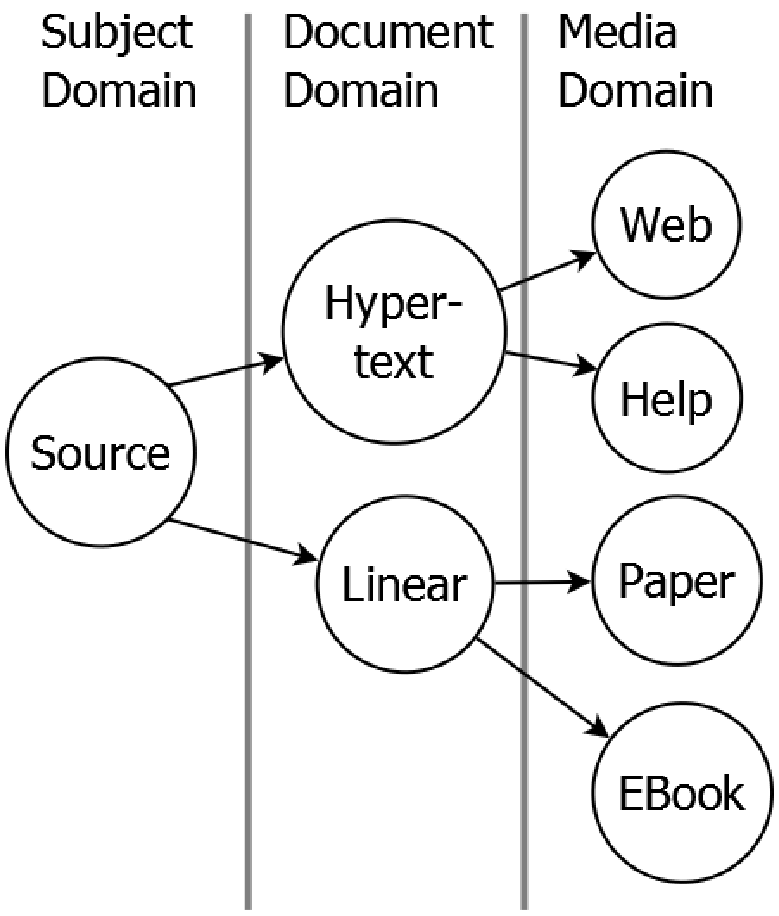
Single sourcing your timetable to print and PDF will not produce the kind of online presentation of your schedule that people expect, and that can have a direct impact on your business.

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 (which I will call the synthesis 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.

Differences in linking between media can also go much deeper than how the links are presented. Links are not simply a piece of formatting like bold or italics. Links connect pieces of content together. On paper, documents are designed linearly, with one section or chapter after another. But online you can organize information into a hypertext[http://everypageispageone.com/2013/03/19/web-organization-is-not-like-book-organization/] with links that allow the reader to navigate and read in many different sequences.

The difference between linear information design and hypertext information design is not a media domain distinction but a document domain distinction. But if you are thinking about single sourcing your content it is one that you have to take into consideration. In other words, single sourcing is not just about one document domain source with many media domain outputs. It can also be about a single subject domain source with multiple document domain outputs expressing different information designs, and outputting them to different media.



More radical forms of differential single sourcing start to look a lot like reusing the same content to build quite different documents (albeit on the same subject) and therefore start to use the techniques of content reuse, which we will deal with in the next chapter.

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 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 approach a 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 an 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.

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 group 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.

Manual-oriented tools such as FrameMaker start with the manual format and then break it down into topics for the help system (usually by means of a third party tool). The results are often poorly structured help topics. For instance, it is common to see the introduction to a chapter transformed into a stand alone topic that conveys no useful help information at all.

Help authoring tools start with help topics and then build them up into manuals, which they may do either by stringing them together linearly, or mapping them into a hierarchy via a map or a table of contents. While help authoring tools should nominally optimize for help and then do the best they can for manuals, users of help authoring tools often focus on the manual format more than the help, so the use of a HAT does not guarantee that the help format gets design priority. The same is true of topic-oriented document domain systems like DITA. They are often still used to produce document-oriented manuals and help systems, with the topics being mostly used as building blocks.

Changing your information design practices from linear paper based designs to hypertext Every Page is Page One designs is non-trivial, but such designs better suit the way many people use and access content today. 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.

Responsive Design

Responsive design is a form of differential single sourcing. Many attempts at responsive design are not as responsive as we would like them to be because they content is not stored in a source format that allows a sufficient degree of differential presentation.

Chapter 12. The Reuse Algorithm

The reuse of content in different contexts has become one of the main drivers of structured writing, particularly in the form of widespread adoption of DITA. Content reuse is not one technique, but a collection of many techniques, each of which requires specific content structures in the subject and document domains.

The simplest content reuse technique is cutting and pasting content from one source to another. This approach is quick and easy to implement, but it creates a host of management problems. When people talk about content reuse they generally mean any and every means of reusing content other than cutting and pasting.

This means that reusing content really means storing a piece of content in one place and inserting it into more than one publication by reference. “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.

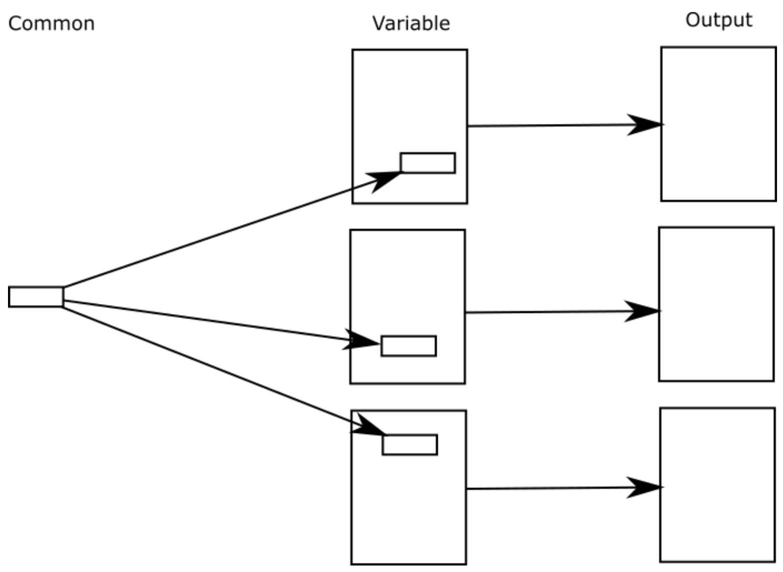
If you cut and paste, this is not a concern. You can cut any text you like, paste it in anywhere, and edit it to fit if need be. But if the content you want to use is used in other places, you can’t edit it to fit because that might cause it to no longer fit in the other places. For reuse to work, the 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.

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.



For instance, if you are writing procedures and there is a common safety warning that must appear on all dangerous procedures, each individual procedure is the variable part and the standard warning is the common part.

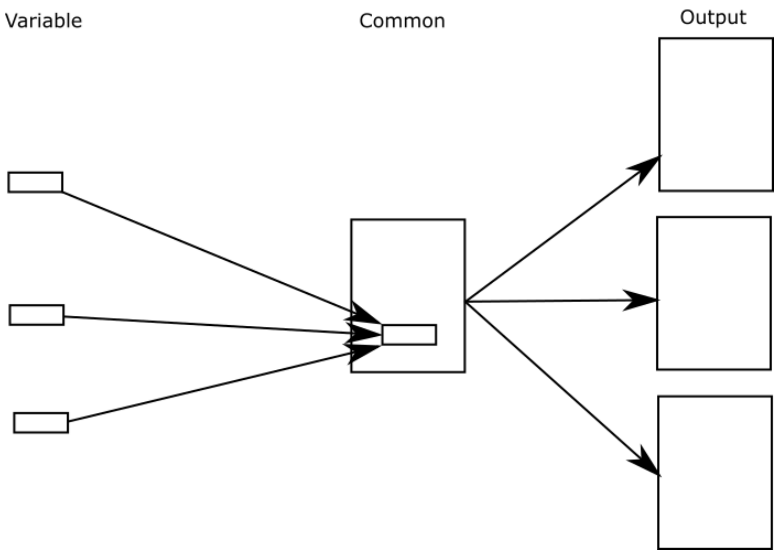
We looked at an example of this in the chapter on the management domain.

```
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 division of responsibilities between the common content and each of the documents it will be inserted into. The inserted content should give the safety warning, the whole safety warning, and nothing but the safety warning. Each document that includes it should include it in the required place in the procedure structure.

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. Lets 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 with variable where variable = $doors
    $number_of_doors = vehicles/vehicle[$model]/doors
    output $number_of_doors
```

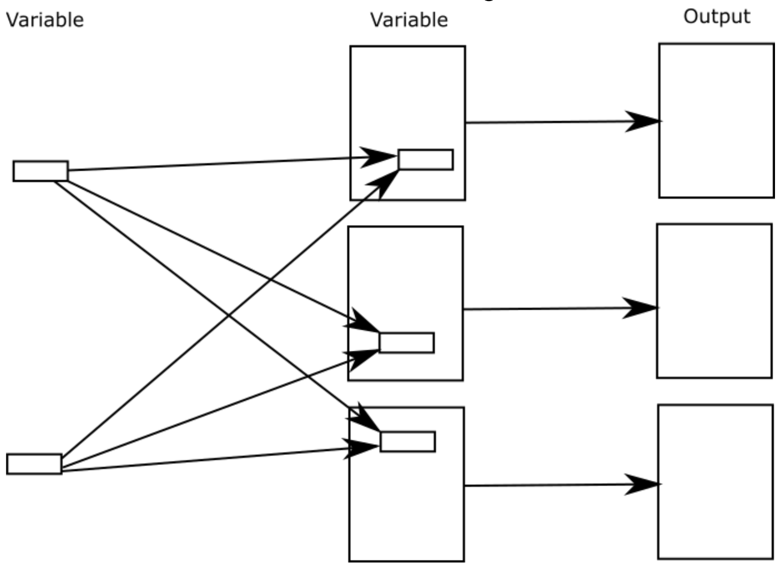
All these insert and query mechanisms are pseudocode, of course. Exactly how things work and exactly how you delineate, identify, and insert content vary from system to system.

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 this approach is fragile. You can't reorganize your files without breaking reuse, and you almost always need to reorganize your files eventually. Plus, it forces you to have every piece of reusable content in a separate file, which may not be efficient.

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.

The most basic way to factor out the file name is to give the content of the file an ID and use the ID to identify the content in a location independent way. 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 means that the synthesis algorithm 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 synthesis 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 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.

Another way to do this is with keys. Instead of assigning an ID directly to the content, the keys approach use an intermediate lookup table to resolve keys to particular resources.

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 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 them 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 use to identify resources that do not have IDs, which may include resources that you do not control.

On the other hand, we should note that any ID can be treated as a key simply by locating it in a lookup file. In fact, that is all a key really is: an ID that is located in a lookup file instead of directly in a resource. As such, it is perfectly possible to create a system where some IDs are located directly in the content and some are located in lookup tables. All that is required for this to work is for the synthesis algorithm to recognize when an ID is located in a lookup table and load the resource it points to. Nonetheless systems that support keys tend to implement them as separate structures from IDs.

One downside of a strictly external approach to keys is that 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:

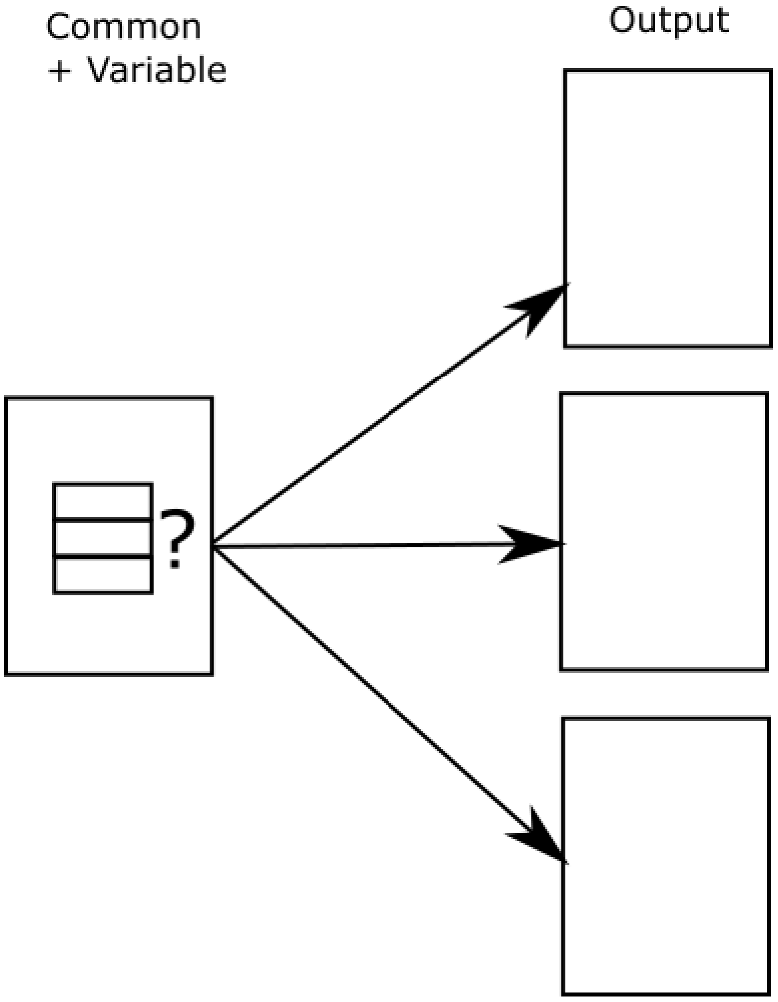
```
keys:
    key:
        name: warn_danger
        resource: files/shared/warnings#warn_danger_domestic
```

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

```
keys:
    key:
        name: warn_danger
        resource: files/shared/warnings#warn_danger_foreign
```

Common with conditions

In some cases of variable into common, the variant pieces will not actually be 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 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. These are the kinds of cases where 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. They 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.

Factor out the common

In the chapter on the management domain????, 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. In effect, this factors out the common content as well.

```
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
```

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 itself 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.

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 -- one to which they should already know the answer.

Structured writing is about factoring invariants and complexities out of content and this approach enables the widest range of reuse possibilities while factoring all the complexities of reuse out of the content.

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:

ol:
    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 then 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

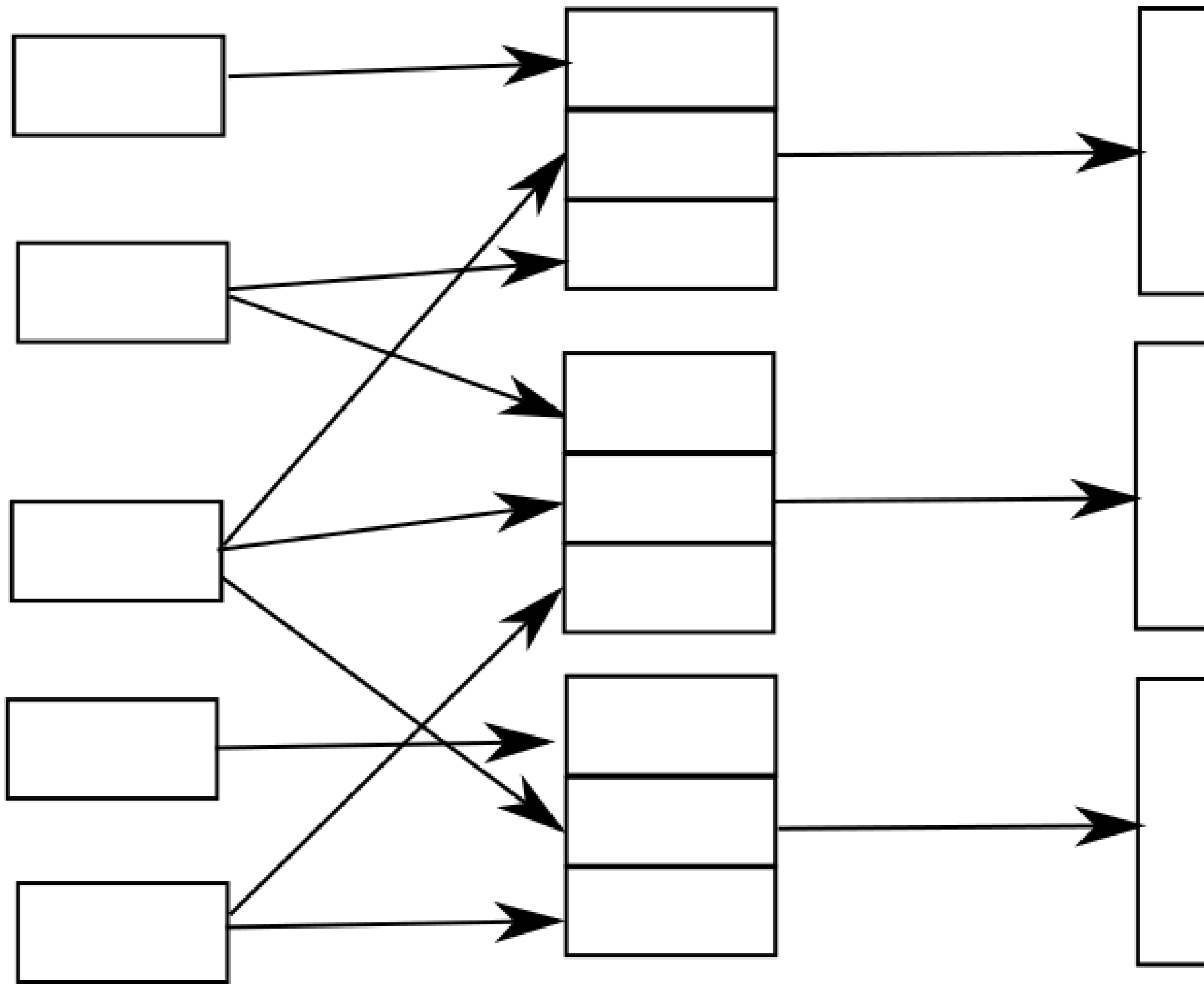
```
keys:
    key:
        name: %warn_danger
        resource: from vehicles select features where model = $model
```

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 these are implementation details.

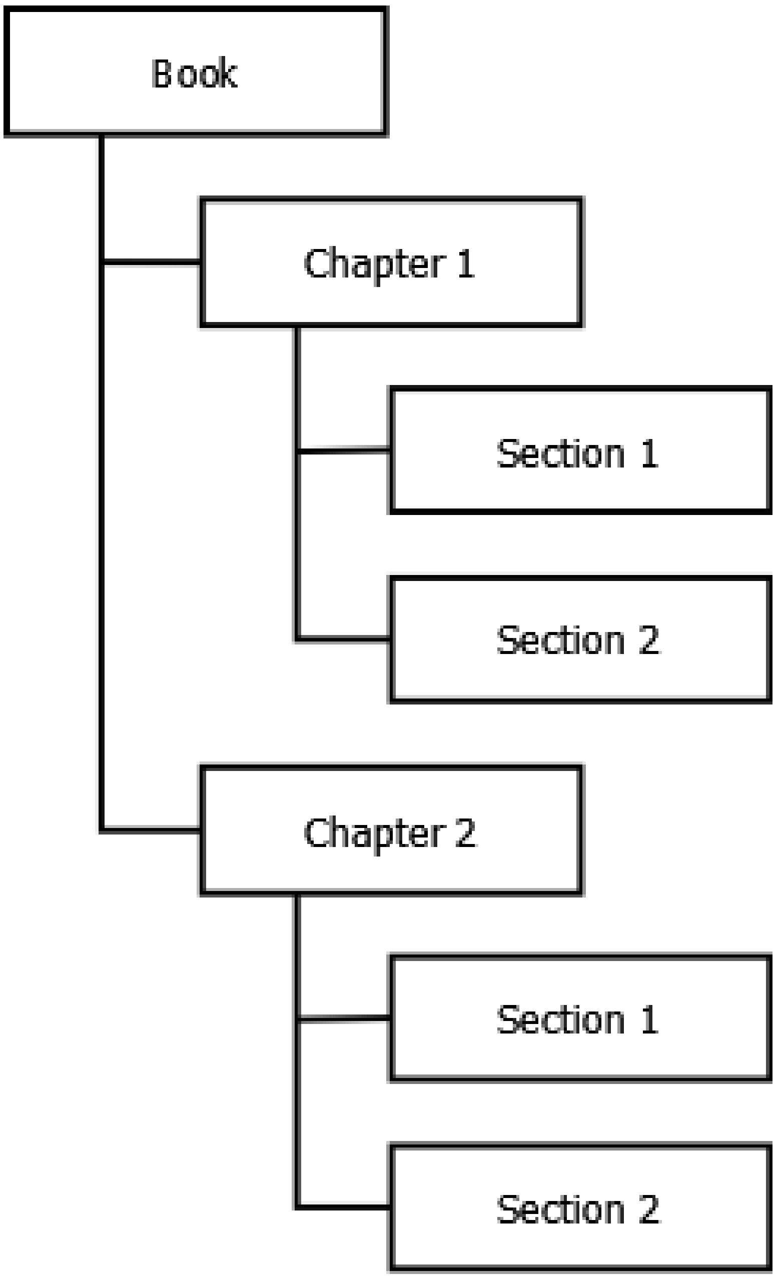
Assemble from pieces

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
```

It is important to note here that a map does not always have to be written by hand by an author. In some cases the map may be created by an algorithm based on the metadata of the units themselves. Whether this is possible depends on what determines the desired order of units. If the assembled units are supposed to form a narrative flow, they may need to be ordered by hand as it is difficult to generate a narrative ordering from metadata. But if the units do not form a narrative flow, they can easily be ordered using metadata. For instance, if you are assembling a cookbook from recipe units you might order them by season or main ingredient without caring if the boiled egg recipe comes before or after the scrambled egg recipe.

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. Foll

    >>>(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.

When using the map approach, it is important to think about how your content will single source to paper-like media and hypertext-media. Some people will output the same map to to both media. In hyper-text media that usually results in the map being turned into a table of contents, often displayed in a separate pane as in a help system. This may be fine if you are creating a help system, but it is not how Web content is usually displayed. Another approach to single sourcing content that uses the assemble from pieces approach is to create completely separate maps -- or even to use completely different assembly techniques that don’t involve maps at all -- to produce paper-like and hypertext outputs. This can help you to work around some of the design limitations that we talked about in the single sourcing chapter.

Combining multiple techniques

Of course, 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 the

    >>>(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.

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.

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.

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 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.

Where costs can really mount, though, it when it comes to modifying the content when the subject matter changes. 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.

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.

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. Taking a disciplined approach to reuse from the beginning is essential to avoid problems down the road.

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. As the size of the content set grows, it can take longer and longer to determine if reusable content exists and to find and reuse it. It is possible for this to end up costing more time than was saved by not rewriting content. (Reuse techniques that factor out the reuse from the author's work avoid this problem.)

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.

Chapter 13. Single source of truth

The third form of single sourcing is the single source of truth. This means that when a piece of information is needed anywhere in your organization, there is one and only one place where that information is stored and from which it should be accessed.

This is, of course, a case of content reuse, but whereas the content reuse algorithm deals with how reuse happens, the single source of truth algorithm is concerned with ensuring that only one copy of a given truth is created and maintained. The content reuse algorithm can work without any such assurance, and frequently does. The writer finds some content to reuse and reuses it, without asking if it represents the one and only source of the truth it expresses.

The task of finding some content that meets your needs and can be reused is very different from that ask 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 of the content doing the reusing. Ensuring that there is only one source for that content is a task for the writer of the content being reused. They have to make sure when they create and store the content that there really is no other piece of content serving the same purpose in then entire body of content.

The problem with the single source of truth algorithm is, as Pontius Pilate asked, "What is truth?"[John 18:38]

For some types of content, this is an easy question 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. (Stating 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 story.)

For other types of information, however, the question becomes much more complex. There are two fundamental problems.

1. Content, by its nature, deals with those subject that do not fit neatly into rows and columns, and thus cannot be formally normalized according to database rules. Where databases describe 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. The same fact may be mentioned in many different pieces of content for many different purposes. It may be elaborated on in one place, explained briefly in another, and merely mentioned in a third. What is the single source of this fact? Can it be effectively factored out and stored separately when it is described (legitimately) in different levels of detail in different documents?

2. 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. What then is the single source of truth for this information?

It comes down to this: there is a difference between a truth and an expression of a truth. You can have one source of a truth, but you often require different expressions of that truth for different purposes and different people.

In the database world, you can store the customer's birthday in a single field of a single database, but you will produce may different expressions of this truth for use by different functions in your organization and for presentation to this or other customers in different forms.

The way we achieve the storage of a single source of a truth is by abstracting it out of all of the expressions of that truth. Abstracting variants from invariants is, of course, what structured writing is about. But content is, per se, not a truth but an expression of a truth (or of several truths and their relationships).

A simple case of the single source of truth algorithm 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
```

There are a couple of reasons for doing this. One is that company names and product names often then to have a precise formal variant that the marketing department wants everyone to use, and the informal variant 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 ensures that the correct version of the name is used at all times.

The second reason is that that product 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.

This is clearly a very tactical approach to creating a single source of truth. 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. We don't for example, have a variable for the name of every ingredient in a recipe:

```
ingredients:: ingredient, quantity, measure
    >($carrots), >($5), >($lb)
```

We don't do this because there is no abstraction here. Carrots are carrots, 5 is 5, and pounds are pounds. Whereas, the company name is not company name, but "Acme Corporation".

When we talk about single source of truth in regard to content, therefore, we are not (or should not be) talking about designating a single piece of content as the sole truth on a particular subject. We are (or should be) talking about abstracting out key information from various pieces of content to store separately.

Single vs Findable

In many cases, the desire to create a single source of truth is to manage change in a content set. Since facts may be scattered across a content set in all kinds of content designed for all kinds of purposes for all kinds of audiences, it can be time consuming and expensive to find and change all of these instances when the subject matter changes. The biggest challenge, perhaps, is making sure that you have found all the instances in which the fact is mentioned, which can be particularly challenging when information is scattered across multiple formats and systems.

Factoring out that information into a single source of truth should, in theory, make the whole update process much easier. Lots of organization use variables to insert product names into documents, for instance, so that they can just update the value of the variable when the product name changes. However, there are a number of problems with this approach:

Historical usage	Sometimes there is historical usage in a document: a reference to past products or services that remain true after the contemporary changes are made. Not ever instance of a product name, for instance, should change when the current product name changes. But it is often hard to make the distinction between historical and contemporary use at the time the variable is inserted into the content.

Validation	It is hard to make sure that the variable is always used by writers. There is generally no way to structurally remind or constrain the author to use the variable instead of the name. Automated checking systems can sometimes catch instances, but they can be confused by historical usage or by other facts that happen to be expressed with the same words. (Think of a company like Henry's Camera, for instance, who cannot guarantee that every use of Henry's is a reference to the company name.
Anticipating change	Finally, there is the problem of which facts to factor out and which not to. Factoring out every fact would be prohibitively expensive and make authoring virtually impossible. You can only afford to factor out those facts are are likely to change. But none of us has a crystal ball, and some of the biggest changes we have to deal with often come from places we did not expect.

For all these reasons, factoring out facts into a single source of truth is a limited proposition. It can be highly useful if we apply it in the right way to the right facts, but it can be cumbersome and error prone is we try to take it too far. Thus we are certainly going to end up with all sorts of duplication of facts and names in our content. And from time to time those facts are going to change and we are going to have to find all the instances and make the appropriate changes.

If we structure our content in a way that makes that search easier, we can potentially save ourselves a large about of time and money. Making expressions of a truth more findable has many of the benefits of creating a single source of truth and can be less extensive, easier to implement, and more resilient in the face of unanticipated changes.

This is all about the ability to recognize information in context, and the ability to recognize non-canonical references to facts (a person or thing referred to by a nick name, for instance, like calling John Wayne "The Duke" or IMB "Big Blue"). Having your content in the subject domain make this much easier. Second best is having your content in the document or media domains but stored in a content management system with a cataloging scheme based in the subject domain.

Two basic problems: the same truth referred to using different terms, and different truths referred to using the same term. In the latter case, context can tell us much.

Need for strict typing. If only source, got to make sure it does the whole job correctly.

Can use differential single sourcing to produce variants.

Chapter 14. Algorithms: Publishing

All structured writing must eventually be published. Publishing structured content mean 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.

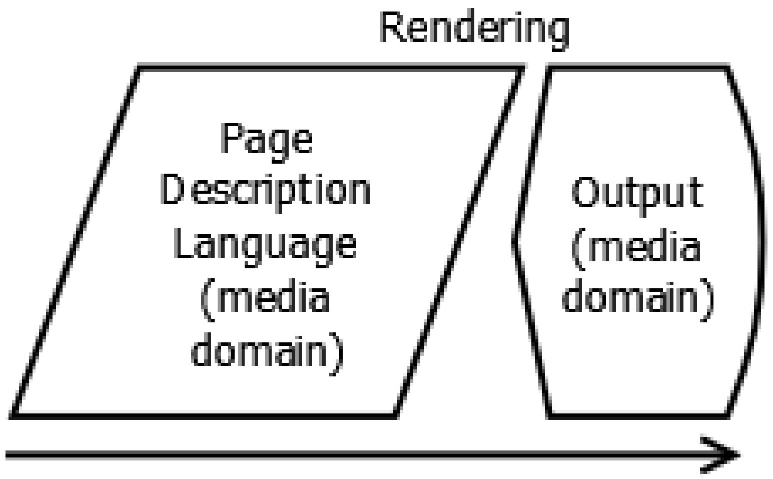
In almost all structured writing tools, this process is done in multiple steps. Using multiple steps makes it easier to write and maintain code and to reuse code for multiple purposes.

In this chapter, I am going to describe the publishing process as consisting of four basic algorithms which I have mentioned in passing in earlier chapter: 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 here.

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 that 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. We looked at a small example of PostScript in the chapter on the media domain:

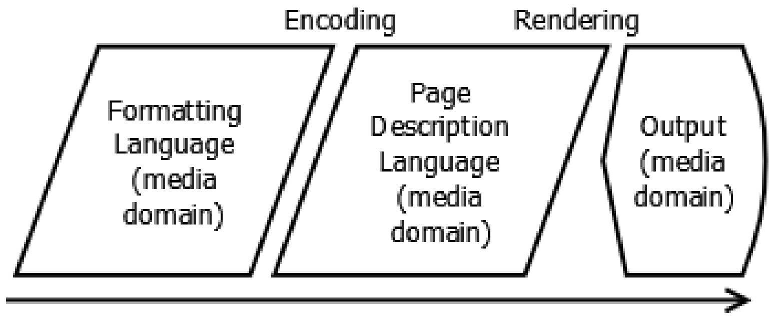
```
100 100 50 0 360 arc closepath
stroke
```

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.

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.


Encoding
Rendering
Formatting Language (media domain)
Page Description Language (media domain)
Output (media domain)

Here is an example of XSL-FO that we looked at in the chapter on the single-sourcing algorithm:

```
<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 figure out how to deal with complex figure 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 the is capable of doing all this the way you want it done.

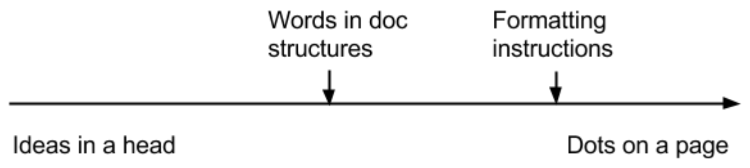
Also, you are going to have to think about just how automatic you want all of this to be. In a high-volume publication environment you want it to be fully automatic, but this could involve accepting some compromises. For example, it is not uncommon for writers and editors to make slight edits to the 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 it 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 HTML+CSS.

The Formatting Algorithm

The job of the formatting algorithm it 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.

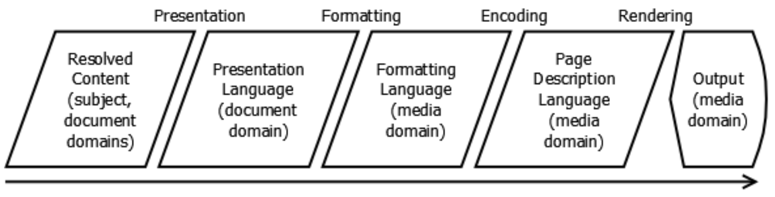


In the case of HTML output, the formatting algorithm generates HTML (with connections to the relevant CSS, JavaScript, and other formatting resources). This is the end of the publishing process for the Web, since the browser will perform the encoding and rendering algorithms internally.

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.

The Presentation Algorithm

The job of the presentation algorithm is to determine exactly how the content is going to be organized as a document. The presentation algorithm produces a pure document domain version of the content.

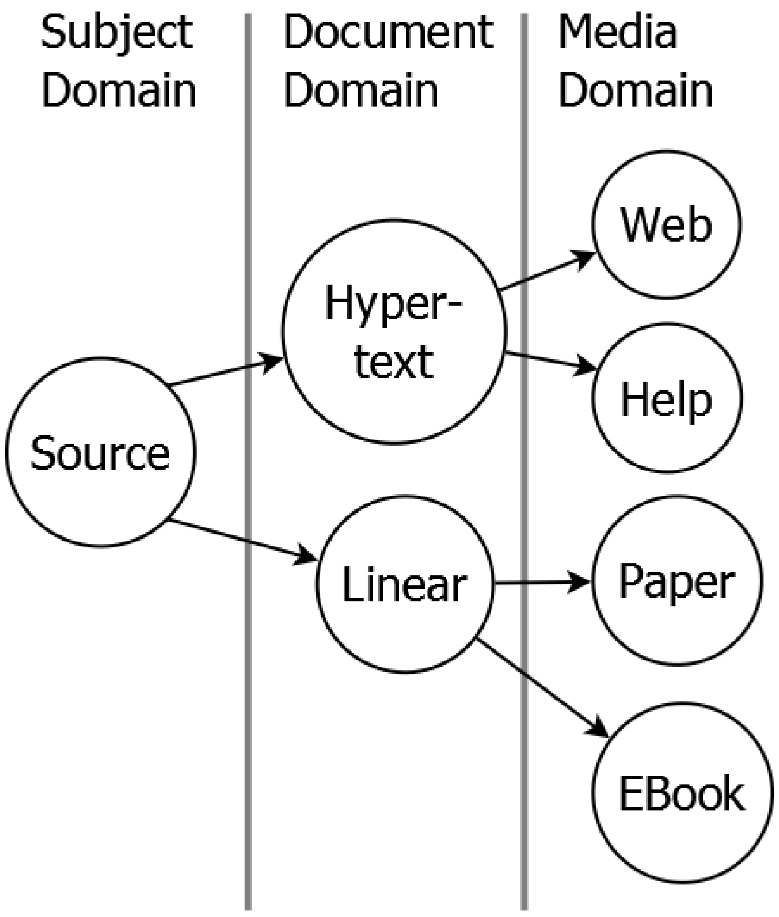

Presentation
Formatting
Encoding
Rendering
Resolved Content (subject, document domains)
Presentation Language (document domain)
Formatting Language (media domain)
Page Description Language (media domain)
Output (media domain)

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 content can be organized in other dimensions by creating non-linear relationships between pieces of content. This includes hypertext links and cross references.

Differential presentation algorithms

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

The presentation algorithm may usefully broken down into several sub-algorithms, each dealing with a different aspect of the presentation process. How you subdivide your publishing algorithm is something you need to decide based on your particular business needs, but the following are some operations that it may well pay to treat as separate algorithms.

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 will look at the linking algorithm in detail in a future chapter.

The navigation 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.

The public metadata algorithm

Many formats today contain embedded metadata designed to be use by downstream processes to find or manage published content. One of the most prominent of these is HTML microformats which is used to identify content to other services on the web, including search engines. This is a case of subject domain information being included in the output. Just as subject domain metadata allows algorithms to process content in intelligent ways as part of the publishing process, subject domain metadata embedded in the published content allows downstream algorithms (such as search engines) to use the published content in more intelligent ways.

If content is written in document domain structures, public metadata is generally created by writers as annotations on document domain structures. But if content is created in the subject domain, the public metadata is usually based on the existing subject domain structures. In this case the public metadata algorithm may translate subject domain structures in the source to document domain structures with subject domain annotations in the output.

This does not necessarily mean that the public metadata you produce is a direct copy of subject domain metadata you use internally. Internally, subject domain structures and metadata are generally based on your internal terminology and structures that meet your internal needs. Public terminology and categories may differ from the ones that are optimal for your internal use. But because this is subject domain metadata, and thus rooted in the real world, there should be a semantic equivalence between your private metadata and the public metadata (the public usually being more generic and less precise than the private). The job of the public metadata algorithm, therefore, is not merely to insert the metadata but sometimes to translate it to the appropriate public categories and terminology.

The document structure normalization algorithm

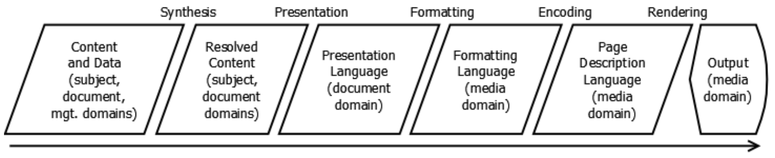
In many cases, content written in the subject domain will also include many document domain structures. If those document domain structures match the structures in the document domain formats you are creating, all the presentation algorithm needs to do is to copy them to the document domain. In some cases, however, the document domain structures in the input content will not match those required in the output, in which case you will need to translate them to the desired output structures.

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.

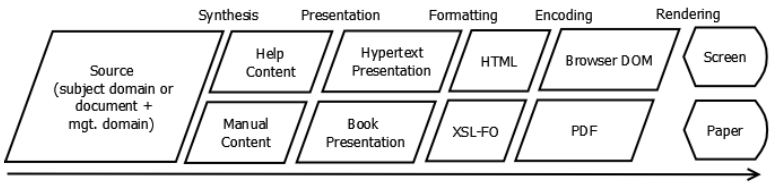
The synthesis algorithm transforms content and data into a fully realized set of subject domain content. Sources for the synthesis algorithm include content written in the subject domain, document domain or subject domain content with embedded management domain structures, and externally available subject data which you are using to generate content.

Content that contains management domain metadata, generally used for some form of single sourcing or reuse does not represent a final set of content until the management domain structures have been resolved. 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. 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.


Synthesis
Presentation
Formatting
Encoding
Rendering
Content and Data (subject, document, mgt. domains)
Resolved Content (subject, document domains)
Presentation Language (document domain)
Formatting Language (media domain)
Page Description Language (media domain)
Output (media 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

The synthesis algorithm can involve a number of sub-algorithms, depending on the kind of content you have and its original source.

The inclusion algorithm

If your content contains management domain include instructions, such as we identified in discussing the reuse algorithm these must be resolved and the indicated content found and included in the synthesis.

As we noted in the chapter on the reuse algorithm you can also include content based on subject domain structures, without any management domain include instructions. Such inclusions are purely algorithmic -- meaning that the author does not specify them in any way. It is the algorithm itself that determines if and when content will be included and where it will be included from. This too is the job of the inclusion algorithm.

The filtering algorithm

If your content contains management domain conditional structures (filtering) they must be resolved as part of the synthesis process. In most cases, you will be using the same set of management domain structures across your content set, so maintaining your filtering algorithm separately makes it easier to maintain.

Again note that you may be filtering on subject domain structures as well (or instead of) on explicit management domain filtering instructions. Such filtering is, again, purely algorithmic, meaning that the author has no input into it. The filtering algorithm is then wholly responsible for what gets filtered in and out and why.

Coordinating inclusion and filtering

It is important to determine the order in which inclusion and filtering are performed. The options are to filter first, to include first, or to include and filter iteratively.

Generally you want the filtering algorithm to run before other algorithms in the synthesis process so that other algorithms do not waste their time processing content that is going to be filtered out. However, if you run the filtering algorithm before you run the include algorithm, any filtering that needs to happen on the included content will not get executed.

Doing inclusion before you filter addresses this problem, but creates a new one. If you include before you filter, you may end up including content based on instructions or subject domain structures that are going to be filtered out, which could then leave you with a set of included content that was not supposed to be there, but no easy way to identify that it does not belong.

The preferred option, therefore, is to run the two algorithms iteratively. Filter your initial source content. When you filter in an include instruction, immediately execute the include and run the filtering algorithm on the included content, processing any further include instructions as they are filtered in.

The rules based approach to content processing that we have looked at in this book makes this kind of iterative processing relatively easy. You simply include both the filtering and inclusion rules in one program file and make sure that you submit any included content for processing by the same rules the moment it is encountered.

```
match include
    process content at href
    continue
```

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. We will look at the extraction and merge algorithms in a later chapter.

The cataloging algorithm

The synthesis algorithm will produce a collection of content, potentially from multiple sources. This collection is then the input to the presentation algorithm. For the presentation algorithm to do its job, it needs to know all of the content it has to work with. In particular, the TOC and index algorithms and the linking algorithm need to know where all the content is, what it is called, and what it is about. They can get this information by reading the entire content set, but this can be slow, and perhaps confusing if the structure is not uniform. As an alternative, you can generate a catalog of all the content that the synthesis algorithm has generated which can then be use by these and potentially other sub-algorithms of the presentation algorithm to perform operations and queries across the content set.

The resolve algorithm

When we create authoring formats for content creation, we should do so with the principal goal in mind of making it as easy as possible for authors to create the content we require of them. This means communicating with them in terms they understand. This may include various forms of expression that need to be clarified based on context before they can be synthesized with the rest of the content. This is the job of the resolve algorithm. Its output is essentially a set of content in which all names and identifications are in fully explicit form suitable for processing by the rest of the processing chain.

Content written in the subject domain is not always written in a fully realized form. When we create subject domain structures, we put as much emphasis as we can on ease of authoring and correctness of data collection. Both these aims are served by using labels and values that make intuitive sense to authors in their own domain. For example, a programmer writing about an API may mention and markup a method in that API using a simple annotation like this:

```
To write a Hello World program, use the {hello}(method) method.
```

In your wider documentation set, there may be many APIs. To relate this content correctly in the larger scope you will need to know which API it belongs to. In other words, you need this markup:

```
To write a Hello World program, use the {hello}(method (GreetingsAPI)) method.
```

The information in the nested parentheses is a namespace. A namespace specifies the scope in which a name is unique. In this case, the method name `hello` might occur in more than one API. The namespace specifies that this case of the name refers to the instance in the `GreetingsAPI`.

Rather than forcing the programmer to add this additional namespace information when they write, we can have the synthesis algorithm add it based on what it knows about the source of the content. This simplifies the author's task, which means they are more likely to provide the markup we want. (It is also another example of factoring out invariants, since we know that all method names in this particular piece of content will belong to the same API.)

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.

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.

The consistency challenge

The biggest issue for every algorithm in the publishing chain is consistency. Each step in the publishing chain transforms content from one part of the content spectrum to another, generally in the direction of the media domain.

The more consistent the input content is, the easier it is for the next algorithm in the chain to apply a simple and reliable process to produce consistent output, which in turn makes the next algorithm in the chain simpler and more reliable.

Building in consistency at source is therefore highly desirable for the entire publishing algorithm. This is an interesting problem because good content by it nature tends to be diverse. Content is the way that we convey the information that is by its nature less consistent and more prone to exceptions. It is the stuff that does not fit easily into rows and columns.

One approach to this problem is to write all the content in a single document domain language such as DocBook. Since all the content is written in a single language it is theoretically completely consistent and therefore should be easy to process for the rest of the tool chain.

The problem with this is that any document domain language that is going to be useful for all the many kinds of documents and document designs that many different organizations may need is going to contain a lot of different structures, some of which will be very complex and most of which will have lots of optional parts. This means that there can be thousands of different permutations of DocBook structures. A single formatting algorithm that tried to cover all the possible permutations could be very large and complex and potentially very hard to maintain.

The alternative it to have authors write in small simple subject domain structures that are specific to your business and your subject matter. You would then transform these to a document domain language using the presentation algorithm. This document domain language could still be DocBook, but now that you control the DocBook structures that are being created by the presentation algorithms, you don’t have to deal with all the complexities and permutation that an author might create in DocBook, just the structures that you know your presentation algorithm creates.

These subject domain documents would have few structures and few options and therefore few permutations. This would mean that the presentation algorithms for each could be simple, robust, and reliable, as well as easy to write and to maintain. You would also be able to do differential single sourcing by writing different presentation algorithms for each media or audience.

The trade-off, of course, is that you have to create and maintain the various subject domain formats you would need and the presentation algorithms that go with them. Its a trade-off between several simple structures and algorithms and a few complex ones.

This trade-off comes up time and time again in structured writing. We will see it again when we look at other algorithms.

Chapter 15. Algorithms: Linking

Few readers read content straight though. Unless the content is perfectly matched to their experience and their goals, there will be points where they need more information, points where they need less, or points where they decide that they need something else altogether. These are defection points in the content, points in which the reader's "next" may not be the thing that comes next in the linear order of the work. That may mean deflecting to other content or to a different way of finding information such as asking a friend or posting a question on a forum.

Deflections are a natural part of information seeking. They are part of what is called information foraging. The reader, in pursuit of their individual ends, will follow where the scent of information leads and that will not always be the next paragraph of the current text.

Writers know they can't meet everyone's needs perfectly every time so they use links and other devices to help readers deflect when they need to. If you have Model A, do this. If you have Model B do that. Supporting deflection helps readers achieve their goals and helps keep the reader in writer's own content, or content they approve of, rather than deflecting elsewhere.

Deflection points can be handled in various ways. The writer may choose to do nothing, leaving it up to the reader to look up a word if they don't understand it, for instance. They may use footnotes, cross references, sidebars, parenthetical material, or hypertext links 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.

Deflection costs the reader more on paper than online. For paper, we may design content to minimize the need to deflect, or to keep deflections inside the local work. For the Web, we may organize content with deflection in mind, allow different readers to choose their own course, rather than trying to optimize one course for all readers. This difference in the ease of deflection between paper and the Web and other hypertext media is one of the main reasons we want to practice differential single sourcing.

Deflection also enters into the discussion of content reuse. We reuse content in multiple documents so that readers don't have to deflect from one document to another to find it.

If we reuse content in different media, we might want to have a different reused strategy between paper and hypertext outputs. Me may want to include the same chunk of content in multiple paper documents but link to a single copy of it when creating a 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. To see how that works, we need to look at deflection and linking in each of the structured writing domains.

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
```

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. (Parenthetical material is a type of deflection; it may be read or skipped.):

```
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="
```

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 from this markup that looks like this:

```
In Rio Bravo, the Duke (see: https://en.wikipedia.org/wiki/John_Wayne) plays an
```

This is not the way we would handle the deflection point if we were designing for paper, but it is a small improvement from a differential single sourcing point of view. At least the link is now visible to the reader. (Technically 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 assuming 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.)

Fundamentally, though, this is not a satisfactory differential single sourcing solution. Unless there were no alternative, 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, commits us to a particular format for the link target. If we link to content that has not yet been published, we gain the freedom to link to any format of that content that we choose to publish. The simplest way to do that is to link to a source file rather than an output file.

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
```

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 taking on this responsibly 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 and 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 you 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 a management domain structure that we looked at in the chapter on the reuse algorithm. 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 `keyref` attribute:

```
<p>In Rio Bravo, <xref keyref="John_Wayne">The Duke</xref> plays an ex-Union co
```

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.

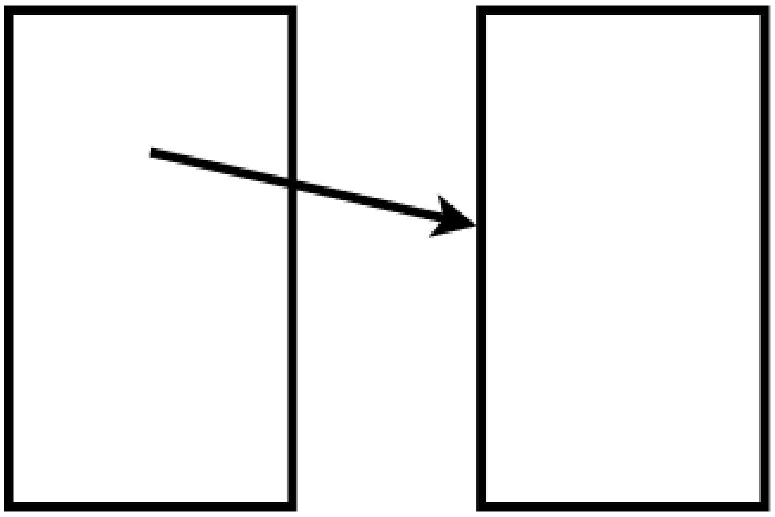
The problem with IDs and Keys

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 seasonably 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 that 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.

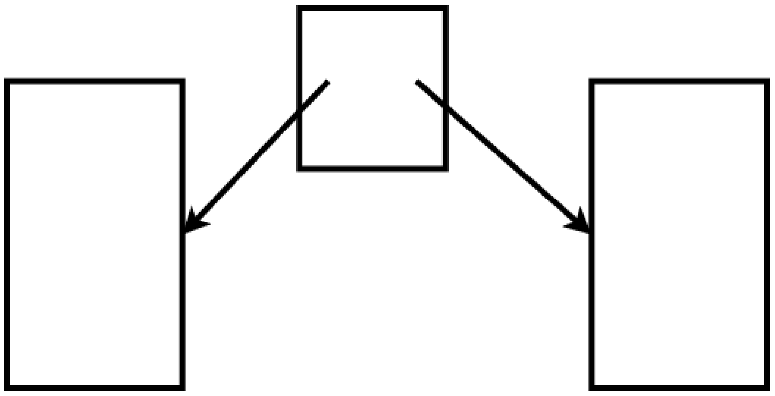
Relationship tables

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, it would be impossible for the Web to grow explosively and organically as it has.

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.



Notice that Dave has three choices about how he does the introduction. He can tell me about you as a seller of ducklings, or you about me as a buyer of ducklings, or he can introduce us both to each other. In the same way a relationship table can describe a link A to B, B to A, or both ways.

One 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 link tables

The problem with link tables is that while they address the problem of link management in reused content and allow you to link to different targets in different publications, they separate the link from the deflection point it supports. The link that marked the deflection point has been factored out of the content so there is no way to put it back inline. Links generate by relationship tables end up in a block, usually at the end of the page.

Since the end of a page is a deflection point in a hypertext system, there is a legitimate case for creating and managing page-level links. But if the page is well designed to fulfill a discrete purpose for the reader, the end of the page is actually the point at which the writer knows least about what the reader might want to read next. The relationship table approach does not support the full array of foreseeable reader deflection points which occur in the body of the page rather than its end.

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.

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:

```
<p>In Rio Bravo, <ph media="online"><xref keyref="John_Wayne">The Duke</xref></
```

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, 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 tagging 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.

For an alternate approach to this problem, and the others we have discussed, we can to turn to the subject domain.

Deflection in the subject domain

There are a number of problems with the management domain approach to links.

- 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.
- 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.
- 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 have the disadvantage the you can't use them to create inline links in your content.
- Unless you resort to ugly conditional structures, you can't use media-appropriate deflection mechanism for differential single sourcing.

As we have seen before, we can often remove the need for management domain structures by moving content to the subject domain. The same it 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
```

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).

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 reveng
```

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 Belize[https://en.wikipedia.org/wiki/R%C3%ADo_Bravo_\(disambiguation\)]). That clarification is what allows us to produce the parenthetical explanation of the phrase in the example above. 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? Writers cannot know with certainty what the deflection points will be for individual reader. But they can anticipate that important related subjects are likely deflection points. This is what they are doing when they create links in the document domain and it is what they are doing in the subject domain. The difference is that in the document domain they handle then mention of an important subject by creating markup that says "create a link to resource X" and in the subject domain they handle it by creating markup that says "this is a mention of important related subject Y". This leaves us with more options about how to handle the deflection point, and that is what we have been looking for.

Marking it 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 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. The markup is giving us the information to make our own decisions rather than forcing us to create a particular structure.

In all our previous examples, 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. 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. We could also use this subject annotation to generate document domain index markers so that we can automatically build an index all mentions of movies in a work.

Subject annotation thus serve 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. You generate whatever links are appropriate to whatever topics are available in the presentation algorithm.
- 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 you 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 presentation 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

Of course, the question remains, what resources do we link to, since they are not specified in the text? If we choose to translate subject annotations into links, we need a way to find resource to link to. We 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 with index 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:

```
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 and 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.

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 is that is of poor quality or that contradicts what you have been saying. The fact that the debate exists 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 records 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. An 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 is 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
```

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? (Than answer here is maybe, but it is not hard to image 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
```

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
            ...
        </ul>
    </li>
    <li>Movies:
        <ul>
            <li><a href="https://en.wikipedia.org/wiki/Rio_Bravo_(film)">Rio Br
            ...
        </ul>
    </li>
</ul>
```

In short, algorithmically deferring document domain links is always tricky, but we can comfortably defer linking of subject annotations.

Different domain, different algorithm

What the linking algorithm illustrates perhaps better than any other is that the movement from one domain to another changes the algorithms in fundamental ways. While the algorithm has the same end in each domain, the way it achieves that end can be significantly different.

One of the points I have tried to make about structured writing algorithms is that they 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.

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 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.

Chapter 16. Conformance

Structured writing is about constraints. Texts that meet appropriate constraints will be of higher quality and greater consistency and will be easier to manage. Texts that record the constraints they conform to can be reliably processed with algorithms. The key to any structured writing systems, therefore, it the ability to assure conformance with the desired constraints.

Constraints have always been part of the authoring process. Style guides and grammatical reference works express constraints that content is expected to follow. Editorial guidelines tell writers what kind of content a publisher is looking for, at what length, and in what format. These are content constraints. If a publisher says that manuscripts must be delivered in DocBook or Word format, that is a technical constraint intended to make the production process flow more smoothly. When the government say that you must submit your online tax return in a particular file format, that is semantic constraint intended to make sure the the government's computers can successfully read and process your tax information.

Some of the constraints described above 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.

This higher level of conformance checking helps make the process easier and the results more reliable. Structured writing is really all about improving conformance to enable automation and improve quality. The conformance algorithm is thus 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. The larger your content set becomes, the more frequent and dynamic your outputs, 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 of content, meaning there is no time to do quality assurance on the output of the algorithm, then reliable content is key, and compliance is how you ensure that content is reliable.

One of the ways in which 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 are going to have a twofold conformance problem: conformance will be expensive, and it will be inconsistent.

The first and best way to ensure conformance with a constraint is to factor out the constraint. If the constraint is that the titles of works 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 names of works. Now it is the publishing algorithm that is responsible for the formatting. 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 or management constraints.

But while we factor out one set of constraints when we do this, we also create a new set of constraint in the new domain. When we factored out the formatting constraint for the titles of works, 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. 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 concerns. 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. It 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 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 not do 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 for algorithms. Inevitably, authors 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. So if the structures you are created for the sake of algorithms prove complex for authors to conform to or difficult to validate, it is wise to 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.

Completeness

Completeness is an obvious aspect of content quality. Unfortunately, lack of completeness is often hard for writers and reviewers to spot. The omission of information they already know is hard to see unless there is an obvious hole in a predefined and explicit document structure. Defining structures that highlight information requirements can significantly improve completeness.

In the subject domain chapter we saw how calling out the preparation time and number of servings for a recipe help 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. By specifically calling out the subjects that are mentioned in every topic you write, you 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 you text to call out the types of certain phrases such as functions names, feature names, or stock symbols.

```
When installing widgets, use a {left-handed widget wrench}(tool) to tighten the
```

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. We will look at annotations, and the ways you can use them, in later articles.

And yes, this is the same markup that I have been using in earlier chapters, and I will explain it eventually.

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. Consistency can apply to many aspects of the content, such as structure and vocabulary.

We have described structured writing as applying constraints to writing. 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 th
```

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.

Accuracy

Accuracy problems too 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 the {rotateWidget()}(function) to ensure the c
```

API function names can be quite tricky to remember sometimes and small typos and 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.

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 quickly? How do make sure updates in one place don’t break other content? Structured writing provides ways to address all of these questions.

By improving validate and automating publishing, structured writing can allow you to release content much more quickly. By changing the way content is organized and linked, it can allow you to add and remove individual pages from a content set without fear of breaking things. We will look at these techniques in more detail in later chapters.

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, such as XML Schema, let you specify the data type[1] 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 schema for these constraints might look like:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qual

    <xs:element name="person">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="age" type="age-range"/>
                <xs:element name="date-of-birth" type="xs:date"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:simpleType name="age-range">
        <xs:restriction base="xs:int">
            <xs:minInclusive value="0"/>
            <xs:maxInclusive value="150"/>
        </xs:restriction>
    </xs:simpleType>
</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 isolating semantic constraints so that they are testable and enforceable.

[1]The data types referred to in the example above are not data types are 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. What a data type really is in a schema is 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.

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 if we choose 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. Reader's will benefit because all the recipes will conform to what you know reader's 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:da
        </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 something 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. 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.

Needless to say, however, 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.

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. For instance, a technical writer documenting an API may produce an API reference, much of which may be extracted from the program source code, and also a programmers guide, which they will 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 constrain as: functions mentioned in the programmers guide must be listed in the API reference.

This is an important constraint. When we implemented 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 being 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 specific 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 {print}(function) to output the text "Hello Wor

    ```(python)
        print("Hello World.")
~~~

Next a function reference listing from the API reference:

```
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).
```

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 difficult 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.

Design for conformance

So much for the mechanical aspects of 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.

While schemas and downstream algorithms can do a great deal to check and enforce conformance, there is also a great deal that they cannot check and enforce, and if the schema and the algorithms are making the writer's live hard with restrictions and errors that the writer does not understand or that get in the way of their creating good content, you are not going to get good 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 we have done it correctly. 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.

The real key to achieving conformance it 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 seek to 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 address the need of the user 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 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 are 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.

Auditing and enforcement still have a role to play, not because authors are hostile to the system, but because they a 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.

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.[1]

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 contains 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 undisciplined 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 the pieces you have are.

Auditing is a content management function and this book is about structured writing, not content management. However, one of the biggest, and least appreciated, benefits of structured writing is that it makes content more auditable. When content management systems fails or become unmanageable, (which they do with disappointing frequency), the root cause is often either lack of attention to regular audits, or the lack of ability to auditing effectively. Without the ability to audit effectively, content sets often end up incomplete, corrupt, and poorly integrated, which reduced quality and increases costs at every stage of the process. And a viscous cycle can develop in which writers, frustrated with the difficulties of the system, create workarounds that further corrupt the information set. Whatever expenses you may incur to implement a more structured structured writing approach could well be offset by the savings associated with more effective auditing of the content set alone.

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 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. An agile approach to content planning is essential in most content projects. But remember that agile is a disciplined approach to evolving a plan. It does not mean hacking away regardless; it means constantly refining your model in a disciplined and deliberate way as learn more about you subject matter, your readers, and your business needs.

[1]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.

But is 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. It is hard to tell if you have the right types of information if you do not closely define those types. If the author if free to write what they will and then tag what they have written with CMS type metadata after the fact, you don't really know what type of content you have, and so you don't have any way to tell if the type definition is correct. The content might work despite not being the type specified, but unless the type is codified and auditable, that success won't carry over to other content. And if the content fails, you won't know whether it failed because it was true to a badly defined type or false to a well defined one.

Having strong well defined topic types makes it easier to audit your topic types to make sure they are doing the job they were designed to do. Similarly, having strong well defined topic types means that you can have greater assurance that each topic is doing the job it is supposed to do, which helps you make sure you have covered all the subjects you should have.

But structured writing can do more than this to help you audit the definition of the content set. If create content in the subject domain, including extensively 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 many 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 markup is how you know what your content is actually talking about, what every author is discovering or thinks needs saying. Without this information it is very difficult to audit that the content set is meeting its coverage goals.[2]

This actually attacks two audit problems. If writers are writing about things outside you 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. 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 author's 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.

There are natural language processing algorithms that will attempt to identify redundant content in a content set, but such algorithms focus on similar texts. But non-redundant content may have similar text, which redundant pieces of content may be expressed very differently. Even when redundancies are found, they may be very difficult to consolidate if they don't have similar boundaries within

[2]With marketing content it is more common to focus on auditing whether content is meeting its 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, but not all behavior 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. And even where behavior is measurable, you want to be able to reproduce the qualities of content that produce the behavior you want, and that means auditing the type and coverage of your content.

their respective documents (the composability problem). Strongly typed content, meaning content that conforms to a model that breaks down and enforces the various components that make up a document, makes it possible to detect duplication in a much more formal and precise way.

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. Strong topic typing helps with all of these things. 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. Whether you express those relationships on output through links or cross references, or whether you relay entirely on tables on contents and indexes, it is still important to understand and manage the relationships between items.

Relationships between items may also be things that matter for management but not to readers. If your have documentation for multiple releases of a product, the relationship between the documentation for 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 share any part of a metadata record in common. But the same problem exists here as it always does with external metadata -- 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 in may cases the important relationship are between parts of one item and the wholes of others. Are function names appearing in the programming topics all listed in the API reference? Are utensils mentions in a recipe all covered in the appendix of kitchen tools?

Structured writing, particularly in the subject domain, helps you discover and manage these relationships by making clear the subject on which these relationships are based.

Making content auditable

I have talked all through this chapter about how using strong topic types makes content more audible. What is a strong topic type? Fundamentally, a strong topic type is one that makes explicit those feature of the content that you need to ensure its conformance and auditability. It is possible for content to conform to all of its rhetorical constraints without the use of structured writing techniques. (Other constraints exist for processing purposes, of course, and do require the use of explicit data structures for that purpose.) But strong topics types provide explicit guidance to the author and facilitate the use of conformance algorithms. They are therefore created to meet you conformance goals. Similarly with auditing, you specify the content structures you need in order to meet your auditing goals.

In many cases, auditing is less of a pass/fail exercise than conformance. Auditing often requires human review. Human review of a large content set is difficult, though, due to the sheer amount of content. An audit algorithm will often work simply 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. 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 can certainly use 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. (Everyone is using the same annotation type, for instance.) 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 covered.
- 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 if our coverage is complete, if writers are getting off track, or if 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, of 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 a advance is difficult.

Chapter 18. The composition algorithm

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 most cases, though, some cleanup is required before the imported files can be used, particularly if they contain subject 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.

Systems that require a high degree of composability 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.

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.

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 it 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 it to make sure that all of your pieces come from the same tagging language. But you must also make sure that the pieces go into a place where they are structurally allowed. Just because all the pieces come from the same languages does not mean that every possible combination results in a valid document. This require planning and careful management to make sure the combinations you create are valid.

However, it is not essential 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 in some cases. 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 structures. 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. Since design for conformance is often more about being specific to the writer's task and knowledge, composability is often best served by creating file formats that do the best job of ensuring conformance from each type of contributor rather than in trying to get everyone to use the same format.

Composition for publication

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 -- document are documents after all, they all have the same basic structures which all document domain languages seek to represent. As long as you can recognize the same basic structures 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.)

Literary 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 piece of writing.

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 it 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 approach to managing the constraints on composition. One is the information chunking 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 though 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.

The difficulties with this approach is that these abstract categories don't always make a lot of sense to writers when they are writing about concrete subject, 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 chunking approach, while it has been shown to improve the quality of writing in some cases, can also impose and artificial clunkiness and lack of flow on the content, leaving it feeling choppy or disjointed.

The other approach to literary 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.)

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 it subject matter in such a systematic or regular way. Then again, the ability to compose such content out of existing pieces is limited. By its very nature such content requires a continuous flow of exposition that is very hard to assemble from pre-written chunks.

Chapter 19. Relevance algorithm

Establishing the relevance of content is essential not only to readers, but to many of the structured writing algorithms as well.

- 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.
- 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 a reader's needs.

There are two fundamental parts to the relevance algorithm: being relevant, and demonstrating your relevance.

Being relevant

All to 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 needs. In many cases, the CMS has a cataloging system for content that require certain metadata fields to be completed, often from a predetermined list of values. If developed correctly, this metadata may have the potential to define the relevance of content, but these requirements are not 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. 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 it exists, it does nothing to ensure that relevant content actually will exist to serve all the needs you intend to serve.

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 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 be 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 will 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 tagging language (or other structured content container) for your content, therefore, one of the most important things to consider is how the reader will recognize what you documents are relevant to.

Relevance is not established the same way for all readers of 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 business page, 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 piece 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 helps 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 source, 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 feed 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. This brings us to the subject of 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 web service the 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 this 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.

It is possible to maintain metadata on type and subject of content in a content management system that hosts content written the 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 unite the showing of relevance to both humans and machines, assuring that the two do not get out of sync.

Finding and filtering

Recognizing the relevance of an individual piece of content is vital to the reader. But there are a number of algorithms that depend on the ability to find relevant content in a large content set, or to apply filters to a content set.

These requirements do not only apply to whole topics. The audit algorithm, for example, my need to be able to judge the relevance of individual fields or phrases to an audit query. If you want to audit the mention of feature names in a documentation set, for example, you need to be able to judge the relevance of individual words or phrases to

Chapter 20. 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:

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.
Transclusion	You can pull in content from another source.
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 graphics and other media that the user can interact with.

As always, you can represent these behaviors in each of the structured writing domains.

- In the media domain, you encode the behavior itself. This means that JavaScript or other forms of executable code may become part of your media domain content.
- In the document domain, you encode an abstract representation of these behaviors or of the document structures on which these behaviors act. In the document domain, the executable code that implements the behavior is factored out, but the document structures that support the behavior or depend on it are made explicit. The document domain, in other words, may call for the behavior, but it does not contain the implementation of the behavior.
- In the subject domain, as always, you record information about the subject matter that may be used to make a decision to implement certain behavior. The subject domain markup neither calls for not implements the behavior. You may, however, choose which parts of the subject matter to markup us explicitly based on the behavior you want to implement.

Supporting active content in structured writing comes down to one thing. If you create static document structures, you will have static content. If you create structures that an algorithm can interpret in different ways, you have the foundation for active content.

In the media domain this means shipping the algorithm itself along with its data set (thought the data set may reside somewhere else and be queried by that algorithm). Of course, all content arrives at the media domain as it is published, so this is always what you are going to deliver. You will always be creating (or borrowing) an algorithm to ship with your content, and setting up your content so that algorithm can read it.

In the document domain, this means defining document structures to explicitly support certain kinds of manipulation. For example, a generic table does not support the action of allowing the reader to sort on any column. Sorting by column only makes sense if the content consists of identically structured rows. In other words, when you sort a table on a column, you are actually sorting the rows. Sorting this table by column would accomplish nothing meaningful:

item	table	stool	shooting stick	chair
legs	4	3	1	4
price	$400	$20	$75	$60

As laid out on a page, a table may group related information by row or by column. Unless you know which is which, you don't know which sort makes sense. Still other tables provide a single value lookup based on two inputs. Sorting this kind of table makes no sense either way. The only logical sorts are on the first column and the first row, which presumable are sorted correctly to begin with.

Still other tables may be lists in disguise. Imagine the consequence of sorting this table on the second column:

1.	Don protective clothing.
2.	Clear the area.
3.	Block all entrances.
4.	Activate the destruct sequence.

Even with row-oriented tables sorting on every column does not always make sense. Sorting on the name column or the size columns or the price column makes sense. Sorting on the description column or the picture column does not.

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 or rows where it makes sense in tables where it makes sense.

In the subject domain, of course, you are not creating a table at all, or at least, not a table in the publishing sense of the word. You are creating a subject-based structure to capture information which can then be used to build different document domain presentations for different purposes. The fact that some of these presentation may be static and some may be active is orthogonal to how the information is represented in the subject domain. In other words, by the time we move content to the subject domain we have factored out the behavior.

This is an idea we have looked at before. When we looked at separating content from formatting we noted that in a world that includes interactive media, we also needed to think about separating content from behavior. We mentioned then that separating content from formatting means at minimum moving it to the document domain. We also saw that when it comes to differential single sourcing you often need to move your content to the subject domain in order to create different document domain presentations for different media. The same is true of separating content from behavior.

In the subject domain you present a document domain structure that is capable of being acted on by an algorithm, but you factor out the algorithm itself. But in a differential single sourcing scenario, you might want a different structure for different devices. You might want a static table, perhaps laid out differently, for static paper or PDF presentation, a sortable table for presentation in desktop web browsers or tables with sufficient room for a table, and a different kind of lookup mechanism altogether for presentation on the limited real estate of a phone. In this case, you can factor out the particular active content structure by moving the content to the subject domain.

In the subject domain, this might mean using a different kind of table: a database table. The form of a database table is simple: it is a series of rows with a common structure. The values in each row are presented in cells in the same order with the same type of data in each cell. For instance, in the recipe example we have been using, the list of ingredients has this form:

```
ingredients:: ingredient, quantity, unit
    eggs, 3, each
    salt, 1, tsp
    butter, .5, cup
```

In SAM, the markup language used for most of the examples in this book, there is a specific markup structure for this kind of information. It is called a record set, and the markup above is an example of it.

With this markup, you can construct a table this way:

eggs	3	each
salt	1	tsp
butter	.5	cup

Or this way:

eggs	salt	butter
3	1	.5
each	tsp	cup

We know, from the semantics of the record set, which rows or columns it would make sense to be sortable, and we can create a document domain sortable table accordingly.

Of course, we can also construct this:

- 3 eggs
- 1 tsp salt
- .5 cup butter

Or this:

- eggs, 3
- salt, 1 tsp
- butter, .5 cup

We can generate almost any kind of active content that we can think of on a piece of content as long as we know its subject domain semantics. For instance, if you annotate a stock symbol:

```
Microsoft ({NASDAQ:MSFT}(ticker)) is a large software company.
```

Then you can look up the current stock price when the page loads and display it like this:

> Microsoft (58.02USD -0.18 (0.31%)) is a large software company.

Alternatively, you could use the ticker symbol to annotate the company name:

```
{Microsoft}(company "NASDAQ:MSFT") is a large software company.
```

This can be output with active behavior in exactly the same way:

> Microsoft (58.02USD -0.18 (0.31%)) is a large software company.

The subject domain markup would be translated to the document domain as part of the publishing algorithm. Here’s what that might look like:

```
<p>Microsoft (<lookup type="stock-price" symbol="NASDAQ:MSFT"/>) is a large sof
```

This would then be translated into the media domain as a call to a particular web service that provides stock quotes, and perhaps some JavaScript code to format the result.

As you can see, 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 to do so without requiring authors to understand or even think about how the active content might work.

This does not mean that active content is a free gift of the subject domain, however. Apart from the fact that you still have to design and implement the content behavior, you also have to think about the subject domain structures and annotation that you will need in your content to drive these behaviors and design them into your content structures. You will also need to make sure that you get a high degree of conformance to these structured 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 21. The extract and merge algorithms

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. If those features are described in any kind of formal data set, such as a database or software source code, we can use structured writing techniques to extract information from those sources to create and/or validate content.

Tapping external sources of content

We 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 therefor 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 and merge algorithm can work effectively with many other structured writing algorithms.

Perhaps most obviously, extract and merge works with the composition algorithm. Extract and merge provides new sources of content for the composition algorithm to work with.

As a source of subject domain content, the extract and merger algorithm also naturally separates content from formatting and contributes to the differential single sourcing algorithm.

By tapping existing information to build content, the extract and merge algorithms also works hand in hand with the content reuse algorithm. 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 -- a process that further reduces duplication within the organization.

Because the extract and merge 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.

If the authoring function is required to conform to information in these sources, the best way to do this is to generate content directly from these sources. Then responsibility for the correctness of the content is shifted to the people who maintain the source you are extracting form. Since they were already responsible for the correctness of the information, this does not create any additional work for them, which means that there is a significant net gain in efficiency for the organization.

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 serves 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 as well as 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 relate 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 not the expertize 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 way in which this is a pity is that when content is produced from these existing systems, for instance by a database reporting process, it 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 matter 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 and turn it into human language content. They also combine this information with formal comments written into the code itself to create an completely human language description of the function. This is an application of subject-domain structured writing and the API documentation tools that do this implement an entire structured publishing system internally, 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, for instance, commonly create knowledge bases to answer commonly asked questions. The material in these knowledge bases is technical communication, plain an simple, yet it usually exists in isolation from the product documentation set, even to the extent that uses 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 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 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 create 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 create the need for extraction and merge algorithms (which are part of the synthesis algorithm within the overall publishing algorithm).

The Extraction Algorithm

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 languages (its grammar):

```
boolean isValidMove(int theFromFile, int theFromRank, int theToFile, int theToR
        // ...body
    }
```

This same informaiton can be extracted by a process 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.

The only problem here is that while this is useful content, there is not enough detail here to build an API reference with this information alone. 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.

The merge algorithm

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 tagging 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[https://en.wikipedia.org/wiki/Javadoc#Example]:

```
/**
 * Validates a chess move.
 *
 * Use {@link #doMove(int theFromFile, int theFromRank, int theToFile, int theT
 *
 * @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 theToR
    // ...body
}
```

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. All you need to be able to merge the too is an unambiguous key that you can find in the source data. You then enter that key as a field in the authored content where it can be used to match the authored content to the relevant extracted data.

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 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.

The Diversity of 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, that each piece of information comes from only one source (this is an aspect of the single source of truth algorithm.

This has nothing to do with keeping it all in the same place. Nor is keeping all content in the same place a particularly useful approach to ensuring that it is only stored once. Ensuring that content is only stored once, a process formally called "normalization" is actually about making sure that information, and the repository in which it is stored, meets an appropriate set of constraints.

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. Furthermore, the business processes and systems that ensure that these constraints are followed are not universal, but specific to each function and organization.

This is not to say that there are never trivial differences between the ways in which different bodies within an organization store and manage information that should not be rationalized. There are all kind of isolated and ad hoc information stores in most organizations that could potentially be much more efficient and (vitally) 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 will 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.

Chapter 22. Information Architecture

Information architecture is the arrangement of content so that it can be found and navigated. 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.)

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. 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 in this way.

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 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 our 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.

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 user 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 it 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). It is designed not as a stand-alone entity but as a hub of its local subject space, offering readers may 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, Ever 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 pattern or type. Structured writing, particularly subject domain structured writing, is very useful in developing Every Page is Page One content.

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. And 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 use-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. I say that it may be inherent, because it is the nature of the subject domain to describe the subject matter and therefore any markup that describes the subject matter may already contain the fields that other functions need for their purposes. 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 structure to support a new algorithm.

If the subject domain information you need is not already there, you may have to add it to the design. But be careful not to reinvent the wheel. Some organizations create complex taxonomies as a basis of categorization, and the terms in the taxonomy may not always match the terms used for the same subjects in the content or the subject domain markup of that content. This does not mean that you need to manually relabel you content. You may well be able to may the two sets of terminology to each other so that you algorithms can find the same subjects under different names. Quite simply, categorization schemes are based on some aspect of reality, and if that corresponds to the aspects of reality captured by the subject domain markup of a piece of content, then categorization can be done algorithmically, even if terms don't match perfectly.

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 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, 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 price 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. They may discover that what they thought the 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 metadata in the subject domain, they should be crated in a disciplined and consistent manner according to a deliberate plan.

Tables of Contents

Tables of content 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 content may see like a document domain language, 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 subjection 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 out a book for map file to create this table of contents. We have an algorithm for creating the TOC, as described in the first sentence of this paragraph. All we need to to write some code to implement that algorithm by reading the structured source files:

```
create-toc
    for each library sorted alphabetically
        create toc-entry library name
        for each function in library sorted alphabetically
            create toc-entry function name
```

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. List are a bottom-up device. You expect to find them as independent pages or as features on a page. In some cases you may want to add a list to page. (Or, to put it another way, if you want the page to have a list in the document domain you may factor out that list in the subject domain and add it back in the publishing algorithm.

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 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 related markup. This is characteristic of the subject domain, which essentially stores information from which documents can be generated. It is in the document domain where reuse is explicit, marking up parts of document to be reused in other documents.

Dynamic content

Another key feature of modern web architecture is dynamic 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 to 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 of 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 generate pages dynamically depends on its ability to identify content that is relevant, based on everything you 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 dynamic content application. There is no possibility for an author or editor to inspect the output of a dynamic 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, that those constraints are completely and correctly expressed by its markup, and that the algorithm correctly processes and delivers the content.

All three of these properties depend on the soundness and simplicity of the markup design. The 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 dynamic 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 (which may be easier to implement and easire for authors to work with).

Chapter 23. The Exchange Algorithm

One of the earliest motivations for structured writing was portability. This was in the days when there were many different computing platforms all using different data formats. Moving a document written on one platform to another platform while retaining any kind of formatting or structure was a major challenge.

To enable the algorithm of porting a document from one system to another, structured writing techniques were used to move the representation of the document away from a series of embedded typesetting codes towards more abstract structures to which style information could be attached (separating content from formatting). The interpretation of this structure and style information into platform-specific typesetting codes or screen display instructions (the media domain) could be done separately on each platform.

System portability is now much easier because the number of platforms has decreased and much of the low level representation of data has been standardized (thought ASCII and UniCode, for instance). Major applications, or compatible equivalents, now run on all major platforms. The reason they can do this is because the files are structured to factor out specific display instructions for different platforms: structured writing at work.

Portability remains a concern, however, but now it has more to do with portability between organizations. How does one organization send content to another organization in a format that it can accept and use?

The problem here is how exactly the receiving organization wants to use the content. Making content exchangeable is not difficult in itself. The most common formats for exchange between organizations are PDF, HTML, and Microsoft Word. There are PDF viewers for every platform. Every platform has browsers that can read HTML. Word files can be read by a variety of applications on virtually every platform while retaining all or most of their source formatting.

If what the receiving organization wants to do is print and read the content, the PDF format fits the bill. If they want to be able to edit it while making minor modifications to formatting, Word or does the job. If they want to edit it and put it on an internal or external Web server, HTML is fine.

But if the receiving organization want to take the received content, manipulate it, query it, reorganize it, and include it in their own publication system, these format are not structured enough to meet the need. Or at least, it will involve a lot of human execution of algorithms because there is not enough structure in the content to hand the job over to machines. This may be find for occasional or ad hoc transfers, but for large or regular exchanges of content it can become inefficient and error prone.

A commonly suggested approach is to exchange content in a common or standardized document domain format. Some common formats suggested are DocBook and DITA. But this is not as easy as it sounds. DocBook and DITA are both large and complex formats that different organizations may use in different ways. Different organizations may specialize either format, and while specialized version should still be formally compatible, they may not process as intended in the recipient's system, especially if one organization has chosen to interpret the standard in different ways -- or to violate it in order to achieve a specific business goal. It is impossible for a standard to express all the constraints that individual businesses or individual writers want to use, so standards are inevitable used in non-standard ways that compromise portability.

Exchange, in other words, is a composition problem and all the levels of composability that we discussing in relation to the composition algorithm apply to exchange. Choosing an exchange format merely because it is common or even a standard does not fully solve the exchange problem unless that format is also highly composable.

As we noted, composability requires a high degree of constraints, particularly if you want to achieve full literary composability. This is not to say that you cannot use one of these standardized languages to exchange content with a particular set of partners, but you will probably need to agree on an additional set of constraints beyond those provided by the standard format, and figure out how to enforce and validate them.

Of course, the best way to enforce and validate content constraints is with a structured writing format that either enforces them or (ideally) factors them out. This, of course, requires that both parties agree to create and accept such formats. As we noted in the case of composability this does not have to mean using the same format, just using formats that are semantically equivalent or can be used to generate a semantically equivalent output.

Some organizations choose formats for their internal work on the basis of those formats being exchangeable with others. As we have seen, different structures support different algorithms. The ideal format for a give organization (cost considerations aside) is the one the supports all the algorithms they want to run on their content. If exchange it the primary goal for adopting structured writing, or if the form that support exchange also supports all the other algorithms you are interested in, choosing a common public format makes sense, since it saves you the cost of any custom development.

But such formats tend to have limited validation and audit capability, because they are large and loose. This can compromise composability at several levels. It is perfectly possible for one organization to sent a file in a public format to another organization that uses the same format and have the exchanged file not be composable at all with the receiving company's content.

A common format, in other words, does not guarantee composability and therefore does not guarantee successful exchange. Different organizations may use a common format like Docbook, for instance, in very different way, rendering their files non-composable with other company's DocBook files. And, of course, other companies may use a different format altogether. To successfully exchange content with another company, you will need to deliver content in the format they use, and in the way they use it.

The capacity to exchange content, then, does not lie so much in having the same format as everyone else, as it does in being able to deliver content in whatever format your customer wants to receive it. And, of course, in having a supplier that will deliver content to you in whatever format you want to receive it in.

We have seen that moving content from the media domain to the document domain helps you deliver the same content to different media or to format it differently. We have also seen that moving your content to the subject domain makes it easier to create different forms of document organization and behavior. The same is true for exchange. An exchange format, in the end, is just another output. The ability to create multiple exchange formats comes from exactly the same techniques that let you create different document designs and output formats.

Portability is often advanced as a principal argument against developing your own structured writing structures. But in fact portability is often best served by custom formats that enforce or factor out the constraints you need to port content reliably to other organizations.

Chapter 24. Change management

Some people talk about future proofing content as a reason for adopting structured writing. This is a misnomer. 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.

But 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. 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. You need to maintain an ongoing capability for making these kinds of changes efficiently and effectively.

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 organization 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.

I have always been skeptical of the value of this practice, however. 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 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. And doing a search and replace allows you to make intelligent choices about historical usage, inflections, and even changes in line breaks. It 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 like this:

```
We here at >($company-name) do not recommend using our product to catch roadrunn
```

I would rather use an annotation like this:

```
We here at {Acme Corporation}(company) do not recommend using our product to ca
```

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 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 names, 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 names 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.

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 you 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 and 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 or 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 introduce 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 the current version of a product suddenly become "previous version" when a new topic is released.

When there is the potential for the effects of 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 often have change management features that can be helpful. For instance, many of them will inform you if a change of deletion of an existing topics 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.

For change management algorithms to be effective, you need to be able to rely on them. If you end up feeling that you have to go through and check all the content or all the output by hand, you will have lost much of the efficiency you sought. Change management, therefore, relies heavily on the conformance algorithm. ...

Chapter 25. Content Management

To many people, structured writing exists mainly as an aid to content management. In this book, I have taken a very different view, focusing on structured writing principally as an aid to content quality. Nonetheless, any content project at any scale requires some form of content management, and so any structured writing project has to take into account how the content it creates will be managed.

Content management principally deals with two problems. One is 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.

The second problem is that or 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. 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.

As we have seen, metadata is pervasive in structured writing. We defined structured writing as writing that not only obeys constraints, it 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.)

There is, therefore, a continuity between content management and structured wiring. Both collect and manage information about the constraints that a piece of content follows. And this means that the boundaries between structured writing systems and content management systems are fluid. Constraint metadata that is held in a CMS in one organization may be embedded in the content in another.

You might expect that the principle 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.[1]

But while you will rarely find much in the way of media domain or document domain metadata stored at the CMS level, CMS's 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 content. And if you are optimizing your content for search you will need to provide the search engine with subject metadata in the form of

[1]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 remain for the author, who had to deal with the structure of whatever sized chunk of content they are being asked to author.

keywords or microformats. All of this depends on subject domain metadata. Subject domain metadata is therefore central to CMS operations. DITA, which is at heart a document-domain system with heavy management domain intrusions, also includes a specification for the external storage of subject domain metadata in the form of the subject schema.

It is a very common pattern for a CMS to store document domain content and attach subject domain metadata to it. 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 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, but are not good at storing or querying the hierarchical structure of content. But this is a case of the tail wagging the dog. It has several disadvantages.

1. 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.

2. The system has no way of telling of the content meets its constraints. It records the content constraints in a separate record without ever validating that the content meets them.

3. It separates the metadata from the content is describes. This allows for drift between the content and the metadata.

4. 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.

There is another trade-off to consider here. Having each piece of content stored in the subject domain makes a lot of sense from a semantic point and makes it easy to store content. The problem them becomes 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 of 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, and 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. You 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 is is still a complicating factor.

To avoid this complexity, and make querying easier to do, some systems may provide a generic metadata label that is attached to all pieces of content in the repository. This will still contain subject domain metadata, but in a less precise format. This has two downsides. First, which it is easier to write queries, because you are always querying the same structure, it can be harder to make those queries return the right content because the subject domain metadata you are querying is not held in such precise containers. Secondly, it forces writers to record the subject domain metadata in a format that does not make it obvious or intuitive what metadata to include or how to express it. This means that the metadata records will be inconsistent, which means that queries will return inconsistent results.

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. There is no generic subject domain metadata record that applies to all subject domain content. 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.

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. You may need to manage you sources and research material, but not your product. 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, require 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.

In fact, the management domain exists largely because of the decision to do structured writing. 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. All of that 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.

Of course, all content is managed. Sometimes by organizing folders on your hard drive. Sometimes by using an elaborate content management system. But management is not part of the content itself. It is a process that exists around the content, that helps the content come into being. But it is not content.

This is not a book about content management. That would be a subject for a book in itself -- and there are several such books already. It is worth noting, however, that content management systems, particularly component content management systems, often use structured writing as a means to enable content management practices. This is absolutely a legitimate use of structured writing, though it is not one that I will focus on.

You should be aware, however, that some discussions of structured writing are based on the premise that it is being used to enable content management. In many cases, the driver for such systems is the content management methodology being used, rather than the writing process of individual pieces, which is the focus on this book. When you are looking at a particular structured writing system, particularly one tied to a content management system, it may be much easier to understand if you look at it as an enabler of content management functionality.

One prominent system that is widely used as an enabler of content management is DITA. This is not to say that DITA is only a content management enabler, but this is the motivation for a great deal of DITA adoption.

Of course, all significant content initiatives need to be manged in some way. The difference is whether you start with structure writing as a means to make content better, and then look for a way to manage it, or if you look for a way to make managing content better, and then adopt structured writing as way to make your content management plan work. If your motivation is the latter, it is entirely possible to end up with a system that makes the content worse rather than better.

The imposition of structure -- any structure -- always affects the quality of content because it changes how authors work and how they think about the structure of the content they are creating. Content quality should always be taken into account in any structured writing system you adopt, even if your goals are not related to quality in themselves.

Conflicting constraints

As we have said all along, structured writing is about imposing constraints on content, and about recording the constraints applied to content so that algorithms can detect them. The first reason to impose constraints on content is to improve content quality. 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.

But the constraints you use to implement content management may not be the same as those you implement to improve content quality. 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.

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 to. 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 the content). 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.

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. (In fact, the desire to do differential single sourcing may be one of the motives for writing in the subject domain.)

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. The consistent document domain format is 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.

While this is not likely to be the approach you would naturally choose if you set out to create implement a content management system without making content quality 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 on data that is known to meet certain constraints, it is necessary to make sure that the constraints are actually being me. This is the role of the guidance algorithm and the validation algorithm. But it turns out that it is much easier to provide effective guidance and preform 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 scheme of concept, task, and reference topics meets the content management algorithms 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, sometimes resulting in a complete breakdown of the system over time.

The variety of constraints and formats found in a subject-domain system may present an 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, 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 simple authoring experience.

We should note that it is possible to implement content reuse of temporary artifacts. If the reuse algorithm points to a URL to source content, that URL could generate reusable content from subject domain sources on the fly, and in a way that is transparent to the reuse system.

Finding the content to use

Once you have determined you criteria for making content fit, you have to figure out how writers are going to find content that fits.

This requires answering the question: does one of these already exist. This is a complex question. It depends on what it required to definitively identify a piece of content as one of these.

- Is it about the same subject?
- Does it have the same scope?
- Is it addressed to the same audience?
- Does it meet my quality constraints?
- Is it in a compatible form?
- Is it blackboxed across all uses? Could one instance change independent of the others.

It is also important to remember that the question of “does one of these already exist” may often be answered in the negative. Every time that a writer gets a negative response, they still have to write the content themselves. Thus the search for reusable content adds an overhead to the writing process even when there is no content to reuse. This can have two undesirable effects:

1. It increases the cost of all authoring activities, which can create greater costs than any successful reuse is saving. It is quite easy for a reuse system to end up making an content organization less productive rather than more so.
2. It creates a disincentive for authors to look for reusable content. This can result in content being duplicated. If this happens the duplicate content might itself be reused, resulting in your systems

having the same content stored twice and each stored instance being reused in other content. This not only increases costs and makes searches more difficult, it raises the possibility that only one of the instances will be changed when a change occurs, resulting in the change being missed in the other reuse chain. If enough inconsistencies like this accumulate, people can start to loose confidence in the whole system.

You can greatly reduce the impact of this problem is you can give authors a clear idea of when they should expect to find reusable content and when they should not.

Dealing with change in the managed content

One of the big motivators for content reuse is to reduce the impact of change on your content. If a change in the subject matter occurs, ideally you would like to only change the content in one place and have that change automatically reflected across the content set.

This can be more challenging that it seems.

Links

When we defined structured writing, we also noted that content should not only meet constraints, it should record the constraints it meets. If you are creating content to be used in multiple places, it is important not only that it meets the constraints (so that it will fit), but that it records what constraints it meets (so that other writers can tell if it will fit).

The leads to two of the biggest questions in planning your content reuse strategy:

1. Do you want all of your content to be potentially usable in many places or only some of it?
2. Do you want to be able to use content by pulling it out of existing documents, or do you want to maintain a separate collection of usable units that are not published themselves, but only exist to be used in other documents?

It is very common to start with the idea that you want all of your content to be reusable, even if it was not written with reuse in mind. This is easy to do at first, but over time it can become hard to manage. Most organizations that attempt to do content reuse at any scale tend to move to more structured methods and may outlaw random reuse altogether.

Chapter 26. Authoring

It may not be immediately obvious that authoring is an algorithm. It is not something done to content, rather it is the process that creates content. The creation of content is obviously a process and activity, but it is (for the most part) performed by people rather than machines, and it is not obvious that it is done using a set of fixed processing rules, which would define it as an algorithm.

But as I have emphasized throughout the chapters on algorithms, the algorithms always begin with structures. Unstructured writing is not an algorithm (or at least, it is a more advanced AI algorithm that we yet know how to write). It is a human action, and an action of the imagination. But structured writing requires something more. It requires that authors not only create content, but that they create content in structures that can drive all the other structured writing algorithms that we want to implement.

Creating content in such structures is an activity over and above the pure act of writing content. The only way we get content in the right structures is if authors create those structures as they write. Our structured writing system can only be a good as the structured content our writers create. Getting the best possible structure from our writers, therefore 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. We need an authoring algorithm.

Like all other structured writing algorithm, the authoring algorithm begins with the design of structures. Creating a set of content structures while thinking only about how they will feed publishing or content management structures is a recipe for a system that is difficult to use, expensive to implement and run, and subject to ongoing data problems. You will not get good data for any downstream algorithms unless your structures are engineered for ease of correct authoring.

All forms of structured writing, even in the media domain require authors 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 a minimal as possible.

But the equation is not quite so simple as this. Structured writing may be an additional requirement, but quality writing does not result from an author 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 author 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 author, and thus potentially improve the quality of the content.

Of course, structured writing can improve content quality in other ways. Most of the algorithms we have looked at pertain to content quality in one way or another. Still, these algorithms work on the structures that the authors create. If those structures are weak, there is a limit to what downstream algorithms can do to improve quality. It really all begins with getting the right structures correctly and reliably applied by authors as they write.

In media-domain systems like word processor 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 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 algorithms we have talked about. It offers little support for composition, reuse, or single sourcing, for instance, and virtually zero support for conformance.

If we want to support these algorithms, we will need something more structured, and this can easily mean something that requires more of the author's attention. If we are proposing to implement management-intensive algorithms, such as reuse, it can mean that authors 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 and how foreign these policies are to the author's experience, this can create a burden far greater that that of creating and manipulating formatting according to a style guide.

We could look at this and say, okay, yes, authoring is now more difficult and more complex than it was before because of all this additional structure and all that they need to learn to apply that structure, but we are getting additional advantages as well, so it is worth it overall. The problem is, as authoring get more difficult, authors do all of the component tasks less well. Attention is a finite resource. The more of it is focused on structure, the less is available for content. The more of the writer's attention is required on structure, the less is available for writing, and the quality of the writing suffers. And 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, the less reliable all of your algorithms become, and the more tenuous all of benefits you hoped to obtain.

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 becomes. But as we saw when we looked at the conformance algorithm, conformance is fundamentally a human activity. Good conformance results from creating structures that are easy to conform to.

One of the most familiar tropes of the content management industry is that problem with content management systems are not technology problems, they are human problems. The solution 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 incorrect, that is the fault of the humans who create the input. But this is and 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. This is really an excuse for poor system design. If humans cannot conform to the structures that the system requires, the fault is in the system design. The structures should be redesigned to be easier to conform to. The authoring algorithm has been neglected in the system design.

In many systems, the attempt to make authoring easy as been an afterthought, treated as a tools question rather than an question of structure design. Markup languages are designed to feed publishing or content management algorithms. The tools such as XML editors are used to provide pseudo-WYSIWY interfaces and various widgets are created to help authors create the often complex management domain metadata involved. The problem with this approach is that at best it addresses the mechanical ease of use issues of creating and managing this complex markup. It does not address the conceptual ease of use issues that arise from the author needing to understand the algorithms that act on this complex and abstract markup.

Hiding markup, in other words, does not factor out structure. But the bigger problem is that hiding markup also hides the structure. It makes the authoring process look superficially easier, but it actually makes things more difficult because it still asks authors to create structures while presenting an interface in which these structures are not visible. In a relatively simple document domain language, this is not the end of the world, because each document domain structure has a media domain equivalent. But any distinction that exists in the media domain that does not directly correspond to a media domain style will be hidden. And if the document domain language is loose, you will often get very poor document domain structures this way, as we have seen in the case of HTML produced by WYSIWYG HTML editors.

I have developed SAM, the hybrid tagging language used for most of the examples in this book, to provide a markup-based solution to this problem. Other solution approaches are possible as well, such as forms-based interfaces.

How is engineering structures for ease of authoring consistent with engineering them to match the specific constraints that we want to impose for the sake of quality and efficient processing? As we have seen, moving content between the domains is a matter of factoring variants from invariants. By refactoring our structures we can produce radically different structures that can still be passed through the same publishing tool chain, simply by adding another processing step. To get to structures that both support our publishing and quality goals and support ease of correct authoring (which is essential to achieving those goals) we refactor our content into a form that is comfortable for our authors while still containing the structure needed for downstream processing.

One of the most important consequences of this, both for ease of authoring and reliability of data, is that in the subject domain, you are not asking the author to think and to structure content in terms of algorithms. In this sense, the move to the subject domain not only factors our specific constraints from the author, it factors out the need to think in algorithms at all, leaving the author free to think in terms of subject matter.

This kind of meta refactoring, where you are refactoring the author's whole working model is extremely powerful. It can have profound impacts on content quality and the ease of the authoring experience. At the same time it may create resistance or confusion for some types of authors, such as technical writers, who tend to be responsible for the whole publishing process, not just the the writing piece of it. This means they have always been responsible for the publishing algorithms. When working in the media domain, they executed these algorithms themselves. When working in the document domain they have created structures that were directly related to the algorithms they understood and felt responsible for.

For instance, for indexing, they created index markers, for links they created xref structures. Yet as we have seen, in the subject domain both these things go away and their functions is replaced by subject annotations. This is much easier for authors who don't want to think about linking and indexing algorithms, but disconcerting for those who are used to thinking about them.

There can be a conflict between ease of authoring and ease of content management as well. 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. But for the writer, writing such small chunks may be difficult. If the chunk the author is asked to create is smaller than the piece the reader is expected to read, it is difficult for the author to get a sense of how the chunk they are writing meets the readers needs. 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.

We noted in the discussion of the relevance algorithm that the use of clearly labeled unambiguous identifiers can help algorithms determine the relevance of a piece of content. The problem is, not everything has a globally unambiguous identifier (such as a company stock ticker) and even for things that do, the author may know know what those identifies are off the top of their heads (for instance, many authors may know the names of companies like Apple, Google, and Microsoft but not know their full ticker symbol. Forcing them to look them up every time you want to to unambiguously identify a company will add a lot of overhead to the authoring process, and it will also make the markup of the information more complex, again impacting the efficiency of authoring.

Fortunately, we don't have to unambiguously identify everything we write about at a global scale. We only have to ambiguously identify it within the context of the content we are writing. There is actually a universal truth about language in this. Very few words have only one meaning or identify only one thing. "Sun" is a big ball of burning gas and a computer company that was bought by Oracle. We distinguish these meaning by context, and we can distinguish content identifiers by context in the same way.

In fact, doing the identification in context is actually more accurate, since there is less possibility of accidentally introducing a confusion with a usage you are not aware of. The more highly contextualized an identifier is, the less ambiguous it is (as long as you specify both the context and the identifier you are looking for).

Functional lucidity

Successful authoring depends on a property I am going to call “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 on paper.

As anyone who has struggled to write even a paragraph in a language they are only beginning to learn can attest, writing in a language in which you are not fluent is painful. 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 tagging language where the tags don’t make intuitive sense, when the structures 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 topic 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 naturally have functional lucidity for everyone who is likely to use it. (Though writers can sometimes disagree about what needs to be said on a subject and how it should be said.)

Functional lucidity and a layered architecture

A layered architecture can be very valuable in providing functional lucidity to a variety of authors. To build a publishing system that is capable of managing a wide variety of content, it is often necessary to create a lot of abstractions in your document markup. If you ask authors to write content directly in those abstract structures you may find that they struggle with the abstraction, and with the complexity and the range of options that go with a language that is designed to handle so many different source and publication types. Providing a set of separate authoring languages (perhaps simplified document domain, perhaps subject-specific subject domain), can deliver functionally lucid authoring languages to different types of authors, greatly improving the authoring process.

At the same time, the simplicity of these special-purpose languages can allow you to be much clearer about what the names and labels mean in the context of that language. Providing and agreeing on precise meanings for terms is much easier the smaller the group of people you are dealing with and the smaller the range of subject matter you are addressing. We have relatively limited vocabularies and we reuse words and phrases between different domain of discourse all the time. Agreeing on what a term means across all domains of discourse is virtually impossible. Agreeing on what it means in a limited domain with a limited audience is much easier.

For purposes of rolling up content from many domains into one larger content set, you will definitely have situations in which the same names and labels are used to mean different things. But this is not a problem as long as you know which domain each piece of content comes from. In other words, every tagging language defines a set of names and labels for content in the context of a particular namespace. (Namespaces are an explicit concept in tagging languages like XML and SAM.)

Placing names into namespaces does not magically resolve all disagreements about what to call things in a wider information space. But it does allow for an information architect to choose an definitive mapping of names from each namespace into the enterprise namespace. The results may still be disputable, but at least they will be consistent. And if they are disputed, and a different mapping is accepted, only the mapping has to change to put the new system into effect. As long as each pieces of content is tagged correctly according to the rules of its local namespace, it does not have to change just because the rules of the enterprise namespace change.

Achieving agreement (and, what is really more important, functional lucidity) within a local domain is easier in some domains than others. The media domain is highly concrete, so there really is not much room for disagreement there. Styles, though, are often given names from the document domain, as they are really a step into the document domain, and this can lead to disagreements because of the more abstract nature of the document domain.

The document domain is the most difficult place to achieve either agreement or functional lucidity. (By functional lucidity, I mean that the language seems easy to use and obvious when you are actually writing as opposed to when you are attempting to hammer out agreement in a committee room.) The document domain is inherently an abstract place, and there are always different ways to abstract from the concrete reality of web pages and books, particularly because of how different hypertext media domains are from paper media domains and the difficulty of truly abstracting beyond those differences. Thus you will frequently hear argument between the proponents of various document domain languages about the correctness and usability of their choices. For instance, partisans of Markdown may praise its functional lucidity (it is very easy to write in) which partisans of ReStructuredText may praise its greater abstraction and range of application.

The management domain is quite concrete, like the media domain, but more arbitrary. It consists either of commands or of management metadata, both of which are particular to a specific management system. Difficulties here are likely to be more about the management processes to be implemented rather than the correct naming of things.

Part III. Structures

Table of Contents

Chapter 27. Rhetorical Structure

Structured writing it an attempt to improve the quality of written work. It's intended outcome is a piece of writing that works well for its intended purpose. (In contrast, as noted in the foreword????, to certain academic or archival use of textual markup where the intent is to represent aspects of content to downstream algorithms regardless of its quality or usefulness.) All writing has structure, from the basic grammatical structures that make sentences comprehensible, to the larger rhetorical structures the make information accessible or frame an argument cogently and persuasively.

While structured writing is not an ontology and does not attempt to express the actual information in a piece of writing, it does aim to support the creation of an effective rhetorical structure. There is, therefore, a relationship between the rhetorical structure and mechanical structure created by markup.

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. Content quality is greatly enhanced when the rhetorical structure is well defined and followed consistently. Also, a well defined rhetorical structured provide an effective baseline against which to compare and measure proposed improvements. Using an explicit predefined rhetorical structure helps enhance and maintain content quality.

In the subject domain chapter we looked at how structured writing in the subject domain can capture the rhetorical structure of a recipe in various ways to serve various business purposes.

A recipe is a fairly well known type of information. The various parts can be organized and presented in different ways, but most people recognize a recipe when they see one. This makes a recipe an example of what I call a topic pattern. A specific markup language for writing recipes would be called a recipe topic type. Different organizations may create different recipe topic types to impose constraints on the recipe format that are specific to their business needs. Each of these recipe topic types is an interpretation of the recipe topic pattern. The topic patterns is an example of rhetorical structure; the topic type is an example of mechanical structure.

Since the point of structured writing it to improve content quality, the definition of topic types should begin with an exploration of topic patterns.

In some cases, topic patterns 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 topic patters is visually distinctive even without looking at a word of the text.

However, topic patterns are not about elements that are visually distinct. They are about the rhetorical structure of the content. They are about the different types of information that are required, they 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 are options within one topic pattern or as defining different topic patterns should probably depend on their effect. Any organization and means of expression that has the same rhetorical affect we can reasonable count as variations on a single pattern.

(Naturally, variations in the topic pattern, though legitimate, make it harder to authors to validate what is written, and to format and publish it consistently. For this reason, topic types usually constrain the topic pattern significantly, insisting on one particular variation as the standard approach.)

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 topic pattern present. But this is not necessarily true at all. It 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 topic pattern. And where we have a topic pattern, we can define a topic type.

This is not to say that once you define a topic type, you will be able to take the markup that you define and wrap it around every existing example of the topic pattern without changing a word. (Again, the point of structured writing is to improve content, not to faithfully represent its current state.) Topic types are more precise than topic patterns and existing unstructured topics that follow a topic pattern will almost always have to be edited to fit a topic type.

What you will find when you start to move content that meets a topic pattern into a defined topic type is that a lot of the content does not fit the topic pattern particularly well. You will find some instances that only partially fit, but which omit information commonly found in the pattern (and perhaps required in the type). You will find that some instance contain information not found in most instance of the pattern, and not supported by the type. You will find information not expressed in the way that the topic type expects.

These discoveries mean one of four things:

- The discernment of topic patterns in incorrect and you are trying to make content with a different pattern fit your topic type. You need to define a new topic types for this new topic pattern.
- The definition of the topic type is incorrect. You need to modify the topic type to more correctly reflect the topic pattern.
- The content is a variation of the topic pattern that is deliberately not supported by the topic type. The content needs to be edited to fit.
- The content is deficient. It needs to be upgraded so that it fulfills it purpose correctly, as defined by the topic type.

Interpreting the mismatch between existing content and the topic 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 it not to represent existing texts, but to make content better. If your current content processed are so good that all your existing content will fit 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 models is not a sign that the models are 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.

(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 your 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.)

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. This is a subject domain pattern for a recipe.

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 topic pattern. 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 pattern.

The make-thing-out-of-stuff pattern is what we might call a meta-pattern. It is not based on seeing similarities between texts, but on seeing similarities in the 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 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 document are composed of just six information types: procedure, process, principle, concept, structure, fact.[http://www.informationmapping.com/fspro2013-tutorial/infotypes/infotype1.html] Document 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 topic pattern 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,[1] which, confusingly, it calls topic types. Like information mapping, DITA assembles documents out of these topic (information) types using a map.

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 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.

[1]Or, at least, it originally proposed these three types. The DITA specification now includes many other topic types, many of which are much more concrete than these original three.

What DITA does do is provide a way to specify a concrete instance type of any of its meta-types. The list of ingredients in a recipe is an instance of the meta-pattern list of stuff your need in the make-thing-out-of-stuff meta-pattern. But a list of ingredients has a specific structure that is not the same as the list of pieces in a flat-pack furniture box, for instance. 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 a 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.

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 both answers 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. This 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 topic pattern.

Because DITA has not 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 topic pattern. 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 about to use it for anything other than to create concrete models. In practice, a meta model tends to be a list of those things that all instances of the 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 topics types (or the even more basic "topic" topic type, of which task, reference, and concept are actually specializations).

Are metamodels 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.

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 metamodel could do this. A metamodel 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 metamodel, and how do we decide which of these types of thing-used-with thing types is relevant to a recipe? (Perhaps, recipe describes foodstuff so thing-used-with-thing relationships are relevant when the object thing is also a foodstuff.)[2]

This is getting complicated enough for me to conclude that, while the {ontologists)(concept "ontology") 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.

Making the rhetorical model explicit

I noted above that there can be a rhetorical model in a piece of text that is just a sequence of paragraphs. You can discern the topic pattern in those paragraph and model that pattern in a topic 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 structure of the rhetorical model more explicit to the reader. 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 type explicit in this way improves the content or not. In its favor, the more explicit rhetorical type 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. (This is a property that the Information Mapping company tends to focus on in their promotional material.) 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 the because you have chosen a plain presentation, that means there is no rhetorical structure, and therefore no topic pattern. The rhetorical structure of the content is a separate thing from the presentation of the content, and the aim of structured is to improve the rhetorical structure, not just to make the presentation more uniform.

[2]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.

Chapter 28. Mechanical Structure

The mechanical structure of structures writing is the structure it presents to machines, which is to say, to algorithms. To create mechanical structures to contain writing, we need a type of structure suitable to what we are capturing and what information we want to capture about it. Traditional computing structures like relational database tables do not work well for this 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.

Flags vs Boxes

A document is fundamentally a linear and ordered data structure. One thing comes after another. The rhetorical structure of a document is not expressed using literary devices (possibly highlighted by formatting changes) within the flow of the text. To impose a topic type to formally constrain the topic pattern that is the rhetorical structure, we need to inject an additional and essentially artificial structure into this linear ordered data structure (one which may, as we have noted, factor out some of the text).

Two commons ways to do this are flags and boxes.

The boxes approach means creating boxes for words and giving names to those boxes. It may also involve putting a label on the box with additional information on it. The name and the label on the box tells us something about the words in the box. To create the boxes, we insert markup into the text to define the beginning and end of the boxes. This is how it works in XML:

```
<box label_1="foo" lable_2="bar">text</box>
```

But not all markup system use the boxes model. Many older systems, particularly those intended for typesetting, used the flags model. To understand the flags model, consider how road signs delineate zones on a road. As you approach a town, you may see a sign saying that you are entering the town. A little further on, you see a sign for a reduced speed limit. On the far side of the town, you pass a sign saying you are leaving the town limits. Here, however, there are still houses and businesses along the road, so the speed limit does not rise until you are past the built up area. Thus the box defined by the town limits and the box defined by the lower speed zone overlap each other. It is not a case of boxes within boxes, but of independent zones defined by begin and end signs.

An example of this approach in the content world is the word processor WordPerfect which takes what it calls a “streaming approach” to document structure in which features are turned on and remain on until they are turned off. Thus in Word Perfect, your could do this, with bold starting and then italic starting, the bold stopping and finally italic stopping:

```
This text is [B>bold, [I>this is bold and italic,<B] and this is just italic<I]
```

This would print as:

> This text is **bold, *this is bold and italic,*** *and this is just italic*.

This overlapping of structures is illegal in XML, so in HTML, for instance, you can’t do this:

```
<p>This text is <b>bold, <i>this is bold and italic,</b> and this is just itali
```

You have to do this, keeping everything nested with no overlapping structures:

```
<p>This text is <b>bold,</b> <i><b>this is bold and italic,</b></i> <i>and this
```

There is more to this question than trivial examples like this. In the academic study of text markup, where markup is used not for the preparation of documents but to mark them up for academic study, there is an debate about whether the hierarchical structure of XML-based languages actually reflects the real structure of the text.

From a structured writing perspective, we don't have to worry so much about whether the markup is objectively true to the text. Our concern is to create structures that improve the quality of content and enable the structured writing algorithms we want to use.

The use of a flags model like that of WordPerfect is rare today. A hierarchical model is almost universally prepared. Whether they truly represent the structure of text or not, hierarchical models are easier to define and process and provide an easier way to express constraints. If your really want to express overlapping structures in a hierarchical language like XML, it is possible to do so by using empty element tags as flags. Thus you can define a language that lets you do this:

```
<p>This text is <b-start/>bold, <i-start/>this is bold and italic,<b-end/> and t
```

However, you would be very hard pressed to find anyone who would tell you this was a good idea. The boxes model works best for most of what you want to do with structured writing and to introduce a flags model into the mix just to express an odd structure like this introduces far more complexity than it is worth.

If you really need to model the highly unlikely bit of formatting in the example above, you should do it as illustrated above:

```
<p>This text is <b>bold,</b> <i><b>this is bold and italic,</b></i> <i>and this
```

In short, structured writing today uses the boxes model, and so should you.

Flat vs. nested structures

But even when we choose the boxes model of markup design, we are still left with some fundamental choices about structure. The first is flat vs. nested structure.

We noted earlier 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 that 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 actually 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. (HTML syntax follows XML's nested model, but it simply does not define document structures in a highly nested fashion like DocBook. Until HTML5, HTLM did not even have a `section` element.)

Which model of a document is correct? You can think of a document as being organized hierarchically, with major ideas expressed in sections, sub-ideas support the major ideas in subsections, etc. There are doubtless 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 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

Today, very few markup languages take the streaming approach of WordPerfect. The problem with it is that when you want to deal with any structure that is naturally hierarchical, the ability to start and stop structures independently just gets in the way and makes things hard to handle. But even with markup languages that are syntactically hierarchical, like XML, you can still define markup languages that are largely flat (like HTML) or more hierarchical (like DocBook).

But 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. (Though this is not universal. Addresses, for instance, which are based on hierarchal 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.) 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 make 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, such that many different boxes are allowed inside each box in many different other. 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. This is a common approach. Its main difficulty is that it involves you in having to do your own language design, which many organization try to avoid. 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.

- 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.

In the document domain, however, you have to make a choice between enforcing a hierarchical model of a document or allowing a more linear one. You will almost certainly do this in the context of hierarchical syntax. You probably don't want to go the WordPerfect route of separate on and off commands. But you do need to decide if you are going to allow arbitrary heading levels. Or, to put it another way, to model a document as a hierarchy of sections with each section having a title, or a single flow of text randomly interspersed with headings, or something in between.

The something in between option can seem appealing but can you can end up with something really unconstrained, meaning that it is difficult to write reliable algorithms to process it. Here is the kind of issues you can run into:

Is a list part of the paragraph in which it occurs? Some markup languages, such as Markdown and SAM, take the view that lists are separate blocks that come after a preceding paragraph. But DITA, DocBook, and HTML will all let you place a list either inside or outside a paragraph:

```
<p> The primary colors are:
    <ul>
        <li>Red</li>
        <li>Blue</li>
        <li>Yellow</li>
    </ul>
</p>

<p> Their complementary colors are:</p>
<ul>
    <li>Green</li>
    <li>Orange</li>
    <li>Purple</li>
</ul>
```

The interesting question here is whether these two structures should be formatted differently for output. Should the list that in inside the paragraph be indented, which the one that comes afterwards be printed flush left?

The default for HTML, at least in the browsers I tried it on (Edge, Internet Explorer, and Chrome) is that both are indented by the same amount, but of course you could change this with CSS and DocBook and DITA processors can do what they like.

But would it be a good idea to make a distinction between the two? Only if authors clearly understood the difference between the two and clearly know when they were choosing one or the other. (If they are writing in a visual editing view of an XML editor, then they almost certainly won't know the difference.)

If you are designing your own markup language, do you want to allow both these forms? Probably not. The more permutations of structure your markup language allows, the more work it will be to write the algorithms to process it. Some allowance may be made for ease of authoring, if you wish, but it may be better to wait for authors to complain about a restriction like this than to build in all the possible variations in advance. (This is also a problem that is more likely to occur in an abstract language like XML than in a concrete language like Markdown or SAM, where the choice is baked into the syntax and the author does not have a choice to make.

The role of syntax

The syntax of your markup language also plays a role in striking the right balance between hierarchy and functional lucidity. Fully explicit markup syntax like that of XML forces every element of the hierarchical structure of the document to the fore in the syntax. The syntax has explicit hierarchical constraints of its own (what XML calls the well-formedness constraint). This pushes the hierarchical structure in the author’s face, which detract from functional lucidity.

Languages like Markdown with implicit syntax are just as hierarchical as their equivalents with fully explicit syntax but they feel flatter. Thus where HTML make you write:

```
<html>
    <ol>
        <li>
            <p>The first item.</p>
            <p>A second paragraph in the first item.</p>
        </li>
        <li>
            <p>The second item.</p>
    <ol>
</html>
```

In MarkDown you write:

```
* The first item.

  A second paragraph in the first item.

* The second item.
```

The latter has just as much structural hierarchy as the first, but feels much flatter. It uses indentation to indicate that the second paragraph belongs to the first list item, but that feels natural and obvious rather than contrived of imposed.

One of my motivations for creating SAM, which is designed for structured writing, and therefore, unavoidably, for hierarchical structures, was to express hierarchy implicit where possible and as naturally as possible where it cannot be implicit, for the sake of improve functional lucidity for strictly constrained structured writing languages. (First and foremost, I created it for myself, to improve functional lucidity for me in the structured writing I do, such as the writing of this book.)

Agreeing on boxes, names, and labels

The names and labels on boxes tell us what kind of thing is in the box. What kind of things they tell us depends on the domain we are dealing with.

In the media domain they tell us what something looks like, either directly (an actual format description) or by reference (the name of a style). In the document domain they tell us what part of a document the words are. In the management domain, they tell us what to do with the content under different circumstances. In the subject domain they tell us what the subject matter of the words is.

The box does not only have to have information from one domain on it. It is not uncommon to have a box with a name in the document domain and a label in the management domain or the subject domain.

The names and labels on the boxes tell us what constraints the words in the box obey, or, as we are writing, what constraints the words we write are expected to obey. Algorithms use the names and the labels to process the content, assuming that they accurately reflect the constraints.

For structured writing to work, it is essential that everyone involved understands and agrees on what the names and labels mean. If we don’t the names and labels will not accurately reflect the constraints we expect, and the whole content quality project falls apart and the algorithms stop working.

Confusion and disagreement about what the names and labels 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 is a concept or a task, while 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, therefor, 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.

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 topic type (mechanical structure) to express a topic pattern (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.

Clear concrete and specific names, and an organization of boxes 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 too munch into the world of the author.

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 the only purpose of which is to serve 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.

Structure and annotation

Broadly speaking, structured markup provides two things: structure and annotation. Structure governs the relationship of pieces. It is structure that says that a list consists of list items, that an API reference must begin with a function signature followed by a list of parameter values, that says that an ingredient listing consists of the ingredient name followed by a quantity followed by a unit of measure, or that a section must begin with a title. It is in structure that we create, impose, and express most of the constraints that are fundamental to structured writing.

Whenever you place content in a structure, you are saying something about that content. What kind of thing you are saying about it depends on the domain of the markup.

Media Domain	The structure says what the content should look like.
Document Domain	The structure says what role the content plays in the document.
Subject Domain	The structure relates the content to its subject matter.
Management Domain	The structure says how the content should be managed.

But markup structures, such as XML elements, only allow you to say one thing about the content they contain. Suppose you want to say more than one thing about a piece of content? In that case, you can add annotations to express additional information.

The information in annotations does not have to be in the same domain as the structure they apply to. For instance:

- In DITA, the keyref attribute is used to add management domain metadata to a number of document structure elements.
- In HTML, the style attribute can be used to add media domain metadata to the document structure.

But not all annotations add additional information to structures. It is possible to have annotations that stand alone. At a certain point in defining the structure of content we get down to what is essentially free-form narrative content -- that is, paragraphs of text. But even within a paragraph, that are pieces of text that we want to add metadata to for use by downstream processes. Examples from each of the domains include:

Media Domain	A `bold` or `strong` annotation on a piece of text you want to emphasize.
Document Domain	An `xref` annotation to create a reference to another part of the content.
Subject Domain	A `function` annotation to identify that a piece of text is the name of a function.
Management Domain	A conditional annotation to identify a piece of text that may be conditionally included or excluded from output.

So we are clear on this: all structure annotates. Merely placing a piece of writing in a structure provides some information about it, and therefore annotates it. We add markup annotations when we want to express additional information or when we want to add information to content that is not explicitly structured. To what extent you need additional annotations depends on how detailed your structures are. More detailed structures provide more detailed annotations (and constraints), meaning you will have less need of additional markup annotations. We should also note that this principle extends to the repository level. The metadata attached to a piece of content by a repository is an annotation, and the need for such additional annotation can be reduces with more detailed structured in the content itself.

Structure and annotation in SAM

In SAM, there are a number of types of structure, but the main extensible structure is the block. Annotations are part of the definition of the language. Annotations can be added to blocks and call also float in text.

```
section:(#annotations) Annotations
    This is a paragraph containing an {annotation}(concept).
```

In the example above, `section` is a structure containing a title and a paragraph. The `section` structure has an annotation which is contained in parentheses immediately after the colon that defines the structure. In this case it is an ID annotation and assigns the ID “annotations” to the structure.

The word “annotation” in the paragraph is annotated with a free-floating annotation. The curly braces delineated the text that is being annotated. The parentheses contain the annotation itself, which in this case is a type annotation, indicating that the word “annotation” is a reference to concept.

Structure and annotation in XML

In XML we have two principal types of markup, the element and the attribute. These more or less correspond to the distinction I have made here between structure and annotation. Elements are for structure. Attributes are for annotation. The only caveat is that XML does not allow for free-floating annotations. To create an annotation inside a paragraph, you have to use an element. A paragraph (or any other block of text) with elements floating in it is called "mixed content".

So the SAM example above could be expressed like this in XML:

```
<section id="annotations">
    <title>Annotations</title>
    <p>This is a paragraph containing an <concept>annotation</concept>.
```

Concept here is behaving as an annotation. There is little in the way of structural restrictions that can be placed on it, other than that it floats in paragraphs or possibly other elements that contain text. There is therefore little you can do either to require its use or to constrain it.

Structure in attributes

While SAM draws a distinction between structure and annotation, XML, as we have seen, does not. Its distinction between element and attribute roughly corresponds to my distinction, but not entirely so. This is a reflection of the fact that while SAM is designed for documents, XML is deliberately more general and designed for data applications where the structure and annotation distinction may not apply, or may not apply in the same way as it does in documents.

This means that it is possible to design XML languages that use attributes (normally an annotation mechanism) to express structure.

Consider these examples of HTML Microformats from Wikipedia[https://en.wikipedia.org/wiki/Microformat]. The first example shows an address formatted as a list.

```
<ul>
   <li>Joe Doe</li>
   <li>The Example Company</li>
   <li>604-555-1234</li>
   <li><a href="http://example.com/">http://example.com/</a></li>
</ul>
```

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 useful about what the content itself is about. It does not tell you anything useful beyond what document structure it belongs to. It is not much use as an annotation.

The second example adds hCard microformat markup:

```
<ul class="vcard">
   <li class="fn">Joe Doe</li>
   <li class="org">The Example Company</li>
   <li class="tel">604-555-1234</li>
   <li><a class="url" href="http://example.com/">http://example.com/</a></li>
</ul>
```

This example adds subject domain metadata in the form of the class attribute. For example, it says that the phrase `The Example Company` is a reference to an organization (`org`). So far this is just regular annotation.

However, there is not just annotation going on here. There is actually subject domain structure being expressed. Not only is the list item `The Example Company` annotated as `org`, the list that contains it is annotated as `vcard`. The meaning of `org` is actually dependent on it being part of a `vcard` structure.

In other words, the annotations in the sample above are equivalent to pure subject domain markup like this:

```
<vcard>
   <fn>Joe Doe</fn>
   <org>The Example Company</org>
   <tel>604-555-1234</tel>
   <url>http://example.com/</url>
</vcard>
```

In other words again, 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. So the microformat adds a second, hidden, subject domain structure to the document for readers who are algorithms rather than people.

Authoring our content this way would obviously be inefficient. But as we have seen, we can process subject domain structures to produce document domain structures with the presentation algorithm. If you write this content in the subject domain using markup like the above, you could transform it to the HTML with microformats example shown above with a presentation algorithm like this:

```
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
```

In other words, we can factor out the document domain structure from the HTML for authoring and storage and then factor it back in for output.

Structure vs. annotation

Since both structures and annotations can contain information about content, you often have to decide whether to use one of the other to capture some of the metadata you need. Here are some of the basic considerations:

Annotations are more difficult to constrain than structures. It is not that you can't constrain which annotation apply to which structures. In XML, for instance, you could require that every element a have an attribute b. But while that may work in some cases, it only applies if b is relevant to all instance of a, and with most of the things you want attributes for, that is not the case. So that means the b needs to be optional, which means it is less constrained. (There is a constraint on which attributes **may** occur, but not on which **must** occur.)

Structures provide superior guidance. Even if all you need for downstream processing is metadata that could be provided by annotations, using structures makes it easier to guide authors to provide the metadata you need, and to audit the content to make sure the information has been provided correctly.

Structures establish context. As we have seen, the ability to establish context is key to many algorithms. Annotations exist in the context established by structures, but do not create new context themselves.

Similarly, if you are annotating words and phrases within a paragraph, it is difficult to constrain which words should be annotated or how. If you really need a particular piece of information to be constrained consistently, you probably need to factor out the paragraph and capture it as a set of field structures that can be properly constrained and audited. (You can turn it back into a paragraph in the presentation algorithm if you want to publish it that way.}

Secondly, annotation are generally harder for writers to create than structures. This depends in part on which markup languages you are using, and which editing tools, but in XML, for instance, where annotation are usually created using attributes, a string of attributes on an element can make the document very hard to read in plain text view, which most XML editors, which are generally optimized for document domain editing, make it much more difficult to add attributes than elements, and generally hide attributed from view (at least by default) making it harder to edit them, or to see how they impact the overall structure of the document.

Of course, if you need to add more than one piece of metadata to a piece of content, you will have to use annotations to do it because one structure, by itself, can only express one piece of information. However, by moving your content to a different domain, you may be able to avoid the need to use as many annotations. When you write content in the document domain, for example, any media, subject, or management domain metadata you want to add to the information must be expressed as annotations. But if you move your content to the subject domain, you can often factor out the document domain, media, and management domain structures, meaning you can rely on the subject domain structures alone and have less need for additional annotations.

(It should be noted here that it is uncommon to have a real world tagging language that is entirely in one domain. While media domain languages are often entirely in the media domain, document domain languages commonly include some subject domain structures or annotations and often extensive management domain structures and annotations. Even subject domain languages usually need at least basic document structures like paragraphs and lists, and may also need some management domain structures and annotations for management information that cannot easily be factored out into sensible subject domain structures.)

Structures are more expressive than annotation. Structures break a document into pieces and label the pieces. This allows you to see the structure of the document as a whole and reflect on it as a whole. It allows you to navigate the document more easily and audit it more thoroughly. In a markup language like SAM which deliberately expresses the structure, it is easier to take in the structure of the document at a glance.

What happens when you need to format part of your document differently, but your document domain language does not have a document domain structure for the document element you are creating? For example, if your document domain language is HTML, it does not have any structures for common document domain objects like procedures, sidebars, of bibliographies?

Of course, you can create these things in an HTML document, using regular features like lists, divs, and paragraphs. To indicate how they are to be formatted, though, you will have to either apply style information directly in the file, or else add some kind of metadata that indicates the special role that the list, div, or paragraph is playing. In HTML, the normal strategy is to use a class attribute. So, you could support special styling of a procedure list by giving it a class attribute of “procedure”:

```
<ol class="procedure">
    <li>Lather.</li>
    <li>Rinse.</li>
    <li>Repeat.</li>
<ol>
```

This is, in some ways, like giving HTML a procedure document domain object. We can now format it as a procedure. And because we are using a hierarchical system, we don’t need to add class attributes to the li elements to format them differently. We can set an invariant rule that all li elements that are children of an ol element of class “procedure” will be formatted a certain way.

```
ol.procedure li
{
}
```

We said that when we define a style, we are extending the media domain. In the same way, if we work in the document domain, we often need to extend the document domain we are working in. HTML is not an extensible language. To fill the need for an extensible document domain language, the W3C adopted XML (eXtensible Markup Language). XML allows you to extend your document domain (or any other domain) in two ways:

1. by creating brand new structures in that domain.

2. by using attributes to specify additional type information for an existing structure.

Another method for extending the document domain is provided by DITA through it specialization mechanism, which allows you to create a new structure by specializing an existing one. In practice, this means defining a new structure but with the difference that the system knows about the new structures relationship to the old one, so that if you specialize a numbered list, the system knows that your new list is a type of numbered list.

Yet another method is provided by SPFE, which supports creating a library of structures from which you can select the ones you want to build a set of document domain (or other domain) structures for your use.

Chapter 29. Metadata

Not available yet.

Chapter 30. Blocks, topics, fragments, and paragraphs

A common theme is many structured writing systems is the idea that documents are made up of various types of blocks. This idea affects both how you look at information design and how you look at the creation of data structures and algorithms.

Every hierarchical markup system, of course, is made up of blocks, and of blocks within blocks. XML, for example, is simply a set of elements inside elements. But when we design markup languages and structured writing systems, we use these basic building block to create more complex blocks that have a wholeness of their own. We recognize them not simply as on step in the hierarchy of a document, but as a distinct type of object that we can give a name to and assign processing logic to.

Semantic blocks

At the risk of adding further burden to an already overloaded term, I am going to call them 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 is a list. 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. It a structure has a name like this in the real world, the block that implements it (in the terminology I am coining for the 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. This is not to say that the building blocks of a semantic block may not also be semantic blocks. In fact, this happens frequently. Nor is it to say that the the distinction between a semantic and non-semantic blocks is fixed or inflexible. A semantic block is a block that implements a structure that has meaning to you independent of its function as markup, and what you regard as meaningful, I may regard as an implementation detail.

The point is not that we must agree on exactly what is a semantic block and what is not, but that you should think of markup design 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.

An example of a semantic block in the document domain is a list. DITA, DocBook, and HTML all define lists, and each of them defines the internals of a list different. Nonetheless we recognize that each of them is an implementation of the idea of a list.

However each of them define it, a list is made up of structural blocks that build the shape of a list. I’ll illustrate this with XML since it make the blocks explicit:

```
<ol>
    <li>
        <p>This is the first item.</p>
    </li>
    <li>
        <pi>This is the second item.</p>
    </li>
</ol>
```

Other document domain examples 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 construct a semantic block while still recognizing different implementations for what they are.

In the subject domain examples 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 such 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.

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 object like lists or **FIXME!**XXXXX rather than a sea of tags and the task becomes easier to understand (and the tags easier to remember).

The structure of a semantic block can be strict of 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 tagging 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 simplicity 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.

Functional 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 function they perform for the reader. I'm going to call these functional blocks. (To be clear, a functional block is a type of semantic block.)

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 functional 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.

What domain do functional 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 blocks, so IM's blocks are components of document, and therefore in the document domain. They simply look at the components of document more from a functional standpoint than the morphological standpoint of a structure like a list or a table.

Functional blocks will typically be made up of morphological blocks. In some cases, a block may fit both categories. For example, a procedure probably counts as both a functional and a morphological unit, since it has both a distinct morphology and a distinct purpose. Alternately, in an Information Mapping context at least, we might regard it as a functional block that consists of exactly one morphological block.

DITA also adopted this idea of documents being made up of functional 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 functional 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 functional blocks: concept, task, and reference.[1] (DITA actually defines more topic types than this today.) This idea is appealing because 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).

This very simple triptych is very 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 other (myself included) criticizes it on the ground that it tends to break content down too fine for be useful and robs content of its narrative thread.

It can fairly be said that this depends very much on how you use it. But from a structured writing point of view, the purpose of structured writing is to use constraints to improve content quality and this approach, by itself, lacks some of those necessary constraints. As we have noted, some approaches to content reuse reject significant constraints in favor or easy composability. DITA does, however,

[1]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. """[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 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.

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.

provide mechanisms for creating more constrained functional blocks, though not a means to constrain how blocks are combined.

All of this is quite distinct from the subject-based information typing of the subject domain. The instructions in a recipe and a knitting pattern are both tasks in DITA terms and procedures in IM's terms. But in the subject domain they are distinct types because there is a distinct way in you which write knitting instructions.

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. All three 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. However, the specific structured writing structures that you would create in each case are different. Both the Information Mapping and DITA approaches, when reduced to specific markup structures, create abstract document domain structures. The subject domain, of course, creates subject domain structures.

At the same time, the subject domain is not attempting to define an ontology. It is not trying to capture knowledge in a formal way. It is trying to capture and to shape content in a way that is specific to its subject matter.

We should be clear that this is not the only view of the structure of a document. We might also see a document as a punctuated flow.

Topics

What is the next unit up from the semantic block? It is the unit that combine semantic blocks to form a complete coherent content item. We might perhaps call it the narrative block. What is the name of the narrative block?

The narrative block obviously comes in different sizes. It could be a help topic, a web page, or an entire book. Or a book might be made up of multiple narrative units, such as a chapter, an article in an encyclopedia, or a recipe in a cook book.

It is probably fair to say that the narrative block is the largest unit of content to which we can usefully assign significant constraints. Or rather, the constraints that we impose on larger collections of content are of a different type an implemented in different ways from the constraints we impose on writing and through structured writing techniques. (As we have noted, DocBook a format primarily designed for whole books, is largely unconstrained, opting almost always to give the author multiple choices as opposed to strict guidance.)

The narrative block differs also between the paper world and the hypertext world of the Web. On paper a book consists of a basically linear series of chapters. On the Web, a site consists of a linked set of pages. But on the Web, pages are not restricted to linking to other pages on the same site, and search engines, which generate dynamic ad hoc links to pages rather than sites so that it is fair to say that the Web consists directly of linked pages, meaning that a narrative unit of hypertext relates itself to other content in a very different way from a narrative unit on paper.

Increasingly, though by no means universally, the term "topic" is used for the narrative block in structured writing terms. There is some confusion caused by DITA's use of the word topic for its functional blocks, and some confusion in the DITA world too about whether Concept, Task, and Reference units should be presented as narrative blocks or not.

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 functional 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 a items in a list conditional based on which of the list items applies to different versions of a product.

Individual list items are not really 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 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, to create relationships and dependencies that hard to manage, because they don't follow any structural logic. You should approach their use with great caution and restraint.

Paragraphs

Paragraphs are the thing that make structured writing different from most other computable data sets. It is rare in any other data set to see a structure floating withing the value of another structure. But that is exactly what happens in a paragraph.

```
In {Rio Bravo}(movie), {the Duke}(actor "John Wayne") plays an ex-Union colonel
```

In this examples, he annotations on "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
```

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 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 format are not a good fit for content. They may be able to model element content and data content, but they lack and 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 Way
```

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 narrative 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 any and all of the 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 structure writing algorithms work better. (You may have noticed that the subject domain provides the most constrained and elegant solution to most 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 still make considerable use of paragraphs and other basic text structures. 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. 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.

Subject-domain structure 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.

And therefore, just as media domain structured writing needs to float elements in the middle of paragraphs to describe formatting, the document domain to describe the role of a passage, the management domain to assign conditions or extract content for reuse, the subject domain needs to do so to describe the subject referred to in phrases within a paragraph.

In planning you 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 content strucures to be modeled below the paragraph level.

Chapter 31. 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 had a height and a width. (Yes, in print the ink also had depth, but that is irrelevant for this purpose.) 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 the job of the rendering algorithm. It is one of the first things that gets factored out as we begin to structure content.[1] 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.

But when it comes to content that has two dimensions, 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:

This particular 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). It is impossible to know exactly how this table was marked up, or which domain the content was written in, or how the 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 into the accompanying text
- 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 really not one good option in the bunch, you are appreciating the extent of the problem. For books being transferred to e-readers, of course, there is not much that can be done

[1]There can, indeed, be some issues with rendering algorithms when it comes to hyphenation, widows and orphans, and the location of titles relative to the text, but with the appropriate document domain markup to delineate basic document structures, algorithms can handle these issues reasonably well. There are some fine points of typesetting aesthetics that may be difficult to automate, and algorithms can't edit the text itself to make a line break work better, as some human book designers will do, but that kind of manipulation is rarely needed. And if it is, you have the options of translating the content into the media domain and manipulating it by hand at that point. Of course, you won't do this for high-volume content that is delivered on a frequent schedule, but then you would not be able to do that kind of hand typesetting manipulation on that class of content anyway. Daily newspapers are not typeset with the same level of attention to aesthetic detail as hardcover books.

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 reader to user small devices, such as tablets, e-readers, and phones for reading means the wide tables are problematic for new content. On a phone, the amount of such 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 of 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 loose 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 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 value, 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 among of text in the each column. But this is tricky because some columns have side heads which means 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 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 boundaries 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 the illustration above 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 reader with a WYSIWYG page width for them to adjust column widths by eye. 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 a simple example, 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>
  </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 DocBook 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.

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

One of the fundamental principles of structured writing is to factor out constraints wherever possible. Also, as I mentioned in the foreword, structured writing is not about creating markup that has fidelity to existing texts, it is about improving content quality and enabling effective content processing algorithms.

One of the most fundamental of those algorithms is single sourcing, and as the above shows, tables are a media domain construct that are hard to create in the document domain and do not single source well to different media.

Another fundamental algorithms is authoring, where the goal it to achieve functional lucidity by ridding the author of distractions. One of the basic ways of doing this is by moving the content to the document or subject domains to remove formatting distractions. Yet tables force the writer either to work in the media domain or to create complex and convoluted markup in the document domain to try and hint appropriate table rendering to the rendering algorithm.

Tables, in other words, as something we should be actively trying to factor out of the authored version of our content. If we are going to factor out tables, though, we need to figure out what to factor the content into. There are a number of alternatives, depending on what the content was being used for.

Alternate presentation

In many cases the use of a table simply isn't necessary. There are other way to present the content with no loss of comprehensibility or quality. In a structured writing environment, prefer the non-tabular version when available.

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.

Semantic 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 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 formatting algorithm to make intelligent choices about column widths and all the other rendering issues that arise with tables and pass on appropriate hits to the rendering algorithm for rendering ingredient list tables in particular.

Another example where tables are frequently 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 very 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 included 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 must 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.

Don’t use tables for layout

In the early days of the web it was common to use HTML tables as a way to lay out elements on a page. This practice is how heavily discouraged. CSS positioning is the preferred way to position elements on a page (in particularly because it supports the use of responsive design to make pages display well on different sized display ports. This is old news. However, thinking about information in terms of page layouts is still an easy habit to get into.

We have talked about topic patterns -- the rhetorical structure of content -- as the basis for the development of topic types -- the mechanical structure of content. But topic patterns are often associated in our minds with a particular layout of page elements. When we imagine or device a topic pattern, we often do it by organizing page elements in certain ways. This is all perfectly legitimate. All content is formatted and displayed for consumption. The point of structured content is not to divorce content from presentation but facilitate a happier marriage, especially when the same content must be married to different media. Thinking about your topic patterns in layout terms, therefore, is perfectly reasonable.

But when it comes to devising topic types based on those topic patterns, the specifics of page layout need to be factored out. This is essential to moving the content to he document or subject domains and to realizing all the benefits that come from that. As long as the semantics of each of the content elements is maintained, so that the presentation algorithm can tell which is which, the page layout can be recreated successfully and consistently, and, if needed, different layouts can be created in different media. The trick is to learn to look at your page and not see tables, but objects with clearly defined types and names. Learn to see data as data, lists as lists, procedures as procedure, and prose as prose.

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 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. 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, making the topic pattern harder to recognize, which would reduce findability.

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.

Fortunately, or unfortunately, 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 you rendering algorithm to make a different choice for different kinds of code.

What is essential is that your document domain or subject domain markup clearly indicates when a piece of text is code. Preferably is 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. The are two basic formats for graphics, vector and raster(concept “raster graphics”}. 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 as 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 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.

Some raster file formats include metadata which may include the resolution (from which you can calculate the intended size). But there is no guarantee that that information is present in all files, or that the tools typically used to implement publishing tool chains can read that information.

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. Photographs of small objects probably should not be blown up to 10 times life size (unless that is a specific intention, to show detail not visible to the eye). 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. 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 artist’s intention.

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 a 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.[2]

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.

```
<img
    src="http://www.example.com/images/example.png"
    height="150"
    width="140" />
```

But do these values represent the size of the graphic or the size at which is it 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)?

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 attributes.[http://www.docbook.org/tdg/en/html/imagedata.html] 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. (Many online media cannot render common vector graphics formats.)

You may also want to use different resolutions of the same raster graphic for different media or for different purposes. This may include redrawing a graphic to reduce fine detail, rather than simply scaling it mechanically.

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.

In some cases we may go event further back and talk about the idea of the image in the artist's head. Rather than creating a single source file and generating various rendering from that, that artist may 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.

Thus we could have a single ideal image with multiple source renderings and multiple derived rendering from each source. We want to use the correct version in each different rendering of our content. 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">
```

[2]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.

```
    <imagedata
        fileref="../graphics/assemble.png"/>
  </imageobject>
  <imageobject condition="fo">
    <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 version 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 a system called XSL-FO 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 have to go back to the distinction between the idea of a graphic and the rendering of a graphic. All of the approaches above have the author include a specific rendering of a graphic. To factor it out, we instead have the author include the idea of the graphic.

There are several ways to do this. In fact, this is really the same idea as we saw in the reuse algorithm were we factored out the filename of the content to be included and replace it with a semantic representation of the file. There we refactored an explicit filename in 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.
```

into 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.
```

and 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.
```

Any of these techniques can be applied to the insertion of graphics just as well as the insertion of text. However, there is the added issue of the metadata that describes the various properties of the image and it 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)
```

The file `assemble.xml` 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 t
    </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 publishers existing tool chain). Using DocBook and a presentation layer language in your tool chain is a useful technique, since it enables you to take advantage of all the existing algorithms for processing DocBook files, without requiring your authors to learn, or write in, DocBook.

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.

You could write:

```
procedure: Save a file
    1. Choose {Save}(menuitem) from the {File}(menuitem) menu. The {Save As}(di
```

Given this markup, we can make a rule that says when the name of a dialog box is mentioned in a step in a procedure, insert a graphic of that dialog box. To implement this, you might maintain a catalog of images that points to the file that describes the idea of the image of that dialog box. That file might then list various versions of the screen shot, perhaps different ones for different platforms. Thus the author does not have to think about graphics at all when writing, but the right screen shot gets inserted in the docs for each platform (and maybe no screen shot when the procedure is viewed on a phone).

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 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 that 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 they **Enter** key.

3. Press to confirm the selection.

This approach allows the document designer to switch out the graphics used for keys to find ones that work best on different displays or at different scales. 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, 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 you 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. 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 to the representation of 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 use 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 it subject matter, the identification of the subject will not change.

Chapter 32. Subject-based structures

Not available yet.

Chapter 33. Patterns

Not available yet.

Part IV. Languages

Table of Contents

Chapter 34. Markup

Through much of this book so far I have talked about markup without really explaining it. Markup is the way much, but not all, structured writing is done. Let's look at the alternative and the varieties of approaches to markup.

First, structured writing, in the broadest sense, is writing that applies any form of structure (any set of constraints) to writing. As such, most forms of writing on a computer, other than perhaps in a straight text editor without the use of markup, are structured writing since they all contain text in structures.

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.

I am being careful to frame the distinction here as between "markup" and "text". It is easy to slip into talking about "markup" and "content", but this is a misnomer. Markup is content every bit as much as text. As we have seen, structured writing consists of factoring variants from invariants in content. They sometimes means replacing text with markup in the document or subject domains. The structures that the markup expresses are just as vital to the information being captured as the text within those structures. Markup is content just as much as text or graphics.

Why markup?

Structured writing is writing to which explicit structure has been applied. But binary formats can express such structures just a well as markup can. Why do we so often use markup rather than binary file formats for structured writing?

There are three main reasons:

Application independence	With binary files, it is possible to obscure how the file is interpreted, making it more difficult to write other applications that can edit the file. With markup, it is generally much easier to create other applications that can read and write the same file. This supports the development of tool chains, often based on open-source tools, to implement whatever set of structured writing algorithms are important to a business.
Ad hoc structure definition	If you need to define a structure to serve a particular purpose, perhaps one that will be used for relatively few documents, then you need a simple and inexpensive way to create that structure. Markup can provide such a format. It is much easier, for instance, to define a subject-domain markup language for a particular purpose than it is to create a binary file format and a program to read and write it.
Human reading and writing	Markup makes it possible for humans to read and write files that are processable by computers. This makes it much easier to build a tool chain for different components and to integrate them to meet your needs. All markup formats make it possible for humans to read and write the files, but not all formats make

it easy. As we shall see, some formats are much easier to write in than others.

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>Moby Dick</h1>
<p>Herman Melville's <i>Moby Dick</i> is a long book about a big whale.</p>
```

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. 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 at this point. What matters is 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 content, or they include markup for inserting individual characters in a way that won't be confused with markup. HTML takes the second approach. To include a `<` character in HTML, you use another type of markup 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" us short for "less than", the name of the `<` character.)

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 content, or they include markup for inserting individual characters in a way that won't be confused with markup. HTML takes the second approach. To include a `<` character in HTML, you use another type of markup 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" us 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 &amp; 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 entities for an ampersand is &amp;amp;.</p>
```

This will display as:

> The character entities for an ampersand is &.

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:

```
<ol>
    <li>First</li>
    <li>Second</li>
    <li>Third</li>
</ol>
```

But at the same time, the numbers look like text to the writer or reader of the Markdown file, and there is no need 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:

```
<ol>
    <li>First comes 1.</li>
    <li>Second comes 2.</li>
    <li>Third comes 3.</li>
</ol>
```

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 it 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. I will have more to say about the usefulness of patterns in structure writing later.

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 plain text 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 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 languages does not describe specific concrete document structures directly. It describes abstract structures which can be named to represent structures in any domain.

XML is an example of an abstract markup language.[1] The markup in an XML file does not directly indicates things like headings or paragraphs. Instead, it indicates a set of 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 (and more specifically, in Docbook:

```
<section>
    <title>Moby Dick</title>
    <para>Herman Melville's <citetitle>Moby Dick</citetitle> is a long book abo
</section>
```

The structure described by the XML syntax here is that of an element which contains two other elements, on of which contains text, and the one of which contains a mix of text and another element. There is no separate syntax for titles or paragraphs here. Everything is an element. To define specific document domain structures, we give names to the elements.

Here, the document domain structures “section”, “title”, and “para” are defined by XML elements named, respectively, “section”, “title”, and “para”. The generic `<i>` tag from the HTML example has been replaced with them more semantically specific `<citetitle>` tag, which in DocBook means the name of a literary work. Both “i” and “citetitle” are modeled as XML elements in DocBook.

XML elements are generic abstract structures. Named elements can be used to represent any media, document, subject, or management domain structure that you want, thus allowing you to create a markup language that suits a specific purpose.

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.

[1]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 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.

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's abstract structures. Many common markup languages are instances of XML[2]. 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.

So let's 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.

By these definitions, abstract and hybrid are not language types you can actually write content in, they are languages that you can use to define other languages that are instances of them. Despite how we often use the term, you don't actually write in XML, you write in DocBook or DITA, which happen to be instances of XML.[3] Thus in terms of actually writing content, we really do have just three types of markup languages: concrete languages, instances of abstract languages, and instances of hybrid languages.

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
- A concrete language based on an abstract language (probably XML)
- A concrete language based on a hybrid language

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:

```
/**
 * Validates a chess move.
 *
 * Use {@link #doMove(int theFromFile, int theFromRank, int theToFile, int theT
 *
 * @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
 */
```

[2]Sometimes also referred to as "applications" of XML, though this usage was far more common in the days of SGML.
[3]Or to put it another way, you write in DocBook semantics using XML syntax. Alternatively, since DocBook originated in the days of XML's predecessor abstract language, SGML, you can write DocBook semantics in SGML syntax.

```
boolean isValidMove(int theFromFile, int theFromRank, int theToFile, int theToR
    // ...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. 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. (Sigh. 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 on an existing one, and coding the parser and all the other tools to interpret that language. And 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. Many versions of Markdown, including the original, contain ambiguities about how certain sequences of characters should be interpreted.

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.

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 `ingredients` and `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 bring discipline to structured writing. They drive content quality and enable efficient reliable processing with algorithms. Constraints make it easier to write good algorithms because they limit the number of permutation of structures that you have to deal with.

All markup languages have constraints. A constraint is simply something that the markup language does not let you do. With a concrete language, you get the constraints that are built into the language. Abstract languages allow you to define your own structures, and therefore your own constraints. However, as we shall see, not all languages that are extensible are also constrainable.

Showing and hiding structure

The whole point of structured writing it to create content that meets constraints and that records the constraints it meets so that it can be reliably validated, audited, and processed. For this to happen, authors need to see the structures they are creating. In the media domain a WYSIWYG interface shows your the media domain structures you are creating. 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 create. 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 (and also very laborious to write).

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 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. As we have discussed before, these kinds of displays tend to encourage authors to backslide into the media domain.

And then there are the problems that arise when you try to edit the WYSIWYG view of an XML document. Underneath is a hierarchical XML structure, but all you can see it the flat media-domain like view of the graphical editor. Editing or cutting and pasting structures you can't see can be an exercise in futility and frustration. You can learn to do it, but it is frustrating and it takes time, and even when you learn, the process is still more complicated than it should be.

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 as well as 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 reST directives are similar to XML elements, reStructuredText predefines 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 oriented 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 reST 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 been developing 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, and have been promising all along that I would eventually explain.

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 is 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".

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 in indented under the section block. This removes the need for end tags, which reduces verbosity and helps make the structure of the document visually clear.

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 reST, however, SAM does not have an extensive set of predefined blocks. It has just a few, which correspond the the basic text structures for which it provides shortcuts or concrete syntax. And SAM is designed to have a constraint mechanism, allowing you to write a schema to define what blocks and annotation are allowed in s SAM document. This will include constraining the use of the concrete syntax as well. SAM thus represents a different type of hybrid.

Also unlike reST, SAM is not intended to have its own publishing tool chain. SAM is really intended for creating subject domain languages, with just enough basic concrete document domain structures to make writing easier. SAM outputs an XML file which can then be further processed by any existing publishing tool chains by transforming it into an appropriate document domain language.

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</annotatio
</section>
```

This book is authored in SAM. Most of its examples are in SAM. I'll describe SAM more fully another chapter.

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.

Chapter 35. Lightweight Languages

The term “lightweight markup language” has arisen in recent years and seems to describe that set of markup languages which 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. The primary appeal of lightweight markup languages rests on two related phenomena.

- They have a high degree of functional lucidity at the syntactic level and often at the semantic level 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 applications 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 existing is multiple variants that have been adapted for different purposes.

Adapted for different purposes mostly means that people have created version with specific additional 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 it 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. This makes it difficult for different authors to produce document that look radically different once published, or to create overly elaborate document structures that might not translate well to different media.

It does not provide any kind of subject domain 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 remain obvious and intuitive, meaning that, within it limits, MarkDown has good functional lucidity.

Wiki markup

Another popular lightweight format is Wiki markup, introduced by Ward Cunningham in 1995 as the writing format for WikiWikiWeb, the first Wiki. 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 feature is how linking is handled. In the original WikiWikiWeb markup, and word with internal capitals was considered a "Wiki word" and instantly became a link to a page with that Wiki Word 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. Wikipedia is by far the largest and most well known Wiki. Wiki's are a preeminent 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."[http://www.wiki.org/wiki.cgi?WhatIsWiki] Like Markup, 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 they have a defining characteristic today it is probably the bottom-up architecture rather in to 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 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 considerable 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 Wiki word can have revolutionary effects on how content is created and organized. Most Wikis today use ordinary words between double square brackets for Wiki words, 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 most of 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 crate texts that are capable of self-organization, that can be assembled into meaningful collections without the imposition of any external structures.

reStructuredText

reSturcuteredText is a lightweight concrete 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 ????.

```
.. image:: images/biohazard.png
```

```
   :height: 100
   :width: 200
   :scale: 50
   :alt: alternate text
```

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 show 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.

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. 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
  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:[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]

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:

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>
  <mrow>
    <mi>x</mi>
    <mo>=</mo>
    <mfrac>
      <mrow>
        <mo form="prefix">&#x2212;<!-- − --></mo>
        <mi>b</mi>
        <mo>&#x00B1;<!-- &PlusMinus; --></mo>
        <msqrt>
          <msup>
            <mi>b</mi>
            <mn>2</mn>
          </msup>
          <mo>&#x2212;<!-- − --></mo>
          <mn>4</mn>
          <mo>&#x2062;<!-- &InvisibleTimes; --></mo>
          <mi>a</mi>
          <mo>&#x2062;<!-- &InvisibleTimes; --></mo>
          <mi>c</mi>
        </msqrt>
      </mrow>
      <mrow>
        <mn>2</mn>
        <mo>&#x2062;<!-- &InvisibleTimes; --></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. (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.)

You might will 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 as well.

LaTeX is not as lightweight a langauge as Markdown for example. It's markup is almost entirely explicit (except for paragaphs, 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. There are a number of LaTeX editors, but they tend to be the same style of side-by-side style of editors also popular for MarkDown. That is, the author writes in LaTeX syntax and a preview is generated continuously in a separate panel. 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.

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 (????) is JavaDoc. Here is the example we looked at there:

```
/**
 * Validates a chess move.
 *
 * Use {@link #doMove(int theFromFile, int theFromRank, int theToFile, int theT
 *
 * @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 theToR
    // ...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 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 a Doxygen which is used for multiple languages and Sphinx which is used for Python and others. Wikipedia maintains an extensive list: 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.

SAM

I've talked about SAM a couple of times now, and said that many of the examples in this book are written in SAM. That is true, in the same sense that some of the examples are written in XML. But this is not the same thing as saying that an example is written in reStructuredText. reStructuredText is a complex document domain markup language which also happens to have an extension facility. You can write a complete document in reStructuredText without defining any new structures.

SAM, on the other hand, while it has concrete lightweight syntax features, is not a concrete markup language out of the box. The most complex SAM document you can write without defining any structures is a single paragraph or list. (A SAM document must have a single root element, so two paragraphs in a row at the root level is an error. You need a block structure such as `topic` or `chapter` or `recipe` to contain everything and SAM does not predefine any blocks.)

SAM is designed for the same purpose as XML: for defining new markup languages. I'll get back to it in ????.

Chapter 36. 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. 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, though 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 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.

If the point of structured writing is improve the quality of content through the application of constraints, why is the structured writing landscape dominated by a few very large and quite loosely constrained markup languages?

Partly because, as I pointed out at the beginning, all writing is structured and when someone says they are moving to structured writing, what they mean is that they are adding a more structure to their writing than they had before. DocBook may not be a highly constrained language, but it is lot more constrained than Microsoft Word or InDesign and somewhat more constrained than FrameMaker.

Partly because a lot of the adoption of structured writing it not motivated primarily by content quality but by a desire to improve content management, particularly content reuse. While it is possible to do these things without resorting to structured writing for a format, structured writing format 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 is all to easy to blame 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 an framework for think 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 of 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 as what it's 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 ????. Here I will look at out-of-the-box DITA.

Out-of-the-box DITA comes in three forms (three different boxes, if you will).

1. The DITA Open Toolkit. You and 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.

3. Customized DITA systems. You may be handed a customized system created by a consultant or vendor, generally built using one of the available DITA CMS platforms. This may have been extended to provide new topic types, new output formats, or new management facilities. These could work in almost any way and it may not even be obvious that DITA has been used in their construction. Obviously there is not much that we can say about them here.

The key features of out-of-the-box DITA that will determine how well its fits with your needs are its topic model and its focus on the reuse algorithm. The description of the document-domain/management-domain approach in ???? is based on the DITA model, which provides comprehensive support in those domains.

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. 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. Specializing DITA may allows 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).

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.

But DITA generalizes this model. 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 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 DocBook map file, on the other hand, is an independent structure. It does not create a single logical document structure.

It does not 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 constraints can be applied (at least in the conventional way).

This has some useful properties. It means that the model of a map is universal. You don't need to create a new top-level model to create a new kind of top-level information set. A map can model any information set, or at least any information set that is structured hierarchically and built by specifying members by name.

Maps are structured like 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, do you create one topic with the list of four items in it or do you 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 using 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.

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 narrative blocks. In the design of a markup language, narrative blocks are made up of semantic blocks which may be made up of smaller semantic blocks. This works fine for developing the structure of a narrative block, which is the work that will be presented whole to the reader. In that scenario, the unit that the writer writes is the narrative 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 narrative 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 narrative block. However, DITA does not provide a high-level way to constrain the structure of a narrative block that is built this way.

If you want a constrained narrative 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 narrative topic often requires different types of information (as in the recipe example we have used so frequently).

This leads to some confusion about whether a DITA topic is a semantic block or a narrative 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 DITA topics are not narrative blocks. To get your narrative 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.[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])

The idea that topics 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 narrative blocks whole (perhaps with some variations in the text). It is quite a different thing to reuse semantic blocks below the level of the narrative block, particularly if it is important to you to constrain the narrative block or to apply any of the other structured writing algorithms at the level of the narrative block.

From a quality point of view, as well, the quality of content can suffer significantly if content is written in reusable units that are not properly assembled into narrative blocks for presentation. And if narrative blocks are being assembled out of smaller reusable units without proper attention to the narrative integrity or completeness and consistency of the result, quality suffers as well. If the author no longer sees, thinks, or works in the context of the narrative block, and if the structure of the narrative block is not constrained, content quality is very difficult to maintain.

DITA, as a technology, does not prevent you from working in whole narrative 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 narrative blocks to optimize for content quality.

A related note here is that in a reuse scenario, you motives for constraining the semantic block may be different from your motives for constraining the narrative block. The reasons for constraining the semantic block might be to adhere to the theory of information typing, or to enhance the composability of the reusable blocks. The reasons for constraining the narrative block might be to ensure the quality or consistency of the information presented to the user.

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 organization 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.

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 format, specifically for the purpose of minimizing constraints on the content.[http://idratherbewriting.com/2016/08/01/paligo-the-story-xml-ccms-in-the-cloud/]. 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.

DITA's sweet spot, therefore, would appear to be content reuse scenarios in which you want to place constraints on the reusable content units that follow the "information typing" model, but can live without placing constraints on the narrative blocks that are built from those units.

Note, though, that this analysis is only focused on DITA as a tool for structured writing. DITA is also a tool for content management and it role in making content management systems work effectively has to be evaluated separately. And, as pointed out above, DITA provides a high-level tool for information typing which can be used for things outside this sweet spot, and potentially having noting to do with its reuse features. How broad this range of applicability is depends largely on where the sweet spots of other tools lie. Your needs may not coincide perfectly with the sweet spot of any one tool. At that point, the right choice is the one that can be adapted for your needs at the least cost. That calculation is outside the scope of this book. I will try to point out where costs lie in each alternative, but quantifying them for individual situations is an exercise left to the reader.

Large generalized systems like DITA tend to create document types that are a mix of multiple domains. At its core, DITA is an information typing system based in the document domain. It creates various topic types by specialization of a core generic document domain "topic" type. (I'll describe specialization in a moment.)

Like other generalized languages that originated in the technical communication space, DITA also includes some subject domain structures, most for describing computer interfaces. These are generally small scale structures, often at the level of annotation in running text, such as annotations for references to parts of a windowed screen display.

One of the key focuses of DITA is content reuse. It would not be far off the mark to describe DITA as a system for reusing content as the decision to use it is almost always justified on this basis. As such, DITA contains a lot of management domain structures, including all of the management domain structures for content reuse and linking that I described earlier. The description of the document/management domain approaches to reuse and linking were based almost entirely on the way DITA does these things.

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.

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 leave 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 result, however, is an extremely complex system that 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 sometimes 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 improve content quality through the application of constraints. 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 is 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.

This is a more reliable process than if I had written the book in DocBook directly. I wrote my previous book 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.

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.

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 languages and leave the formatting the CSS sytlesheets. 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 subject domain languages written in XML. This includes both languages that are content oriented and languages that are data oriented, but from which one might still derive content using the extract and merge algorithm. Wikipedia maintains and extensive list at https://en.wikipedia.org/wiki/List_of_markup_languages.

One interesting example is BeerXML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<RECIPES>
    <RECIPE>
        <NAME>Dry Stout</NAME>
        <VERSION>1</VERSION>
        <TYPE>All Grain</TYPE>
        <BREWER>Brad Smith</BREWER>
        <BATCH_SIZE>18.93</BATCH_SIZE>
        <BOIL_SIZE>20.82</BOIL_SIZE>
        <BOIL_TIME>60.0</BOIL_TIME>
        <EFFICIENCY>72.0</EFFICIENCY>
        <TASTE_NOTES>Nice dry Irish stout with a warm body but low starting gra
        <RATING>41</RATING>
        <DATE>3 Jan 04</DATE>
        <OG>1.036</OG>
        <FG>1.012</FG>
        <CARBONATION>2.1</CARBONATION>
        <CARBONATION_USED>Kegged</CARBONATION_USED>
        <AGE>24.0</AGE>
        <AGE_TEMP>17.0</AGE_TEMP>
        <FERMENTATION_STAGES>2</FERMENTATION_STAGES>
        <STYLE>
            <NAME>Dry Stout</NAME>
            <CATEGORY>Stout</CATEGORY>
            <CATEGORY_NUMBER>16</CATEGORY_NUMBER>
            <STYLE_LETTER>A</STYLE_LETTER>
            <STYLE_GUIDE>BJCP</STYLE_GUIDE>
            <VERSION>1</VERSION>
            <TYPE>Ale</TYPE>
            <OG_MIN>1.035</OG_MIN>
            <OG_MAX>1.050</OG_MAX>
            <FG_MIN>1.007</FG_MIN>
            <FG_MAX>1.011</FG_MAX>
            <IBU_MIN>30.0</IBU_MIN>
            <IBU_MAX>50.0</IBU_MAX>
            <COLOR_MIN>35.0</COLOR_MIN>
            <COLOR_MAX>200.0</COLOR_MAX>
            <ABV_MIN>3.2</ABV_MIN>
            <ABV_MAX>5.5</ABV_MAX>
            <CARB_MIN>1.6</CARB_MIN>
            <CARB_MAX>2.1</CARB_MAX>
            <NOTES>Famous Irish Stout.  Dry, roasted, almost coffee like flavor
        </STYLE>
        <HOPS>
            <HOP>
```

```
                <NAME>Goldings, East Kent</NAME>
                <VERSION>1</VERSION>
                <ALPHA>5.0</ALPHA>
                <AMOUNT>0.0638</AMOUNT>
                <USE>Boil</USE>
                <TIME>60.0</TIME>
                <NOTES>Great all purpose UK hop for ales, stouts, porters</NOTE
            </HOP>
        </HOPS>
        <FERMENTABLES>
            <FERMENTABLE>
                <NAME>Pale Malt (2 row) UK</NAME>
                <VERSION>1</VERSION>
                <AMOUNT>2.27</AMOUNT>
                <TYPE>Grain</TYPE>
                <YIELD>78.0</YIELD>
                <COLOR>3.0</COLOR>
                <ORIGIN>United Kingdom</ORIGIN>
                <SUPPLIER>Fussybrewer Malting</SUPPLIER>
                <NOTES>All purpose base malt for English styles</NOTES>
                <COARSE_FINE_DIFF>1.5</COARSE_FINE_DIFF>
                <MOISTURE>4.0</MOISTURE>
                <DIASTATIC_POWER>45.0</DISASTATIC_POWER>
                <PROTEIN>10.2</PROTEIN>
                <MAX_IN_BATCH>100.0</MAX_IN_BATCH>
            </FERMENTABLE>
            <FERMENTABLE>
                <NAME>Barley, Flaked</NAME>
                <VERSION>1</VERSION>
                <AMOUNT>0.91</AMOUNT>
                <TYPE>Grain</TYPE>
                <YIELD>70.0</YIELD>
                <COLOR>2.0</COLOR>
                <ORIGIN>United Kingdom</ORIGIN>
                <SUPPLIER>Fussybrewer Malting</SUPPLIER>
                <NOTES>Adds body to porters and stouts, must be mashed</NOTES>
                <COARSE_FINE_DIFF>1.5</COARSE_FINE_DIFF>
                <MOISTURE>9.0</MOISTURE>
                <DIASTATIC_POWER>0.0</DISASTATIC_POWER>
                <PROTEIN>13.2</PROTEIN>
                <MAX_IN_BATCH>20.0</MAX_IN_BATCH>
                <RECOMMEND_MASH>TRUE</RECOMMEND_MASH>
            </FERMENTABLE>
            <FERMENTABLE>
                <NAME>Black Barley</NAME>
                <VERSION>1</VERSION>
                <AMOUNT>0.45</AMOUNT>
                <TYPE>Grain</TYPE>
                <YIELD>78.0</YIELD>
                <COLOR>500.0</COLOR>
                <ORIGIN>United Kingdom</ORIGIN>
                <SUPPLIER>Fussybrewer Malting</SUPPLIER>
                <NOTES>Unmalted roasted barley for stouts, porters</NOTES>
                <COARSE_FINE_DIFF>1.5</COARSE_FINE_DIFF>
                <MOISTURE>5.0</MOISTURE>
                <DIASTATIC_POWER>0.0</DISASTATIC_POWER>
                <PROTEIN>13.2</PROTEIN>
                <MAX_IN_BATCH>10.0</MAX_IN_BATCH>
            </FERMENTABLE>
        </FERMENTABLES>
        <MISCS>
            <MISC>
                <NAME>Irish Moss</NAME>
                <VERSION>1</VERSION>
                <TYPE>Fining</TYPE>
                <USE>Boil</USE>
                <TIME>15.0</TIME>
                <AMOUNT>0.010</AMOUNT>
                <NOTES>Used as a clarifying agent during the last few minutes o
            </MISC>
        </MISCS>
        <WATERS>
            <WATER>
                <NAME>Burton on Trent, UK</NAME>
                <VERSION>1</VERSION>
                <AMOUNT>20.0</AMOUNT>
                <CALCIUM>295.0</CALCIUM>
                <MAGNESIUM>45.0</MAGNESIUM>
                <SODIUM>55.0</SODIUM>
                <SULFATE>725.0</SULFATE>
                <CHLORIDE>25.0</CHLORIDE>
                <BICARBONATE>300.0</BICARBONATE>
                <PH>8.0</PH>
                <NOTES> Use for distinctive pale ales strongly hopped.  Very ha
                </NOTES>
            </WATER>
        </WATERS>
        <YEASTS>
            <YEAST>
                <NAME>Irish Ale</NAME>
                <TYPE>Ale</TYPE>
                <VERSION>1</VERSION>
                <FORM>Liquid</FORM>
                <AMOUNT>0.250</AMOUNT>
                <LABORATORY>Wyeast Labs</LABORATORY>
                <PRODUCT_ID>1084</PRODUCT_ID>
                <MIN_TEMPERATURE>16.7</MIN_TEMPERATURE>
                <MAX_TEMPERATURE>22.2</MAX_TEMPERATURE>
                <ATTENUATION>73.0</ATTENUATION>
                <NOTES>Dry, fruity flavor characteristic of stouts.  Full bodie
                <BEST_FOR>Irish Dry Stouts</BEST_FOR>
                <FLOCCULATION>Medium</FLOCCULATION>
            </YEAST>
        </YEASTS>
        <MASH>
            <NAME>Single Step Infusion, 68 C</NAME>
            <VERSION>1</VERSION>
            <GRAIN_TEMP>22.0</GRAIN_TEMP>
            <MASH_STEPS>
                <MASH_STEP>
                    <NAME>Conversion Step, 68C </NAME>
                    <VERSION>1</VERSION>
                    <TYPE>Infusion</TYPE>
                    <STEP_TEMP>68.0</STEP_TEMP>
                    <STEP_TIME>60.0</STEP_TIME>
                    <INFUSE_AMOUNT>10.0</INFUSE_AMOUNT>
                </MASH_STEP>
            </MASH_STEPS>
        </MASH>
    </RECIPE>
</RECIPES>
```

As you can see, this is a recipe language, but a much more detailed and precise recipe language than any that we have looked at before. It is a recipe language for beer and for beer only. It takes a lot of information that might normally be written in paragraphs and breaks it up into precisely labeled fields. This means, of course, that it is far more constrained than generic recipe markup. You can do far more conformance testing on this recipe than you could on a normal one, and manipulate and query the information in far more ways that you could with a conventional recipe format.

Chapter 37. Extensible and Constrainable Languages

The languages we have look 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. Choosing one of them makes sense if the constraints they express and the algorithms they support are the ones that matter most to your organization, and they support them sufficiently well to meet your needs. If not, you will need to create your own structures. Rather than starting from scratch, you should consider using and abstract and/or extensible markup language as you starting point.

If no existing language meets your needs, you can create your own. You can either start from scratch or you can extend or constrain an existing language.

A word on the distinction between extension and constraint. Extension is the ability to add new tags to a language. Constraint is the ability to set limits on where and in what relationships those tags may occur.

The tools you use for developing your language may separate these functions or unite them in one operation. For example, the XML schema language RelaxNG will allow you to specify a set of element names and the order and relationship in which they can occur. This does constraint and extension in one operation.

The schema language Schematron, on the other hand, is purely a constraint mechanism. With a Schematron schema, everything is allowed unless it violates a specified constraint.

If you start with an existing concrete language, it has its inherent constraints: the tags and the order and location in which you are allowed to use them. If you want to add additional tags, that is extension. If you want to limit or remove the use of existing tags, or if you want to limit the placement of new tags, that is constraint. Existing concrete languages may have mechanism for extension and for constraint, or for one but not the other.

XML

The X in XML stands for eXtensible, but, as we noted in ???? XML is an abstract language that does not define any document structures itself. Extension in XML, therefore, is extension from zero. Starting from scratch. You define a new XML tagging language using one of the several available schema languages.

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 is actually 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.

XML schema languages are information typing languages. Their sole function is to define information types. So what does an information typing architecture provide over and above what an information typing 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 DITA's case, the architecture consists of a set of predefined tagging languages that are intended as a basis for extension through a mechanism known as specialization. It also includes 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. 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.

Inherent in the process of constructing an architecture is that you constrain the field in certain ways. Architectures move functionality to a higher level by choosing some options and rejecting other. I said that 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. Describing banking transactions 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. 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 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 we have noted many times, many of the structured writing algorithms work 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, 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?

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 types of 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 will have 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 topics 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 is was specialized from. So `knitting-pattern` "is-a" topic.

But DITA specialization is different from naming elements in XML in a number of ways.

First, the base DITA topic type is not an abstraction like and XML element. You cannot instantiate an element without giving it a name. The base DITA topic type, on the other hand, is a fully implemented topic type that you can instantiate directly. You can, and people do, write directly in the base topic type. We noted in the discussion of 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 for 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 necessarily 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 and is-a relationship to the element it specializes. Thus a procedure element might be a specialization of an ordered list elements and its step elements might be specializations of list items. (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 as "1." in a plain list.)

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 type 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 not other processing were specified for them). Then you might decide that you wanted 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` is provided.

The value of the specialization model for creating new topic types is a matter of disagreement, some holding that it is an important breakthrough and other not seeing any practical advantage over other approaches.[1]

The value of the fall back processing is also disputed. Usually when you impose a constraint in structured writing, it is to support one or more structured writing algorithms. What then is the point of creating the structures and not the corresponding algorithms. Two potential reasons are to impose constraints on authors, or to improve functional lucidity by more appropriate labeling of elements, without needing any changes in processing, and to facilitate exchange of content between organization who use different specializations.

The second 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.

As a generality, the less 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 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 of course.

Can you specialize from the document domain to he subject 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? 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.

[1]I am in the camp that doubts the value of specialization (or else I would probably not have brought it up). I prefer an approach that builds models up out of smaller models: building narrative blocks out of a collection of reusable semantic blocks. In other words, I prefer to create models the way DITA creates documents, not the way it create models.

How big a deal it is if your specialization lacks a true is-a relationship to the thing is it specialized from? If you ignore the fallback mechanism and supply a full set of processing code for your specialization, it may not be a big deal at all, though in these circumstances, you will have done exactly the same work -- writing a schema and one or more processing algorithms -- as if you had created the type from scratch, only with the additional overhead of describing the specialization relationships of each element in your new schema.

Where specialization could save you work over writing from scratch is if you do a light specialization, that is, on that only modifies a few elements. Then you don't have to worry about designing the rest of the topic type and you only have to write new code for he elements you specialize. The rest you can inherit from the base type. Similar savings can also be achieved in other systems by building new models from predefined modules of structure and code.

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'a specialization. What it does provide is a deliberately modular construction that makes it easy to create new schemas that include elements from DocBook. In short, DocBook takes full advantage of the extensibility features built into XML schema languages.

Does the fact that does not invent its own schema language 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 and constraint mechanism, DocBook is as extensible as it is possible for any XML vocabulary to be.

Where it differs from DITA is that DocBook extensions are not DocBook and cannot be processed by standard DocBook tool chains. 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, but potentially holding up the entire production process) then DocBooks extension mechanism will give you that.

It is fair to say though, that the DITA approach can make the writing of algorithms easier in one way. It is easier to modify code that works than to fix code that is broken. With DITA's fallback mechanism you can develop the algorithm for processing the specialization by running the default code, observing the result, and then adding your own processing rules one at a time, running the process again after each change to validate the effect. You can repeat this process until the output formats correctly.

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 provided for just about every document structure out there, so if you are building a document domain language, chances re 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 therefor be published by the DocBook tool chain. You will not have to write any algorithms at all if you take this approach. Creating a subject 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

Restructured text defines 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 RestrucuredText. 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.

This is not so bad for a language which is only processed in one way. Since any validation of markup requires running code, running the processor is no more work than running a validator.

But what if you want your language to be processed by more than one processing algorithm. This can easily be the case if you are implementing multiple structured writing algorithms, since each requires their own implementation code. In this case you need an independent standard for determining the validity of the input.

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 be 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 a processor fails to process a given input file? Is the processor incorrect or the source file.

A schema create a language definition that is independent of any processor. 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 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.) 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 ????.

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

As you can see, 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 been developing SAM, the markup language used for most of the examples in this book. (The book was also written in SAM.)

As described in ????, 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 for common features such as insertions, citations, and variable definitions.

And as I mentioned in ????, SAM is not a standalone concrete language. It is designed, like XML, for defining specific tagging languages. However, all languages defined in SAM share a small common base set of document related structure for which SAM provide concrete syntax. This allows sam to combine lightweight syntax of the most common document structures with the ability to define specific constrained markup languages for particular purposes, particularly subject domain languages.

So when I say that this book was written in SAM, what I mean was that it was written in a specific tagging language based on SAM that I created for the purpose of writing this book. It is a mostly document-domain language, because discursive texts like this one don't offer a lot of scope for useful subject domain constraints, but it make significant use of subject domain annotation, which was used for auditing, linking, cross referencing, formatting, and indexing purposes.

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 elements 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 it schema language should also be simpler and make it much easier for writers to develop their own SAM-based markup languages.

SAM (Semantics Authoring Markdown) is a hybrid markup language that I created because I got frustrated writing in XML and because I want to write in the subject domain and all the other concrete markup languages and hybrid markup languages I can find are principally or wholly document domain languages. Also, I don't know of any hybrid languages that are constrainable, and I wanted a hybrid language that is constrainable. (Again, because I want to work in the subject domain where constraints are particularly important.) This book is written in SAM.

SAM is the language used for the majority of the examples in this book. I have not explained it fully until now because SAM is designed to make structure clear and that is all I have needed to do in most examples. Naturally, to write in SAM you would need to know more about the rules, 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 the the markup expresses the document structure they create in a way that is readable.

SAM has the same goal, except that SAM was designed 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 reST 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 many ways it is similar to a markup language called YAML, which is designed for data rather than documents.

Here is an example of YAML courtesy of Wikipedia:[https://en.wikipedia.org/wiki/YAML#Sample_document]

```
receipt:     Oz-Ware Purchase Invoice
date:        2012-08-06
customer:
    first_name:   Dorothy
    family_name:  Gale

items:
    - part_no:   A4786
      descrip:   Water Bucket (Filled)
      price:     1.47
      quantity:  4

    - part_no:   E1628
      descrip:   High Heeled "Ruby" Slippers
      size:      8
      price:     133.7
      quantity:  1

bill-to:  &id001
    street: |
            123 Tornado Alley
            Suite 16
    city:   East Centerville
    state:  KS

ship-to:  *id001

specialDelivery:  >
    Follow the Yellow Brick
    Road to the Emerald City.
    Pay no attention to the
    man behind the curtain.
```

Key features of YAML are the use of names ending in colons to introduce blocks, and the use of indentation to indicate the hierarchy of the document. SAM uses the same principles.

```
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.
```

Chapter 38. Ad hoc languages

Not available yet.

Part V. Tools

Table of Contents

Chapter 39. Monoliths vs tool chains

Not available yet.

Chapter 40. Editors and development environments

Not available yet.

Chapter 41. Parsers and processing tools

Not available yet.

Chapter 42. Static site generators

Not available yet.

Chapter 43. Content management systems

Not available yet.

Chapter 44. DITA-OT

Not available yet.

Chapter 45. SPFE

Not available yet.

Chapter 46. Buy vs build

Not available yet.

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] or call us at (970) 231-3624.

Appendix A. Copyright and Legal Notices

Credits

Cover Image:	Wall Assemblage – Copyright © 2006 Eric Lin on flickr (CC BY-SA 2.0)
Cover Background:	Another Brick in the Wall – Copyright © 2009 Coun2rparts on flickr (CC BY-SA 2.0)
Foreword:	Scott Abel, The Content Wrangler

Disclaimer

Trademarks