

WORDS



A Quarterly Bulletin for Technical Writers & Communicators

Volume 2 | Issue 2 | May 2010

The beginning

- More and more technical writers working in the software industry are finding themselves enveloped in a relatively new software development methodology: *agile development*. The name is slightly belittling of earlier methodologies. Further, proponents of agile development often support it in a way that is somewhat inaccurate: by comparing it with a methodology that was rarely, if ever, adopted. This is the pure waterfall methodology, where one phase of a project was completed before another phase began. I suspect that there would be few, if any, project managers who directed pure waterfall projects. Technical writers rarely, if ever, stood on their hands waiting for the documentation plan to be signed off before proceeding to the next phase. No, they got on with creating the required templates on the assumption that the, or a, documentation plan would get signed off—eventually. Likewise, writers would pass chapters to reviewers before the entire first draft was finished. This is *parallel* development rather than waterfall development.

And it was much the same with programming. Some features were usually being coded while others were still being designed. And QA didn't wait until all the coding was completed. There was, in other words, always parallel development. The pure waterfall method was always little more than just a mental construct.

Be that as it may, agile is the way many software companies are going, and its atomisation of the development process poses particular challenges to technical writers, as Dan Smith shows in this issue's leading article.

- Dave Reynolds continues his useful series of articles on how FrameMaker can be customised to better match our personal work practices. And Dave Gash explains the simplicity of repurposing chunks of text built on the DITA model of structured authoring: it's all a matter of attribute setting and filtering.
- And here's something unexpected in a journal on technical communication: poetry. Many thanks to those intrepid folk who applied their poetic muscle to the dull old subject of technical writing.

Geoffrey Marnell

Editor [geoffrey@abelard.com.au]

Contents

Documenting in an agile environment.....	1
Custom keyboard shortcuts in FrameMaker	4
Introduction to DITA conditional processing.....	7
Language: the core skill in technical writing	11
Journals.....	14
Book review.....	15
Poetry.....	18
Miscellany	20
Mindstretchers	21

Documenting in an agile development environment

Dan Smith

Agile project management methodologies are designed to improve on the traditional waterfall project management model. The waterfall model is so named because a project's development stages are like a sequence of rock pools in a waterfall. Each stage is dependent on the previous stage. Development on one stage does not start until the previous stage's rock pool is full and resources can flow onwards. You set up timelines and milestones, and monitor a project's progress against these.

The agile software development environment, by contrast, is designed for flexibility. The idea is that project management can react quickly to changing situations and changing customer requirements while, at the same time, keeping focused on concrete goals. The methodology is designed to deliver functionality to testers, the customer and users as early as possible, and to react to feedback.

Although the agile project management methodology is well defined, the ways in which it is implemented vary widely across organisations. Usually, the development resources are broken up into small teams of between 3 and 6 developers—known as *scrum* teams—charged with completing a chunk of work. The small-team nature of the process encourages the team members to work closely together, and to exchange ideas and suggestions for resolving problems.

The functionality that a scrum team is responsible for is broken up into user stories and the tasks required to deliver the functionality that each user story requires. A user story is typically a chunk of functionality. For example, in a project to develop a stock ordering system, one user story could be to provide the ability to process orders. Typically in an agile environment the user story would be framed thus: “As a stock clerk, I need to enter purchase request details into the stock and ordering system. For each request I enter, the system checks for existing customer records. The system either displays existing customer details, or provides an input mechanism so that I can enter the new customer details”. Tasks for this story might include:

- validate the input
- check against the customer database and retrieve existing details, if there are any
- provide a way to record the details if the customer is new.

The project development time-span is broken up into two or three week intervals, called *sprints*. At the start of each sprint there is a planning meeting at which each team member commits to the tasks that he or she will complete in the sprint.

Each day, the development team holds a short meeting called a *scrum meeting*. At the meeting, each developer reports briefly on what he or she did in the last 24 hours, what he or she will do in the next 24 hours, and reports any impediments. From these meetings, the scrum leader can monitor the development process, identify any problems and juggle resources. In line with the ideal of keeping scrum meetings brief, co-located team members often stand during these meetings.

At the end of each sprint, the team holds a review meeting at which each team member demonstrates the functionality they have produced. Based on these demonstrations, the scrum leader decides whether a story can be accepted or if it needs more work (in which case it is carried over to the next sprint). At this review meeting, team members discuss what went right and what went wrong during the sprint.

At the extreme end of the agile environment, a scrum team sits in a dedicated room, often at two rows of desks facing each other. There is a couch in a corner where members can take time out, think through problems or discuss some point or other. The technical writer sits in the scrum room and interacts with the team and contributes to the process.

At the other end of the spectrum, scrum team members are in different locations and meet only by instant messaging, emails or teleconferences. An agile project management environment can vary between these two extremes.

From a documentation perspective, the agile development process requires some coordinating leg-work to ensure documentation consistency across a project. Methodologies such as DITA—that closely define content structure, and hence appearance—are a help, and the organisation needs a well-policed, unambiguous writing style guide. It helps to have a dedicated editor, regular writer team meetings and a strong peer review process in place.

The coordination of a major software release I worked on recently, which had ten or so writers working in scrum teams spread across six sites in the US and Europe, worked as follows:

- The project had a documentation lead who maintained a project-wide documentation perspective. She kept track of where each scrum team fitted into the project, what each team’s deliverables were, and each team’s documentation requirements. The documentation lead also attended the development and marketing project meetings. She kept track of the release’s documentation requirements and due dates, and was kept aware of date changes and any changes in the project’s direction. She also communicated any relevant documentation issues to release management.

Effective Onscreen Editing: new tools for an old profession

Editors are increasingly being asked to edit on the screen using a word processor, but most are finding it challenging to transfer their skills to editing with a word processor. *Effective Onscreen Editing* teaches the basics you need to learn to make the transition, plus proven tips and tricks to maximise your productivity and effectiveness. The book describes general principles valid for any software, then illustrates the principles using Microsoft Word to make them more concrete.

Available as a printed book or as an eBook optimised for onscreen reading.

Learn more at the book’s Web page:
<http://www.geoff-hart.com/books/eoe/onscreen-book.htm>

 **Diskeuasis
Publishing**

- The project also had a help-build lead, responsible for producing the interim documentation outputs. This involved ensuring that the build processes for the online help and context-sensitive help were in place and were integrated with the software builds. He also coordinated the process whereby writers could install the latest software build and verify that their help components were in place and correct.
- The project-wide documentation work was organised into documentation sprints. The scrum leader (not the documentation lead) coordinated the scrum meetings, took minutes and kept records. At the daily scrum meeting, each writer would report their progress and any issues they had. These meetings were useful for resolving any style issues and discussing any cross-team coordination requirements. They ensured that everyone knew what everyone else was doing.
- At the review meeting at the end of each sprint, each writer would share their desktop with the group and demonstrate their completed stories. The group would provide feedback and each story would be either accepted or carried over to the next sprint. The review meeting was followed by a planning meeting where the stories and tasks to be covered in the next sprint were defined and allocated.

So what is it like for a technical writer working in an agile project management environment? In my experience it is a mixed bag of positives and negatives. While the focused nature of the scrum environment can have benefits for the development process, it is not necessarily as beneficial for documentation. The development scrum leader is usually a developer or at least from a development background. Depending on how involved in the documentation the development scrum leader wants to be, and how strict in its adherence to the agile method the team is, development scrum leaders can expect technical writers to adhere strictly to the agile guidelines like the developers do. If you are dedicated to a single scrum team, you as a technical writer can sometimes be expected to develop the documentation only for the user stories covered in each sprint. Or, in some situations, you can be expected to cover the functionality delivered in the previous sprint. It can sometimes require effort to convey the differences between developing functionality and developing documentation for the functionality.

While you can be located in your prime project's scrum room, you as a writer are also involved in the documentation scrum, along with scrums for internal documentation projects. The intense working environment of a scrum room means that some

developers don't like it when you join other scrum meetings by teleconference. Even when you believe they will benefit from the experience if they would open their minds even slightly, some developers have been known to claim that it affects their concentration when you embark, say, on a heated defence or repudiation of the use of semicolons or the use of initial capitals in headings.

For projects where localisation is required, the often changing nature of the agile environment can mean that you need to have content re-translated to accommodate changing functionality. You need to negotiate this with your localisation company upfront, and check localisation contracts for related penalty clauses that will affect costs. Localising in an agile environment can provide many good arguments for implementing a content management system that manages the localisation process and provides functionality for flagging changed content for re-translation.

On the positive side, it can be very productive to be sitting with the developers. It is easier to maintain a rapport with developers after participating in their Monday morning sharing sessions, where they discuss and compare their experiences over a wild weekend. It is also good to be in the thick of the development process—you get a feel for what is going on in the project and what is changing. The sheer proximity means that it is much easier to get information from developers. You can ask questions to clarify some point or other and immediately get the information you need.

It is also easier to demonstrate your worth when it comes to usability and user interface design. I find that developers take your suggestions more seriously in a scrum room environment, and sometimes even ask you for advice. Generally, developers are more likely to regard you as a useful team member rather than as someone who takes up their valuable time with questions, and lumps on them content that must be reviewed.

To sum up, the agile project management model is more realistic than the waterfall method (where you set milestones that you know right from the start are going to shimmer and transmute like mirages in the desert). As a writer, you get an intimate, bird's eye view of the development process, and this is a useful experience for many reasons. So while working in a scrum room might not be something you would want to do forever, it is definitely an experience that you would not want to miss out on entirely.

Dan Smith

Dan Smith is an expatriate Australian working in the UK. He has many years experience as a technical writer, working with large technology companies in Australia, Italy, France and the UK.

Custom keyboard shortcuts in FrameMaker

Dave Reynolds

I'm a great believer in using keyboard shortcuts as much as possible when working at the computer. I do this for two main reasons: to reduce the chances of getting Repetitive Strain Injury (RSI) or Occupational Overuse Syndrome (OOS) through using the mouse, and because it speeds up operations that would otherwise require navigating through menus or even layers of menus.

FrameMaker provides the means to set up your own keyboard shortcuts. Some people may be put off doing this because you can't do it via the FrameMaker GUI, but have to write code in a configuration file. However, it is not really that difficult. Once you've got your first simple keyboard shortcut working, I'm sure you'll see the benefits and will be encouraged to try some more.

This tutorial explains how to create custom keyboard shortcuts in FrameMaker. This is done by creating a new configuration file called `customui.cfg` which contains simple code statements to apply key sequences to standard FrameMaker commands.

The `customui.cfg` file is read last by FrameMaker when it starts up. The shortcuts defined in this file then override any standard FrameMaker or Windows keyboard shortcuts.

Constraints

I have used this method successfully in FrameMaker 4, 5, 6 and 8. I have not tested it in FrameMaker 7 or 9, but I suspect there will be no problems similarly customising keyboard shortcuts in those versions. I have also successfully used my FrameMaker 6 `customui.cfg` file with my current FrameMaker 8 installation.

Preparation

Before attempting to create your own custom keyboard shortcuts, I suggest you look through the *Customizing Frame Products* reference guide included in the FrameMaker installation. You should also become familiar with `cmds.cfg`, the file that contains information on all FrameMaker commands. In particular, you need to know the exact internal name of any command you want to apply a keyboard shortcut to.

Customizing Frame Products reference guide

If you have a default installation of FrameMaker 8, the *Customizing Frame Products* reference guide can be found at `C:\Program Files\Adobe\Adobe FrameMaker 8\Documents`.¹ The file is called

`Customizing_Frame_Products.pdf` and contains much useful background information on customising various aspects of FrameMaker. I strongly recommend that you read the "Adding key sequences" section on page 28.

The `cmds.cfg` file

Because the `customui.cfg` file modifies standard FrameMaker commands, you first need to know which standard commands you want to add shortcuts to and the internal name of each of them. I suggest you make a list of the commands and the menus they are on before proceeding.

You can find the internal name of a command by looking in `cmds.cfg`. This file, which contains information about all the FrameMaker commands, is in the `configui` sub-folder. If you have a default installation of FrameMaker 8, the sub-folder is at:

```
C:\Program Files\Adobe\FrameMaker8\
fminit\configui
```

Navigate to that folder and open `cmds.cfg` in a text editor or other suitable program. (*WordPad* is suited to this task, more so than *Notepad*.)

As you page through the file you will see that the commands are grouped by menu. Figure 1 below shows some of the commands grouped under the **File** menu:

```
*** File Menu ***

<Command NewDocument
  <Label Document...>
  <KeySequence \!fn>
  <Definition \x300>
  <Mode All>>

<Command NewBook
  <Label Book>
  <KeySequence \!fn>
  <Definition \x308>
  <Mode All>>

<Command RepeatNew
  <Label Repeat Last New>
  <Definition \x31d>
  <Mode All>>
```

Figure 1: Default `cmds.cfg` file

Note that the internal name of a command is the string of words to the right of the word **Command** (such as **NewDocument** in Figure 1). The word or words that appear on the menu is the string to the

1. In earlier versions, the reference guide can be found in the *Online Manuals* sub-folder.

right of **Label**, and the default keyboard shortcut, if there is one, is the specified **KeySequence**. For the purposes of this tutorial, the other information in the file can be ignored.

! FrameMaker uses `cmds.cfg` when it is running, so do not modify it. Make a copy of it if you want to play around with it.

This file is an essential reference. I found it very useful to have a printed copy when first setting up my `customui.cfg` file. So I copied the whole file into a two-column FrameMaker document, set the point size fairly small, and printed it out in about 32 pages.

Method

Which keys can I use in shortcuts?

Any key on the keyboard can be used in a custom keyboard shortcut. Most keys are represented in a custom shortcut by their actual character, the character you see on the keyboard. But some keys need special treatment. For example, the following keys are wholly or partly spelt out and preceded by a forward slash (/):

```
/Up, /Down, /Left, /Right, /Home,  
/End, /PgUp, /PgDn, /Return, /Tab,  
/BkSp, /Space, /Delete, /Insert, /F1  
through to /F16, /Apps and /Escape.
```

Modifier keys are represented thus:

+ for SHIFT, ^ for CONTROL, ~ for ALT and \! for ESC.

The other keys are represented by their actual characters.

Note that the Apps key (also known as the **MENU** or **APPLICATION** key) is used like the **ESC** key: press and release, and then press the next key.

Syntax

The syntax for the statements in `customui.cfg` that assign keyboard shortcuts to commands is as follows:

```
<Modify CommandName <KeySequence keys>>  
<Modify CommandName <KeySeqLabel menu text>>
```

The first line specifies the command being modified (*CommandName*) and the keyboard shortcut being assigned to it (*keys*). The second line specifies the command being modified (*CommandName*) and the label for the keyboard shortcut that is to appear on the menu (*menu text*).¹ Note that each statement and embedded statement is enclosed within angle brackets. These brackets are essential.

Any text outside angle brackets is treated as a comment. Hence you can add comments throughout

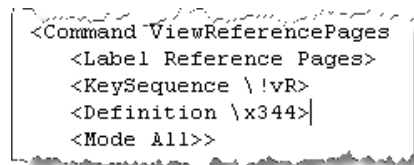
1. You can omit the second statement if you don't want the keyboard shortcut added to the menu.

the file to explain what you've done, or to group modified commands under headings.

Figure 1 on page 4 shows the components of each typical command in `cmds.cfg`. When creating a custom shortcut for a command in `customui.cfg`, you must use the same name for the command as it appears in `cmds.cfg`. Note that names are case-sensitive. See *Customizing_Frame_Products.pdf* for more information.

Create the shortcut

1. Decide on a command to assign a keyboard shortcut to. In this example, we will create a shortcut for viewing reference pages (for which the standard menu sequence is **View > Reference Pages**).
2. Open `cmds.cfg` and locate the details for this command. Since the command in question is on the **View** menu, its details will be listed under the heading ***** View menu *****. The details are shown in Figure 2 below.



```
<Command ViewReferencePages  
<Label Reference Pages>  
<KeySequence \!vR>  
<Definition \x344>  
<Mode All>>
```

Figure 2: Command details—View Reference Pages

3. Note the exact name of the command, namely, `ViewReferencePages`.
4. Create a new text file and save it as `customui.cfg` into the same folder as `cmds.cfg`.
5. Enter the following text into `customui.cfg`:

```
<Modify ViewReferencePages <KeySequence ^r>>  
<Modify ViewReferencePages <KeySeqLabel  
Ctrl+R>>
```

This assigns the key sequence **CTRL+R** to the **View > Reference Pages** command, and adds the text **Ctrl+R** beside **Reference Pages** on the **View** menu.
6. Save the file. If you are not using a text editor, make sure you save the file in text-only format.
7. Run FrameMaker and check that pressing **CTRL+R** displays the reference pages. Check too that the text **Ctrl+R** appears beside **Reference Pages** on the **View** menu.

Note that FrameMaker loads `customui.cfg` automatically when it starts. However, if you change the file while FrameMaker is running and want to see the effects without restarting FrameMaker, select **View > Menus > Modify**, open the `configui` folder, select `customui.cfg` and click **Modify**.

Conclusion

The key combinations you use for your custom shortcuts are entirely up to you. You may decide to use the initial letter of the command, a key combination that is physically convenient or a Function key. You may also decide not to override standard Windows or FrameMaker shortcuts. For example, when first setting up a template, we constantly swap between body, master and reference pages. It might be convenient to assign **CTRL + R** to reference pages and **CTRL + B** to body pages, using the initial letters as a reminder. But we might not want to assign **CTRL + M** to master pages as this keyboard shortcut is already used to display the Paragraph Designer. Instead, we might choose **CTRL + F** because **F** is between **R** and **B** on the keyboard and it is convenient to keep like keys together.

When you are satisfied with what you've done, I recommend you print `customui.cfg`, date it and file it as a reference.

I also have a list of custom keyboard shortcuts taped to my monitor in case I forget some of the less frequently used ones.

More examples

Here are some more examples of possible shortcuts.

Using the Ctrl key

To open the Cross Reference dialog

```
<Modify CrossReference <KeySequence ^h>>
<Modify CrossReference <KeySeqLabel Ctrl+H>>
```

To move to the previous or next screen

```
<Modify GotoPreviousScreen <KeySequence ^/PgUp>>
<Modify GotoNextScreen <KeySequence ^/PgDn>>
```

Ctrl + Shift keys

To open the Resize Selected Columns dialog

```
<Modify TableResizeColumns <KeySequence ^+r>>
<Modify TableResizeColumns <KeySeqLabel Ctrl+Shift+R>>
```

To apply change bars to selected text

```
<Modify StyleChangeBar <KeySequence ^+b>>
<Modify StyleChangeBar <KeySeqLabel Ctrl+Shift+B>>
```

To move the insertion point around table cells

```
<Modify MoveIPToCellAtRight <KeySequence ^+/Right>>
```

```
<Modify MoveIPToCellAtLeft <KeySequence ^+/Left>>
<Modify MoveIPToCellAbove <KeySequence ^+/Up>>
<Modify MoveIPToCellBelow <KeySequence ^+/Down>>
```

Function keys

To zoom in or out

```
<Modify ZoomIn <KeySequence /F2>>
<Modify ZoomOut <KeySequence /F3>>
<Modify ZoomFitPageInWindow <KeySequence /F4>>
```

To insert some frequently used special characters

```
<Modify CharNonBreakHyphen <KeySequence /F5>>
<Modify CharThinSpace <KeySequence /F7>>
```

Application key

To change capitalisation without using the buttons on the Quick Access bar

```
<Modify UpperCaseText <KeySequence /Apps u>>
<Modify LowerCaseText <KeySequence /Apps l>>
<Modify InitialCapsText <KeySequence /Apps i>>
```

Dave Reynolds

Dave Reynolds is a senior technical author at Tait Electronics in Christchurch, New Zealand. He has been with the company since 1985, during which time he has worked on numerous documents relating to the company's two-way radio equipment.



Introduction to DITA conditional processing

Dave Gash

One of DITA's primary strengths is combining discrete data chunks into cohesive documents. But it also excels at the other end of the spectrum—separating data chunks when necessary. This feature, called *conditional processing*, allows you to produce separate documents for different products, platforms, audiences, and more, all from the same input. This article introduces you to conditional processing and its control mechanism: *metadata*.

What is DITA?

Just kidding! Every DITA-related article in the world seems to start with this section, whether it's needed or not. I'm pretty sure that if you don't know what DITA is, you aren't even reading this article. Movin' on.

DITA metadata

Try to say that five times fast.

Let's first consider a basic DITA Open Toolkit build process. A build file collects information from a ditamap file, which in turn references a group of topic files. The build file also locates a set of XSL transforms appropriate to the requested output type, and sends all this along to the DITA Open Toolkit, which collects the topics, applies the transforms and produces the output.



**MACQUARIE
DICTIONARY
ONLINE**

www.macquariedictionary.com.au

Subscribe to the complete Macquarie Dictionary online, updated annually with new words and definitions.

Also available online is the full Macquarie Thesaurus - that perfect word is just a click away.

Try it out now for FREE!

Macquarie Online is offering free extended trial access. Simply contact Macquarie Online to set up your 3 months free access. Quote code: 3mfTrialAC

Macquarie Online Support
phone: 1800 645 349
email: support@macquarieonline.com.au



Australia's national dictionary

That's fine when we want all the content in all the referenced topics to be included in the output, but what if we don't want all of the content, just parts of it? That's where conditional processing comes in, the goal being to intelligently control which topics or parts thereof end up in the output. This control is achieved using metadata.

Metadata, often called "data about data", is a characteristic or trait that helps identify, clarify or classify an informational element. For example, an HTML paragraph tag might read:

```
<p class="dropcap"> ... </p>.
```

Here, the content of the `<p>` element is the data and the attribute, the `class="dropcap"` name-value pair, is the metadata. It classifies the type of paragraph (a CSS class in this case) so that it can be processed correctly. Or, in an XML document, a tag might read:

```
<cost currency="aud"> ... </cost>.
```

Again, the content of the `<cost>` element is the data, and the attribute, the `currency="aud"` name-value pair, is the metadata. It specifies that the cost element should be taken as Australian dollars. Metadata is often coded as attributes, as in these examples, but not always.

Metadata has various uses—such as workflow support, searching assistance and index preparation—but is really good at one thing in particular: conditional processing. The primary function of conditional processing is omitting undesired content, or *filtering*. DITA provides four standard attributes to control filtering:

- audience
- product
- platform
- rev.

It also provides a fifth attribute you can use to specify other properties, reasonably (albeit uncreatively) called *otherprops*. Using these attributes, you can classify everything from individual elements to entire topic groups, applying appropriate metadata to the objects to drive the filtering process.

The big benefit in terms of editing and maintenance is that mutually exclusive content elements don't have to be stored separately. You can put them all together in a single topic or map and leave out the pieces you don't need at build time. This technique prepares the content so that it can be conditionally processed, while simplifying maintenance by keeping logically related items physically together in a single source location. It's a great way to cram a lot of stuff into a small space—sort of like the Kardashian sisters.

Put 'er there

There are three standard places where you can put metadata: on individual elements, on topics and on map references.

Element metadata is used at the tag level to apply properties by which the elements can be identified and filtered during the build. Let's say we want to customise the first step in a task by user experience level. We could use the *audience* attribute to attach the appropriate metadata to three versions of the step, like this:

```
<step audience="novice"><cmd>Plug in
your PC.</cmd></step>
<step audience="intermediate"><cmd>Turn
on your PC.</cmd></step>
<step audience="advanced"><cmd>Boot up
your PC.</cmd></step>
```

Using this markup, we can easily produce a task topic with steps tailored to the specific audience we're trying to reach, regardless of PC expertise.

Topic metadata is used at the topic level to specify characteristics with which the topic can be filtered. If we wanted to produce a review document containing all topics written by a given content provider, we could use the *otherprops* attribute to identify each topic's author like this:

```
<task id="remove"
otherprops="AnnaGraham">
<title>Removing WhizBang</title>
...
</task>
<task id="repair"
otherprops="OttoPalindrome">
<title>Repairing WhizBang</title>
...
</task>
```

While the use of *otherprops* to indicate author name is entirely arbitrary, it demonstrates the power and flexibility of having a generic, user-definable attribute. The topics can now be identified by author and filtered appropriately during the build.

Map metadata is used at the top of the metadata food chain to apply filtering characteristics to whole topics or topic groups within maps. We could, for example, construct a single map that allows us to produce a user guide for any of several product releases by adding *rev* metadata attributes to the topic references, like this:

```
<map title="User Guide" id="userguide">
<topicref href="inst-demo.dita"
rev="demo"/>
<topicref href="inst-std.dita"
rev="1.x"/>
<topicref href="inst-upd.dita"
rev="2.x"/>
...
</map>
```

We're now able to select the correct installation topic (or a set of correct topics, regardless of number or hierarchical placement) for any current product release, from the demo version to 1.x to 2.x, without creating—and maintaining—separate map files. Also, recall that in a map, child topics (topicrefs inside topicrefs) inherit their parents' attributes, so conditional processing metadata attributes cascade or flow down just like other attributes. This allows you to affect whole groups of topics by placing just one filtering attribute on the parent.

How do you know on which layers to put your metadata? Well, it depends on several factors: content complexity, number of authors, the variety of attributes you use, and so on. In general, assign metadata to the highest level of specificity that makes sense. For example, if you need to easily swap out entire blocks of content, use map metadata to control topics by groups. If you have topics that are similarly structured but different in content, use topic metadata to differentiate them. If you have broad, generic content with many small, specific differences, use element metadata to keep the content together but allow it to be easily filtered.

Testing, 1 2 3...

Here's a great joke: "What do you call a musician with no girlfriend?". Wait, that's not funny, you say, and you're right. But why is it not funny? Because it's just a setup with no punchline. In comedy, technical communications, and most other worthwhile human endeavours, preparation is useless unless you deliver the kicker—and that's the problem with our examples so far.

Identifying unique elements, topics and maps and applying metadata to differentiate them is only half the job. Metadata itself doesn't do anything; it just sits there patiently waiting until it's needed. To make it useful, we have to tell the build process what to do with it; that is, we have to define the filtering conditions for the build.

The *ditaval* file is the mechanism we use for that purpose. Like the map file and the XSL transforms, the ditaval file is read by the build and used to drive the filtering process as the output stream is created. The ditaval file essentially contains two things: conditions to be matched and actions to be taken when they're found.

Ditaval conditions are defined with the `<prop>` (or *property*) element, which has three attributes:

- *att*, the metadata attribute to search for
- *val*, the metadata attribute value to match
- *action*, the action to be taken when the metadata attribute value is matched.

Think of it rather like a CSS rule: look for elements that contain the metadata attribute *att*; if you find

one, see if its value is equal to *val*; if so, perform the specified action.

You can include as many `<prop>` elements as you like, in any order. Much like CSS and XSLT, it's a wonderful demonstration of declarative processing at work. Let's look at some examples.

Earlier, we added the audience attribute as element metadata to some task steps (and presumably to other elements, topics and topic references in our content repository). Now, if we want to produce a user guide for novices, we might code conditions in the ditaval file like this:

```
<val>
  <prop att="audience"
val="intermediate" action="exclude" />
  <prop att="audience" val="advanced"
action="exclude" />
  ...
</val>
```

These conditions allow the *novice* audience elements through while filtering out the *intermediate* and *advanced* audience elements.

Next, we added the *otherprops* attribute as topic metadata to some topics, naming two contributing authors. If we want to produce a review document containing only those topics written by a single author, we can do it by excluding the other with a ditaval condition, like this:

```
<prop att="otherprops" val="AnnaGraham"
action="exclude" />
```

This will filter out Anna's topics and leave us with only topics written by her colleague Otto.

Finally, we added the *rev* attribute to some topic references in a ditamap, identifying installation topics for demo, 1.x and 2.x software versions. When we're ready to produce an installation guide for the 2.x version, we can code ditaval conditions to exclude the others like this:

```
<prop att="rev" val="demo"
action="exclude" />
<prop att="rev" val="1.x"
action="exclude" />
```

The result will be our desired document, an installation guide for the 2.x product only, with the demo and 1.x topics filtered out. Thus, the ditaval file's `<prop>` element becomes the killer punchline for the clever metadata setup.

Oh yeah, speaking of punchlines: "Homeless".

A hippo in the ointment

Now if you're ahead of me on this, and you probably are, you'll note that these examples seem to approach the document assembly process somewhat, well, backwards. We don't include the elements we want.

Instead, we exclude the ones we don't want. Odd as it seems, that's exactly how the *exclude* action works. Gosh, wouldn't it be nice if there were also an *include* action? Well, there is ... sort of.

The original ditaval scheme offered only the *exclude* action (and *flag*, which is beyond the scope of this article). But when ditaval officially became part of the DITA standard, the *include* option was added. It sounds promising, but—to use a phrase with which I'm painfully familiar—"it isn't what it looks like!". For example, you'd think that the single `<prop>` tag below is equivalent to the two `<prop>` tags above, including just 2.x content and excluding demo and 1.x content:

```
<prop att="rev" val="2.x"
action="include" />
```

But you'd be wrong. Yes, given that tag, the 2.x topics will be included, but so will the demo and 1.x topics. That's because the default action for all elements, marked or unmarked, is *include*. Let's say that again, because it's hugely important: the default action for all elements is always *include*. Since that's the case, you might be wondering if you could at least add that third `<prop>` tag to the first two, just to make your intentions clear. The answer is yes, but it's just like calling in your vote for Australian Idol: you can do it, but it won't make any difference.

The reason *include* doesn't work quite as intuitively as we'd like is because its primary use is for elements with multiple metadata values in the same attribute. The filtering logic for multiple values can get sticky pretty fast, so let's leave that for another article. Bottom line: *include* doesn't really do us any good in ordinary, everyday filtering, but that's really not a bad thing. Read on.

For now, we can safely say there is just one absolute, immutable rule for ditaval conditions. This rule is true regardless of your DITA Open Toolkit version, authoring tool or processing environment. It's true for all maps, topic references, full topics and individual elements, whether marked with metadata or not. It's true all the time, for all builds, in all cases, period, full stop, end of story. The rule is this:

Everything not explicitly excluded is included.

At first blush this rule seems restrictive, but in practical terms it greatly simplifies the process of marking up content for conditional processing. We can now approach our content with a simple plan: add metadata to anything we might want to exclude later and leave everything else alone! Because most content in a documentation set is included in most output formats (if not, you're doing it wrong), it's obviously easier to mark up some content you want to *exclude* under certain circumstances than to mark up all the content you want to *include* under most circumstances. Sweet.

Loose ends

But as you might guess, that's not quite everything. You can almost hear that fellow with the glass eye, cigar butt and rumpled trenchcoat say, "There's just one more thing".

We know that we add metadata to DITA elements and that we add `<prop>` conditions to a ditaval file so the build can properly filter the elements. But there's our missing connection: how does the build process know where our ditaval conditions are? The answer is simple, if inelegant. We tell it where to look.

A build file contains a number of `<property>` tags (not to be confused with `<prop>` tags in the ditaval file) that provide the build process with the required information, such as the input file location, the output file location, the desired output type and so on. To specify the location of the ditaval file containing the filtering conditions, we just add one more `<property>` tag to the build file, like this:

```
<property name="dita.input.valfile"
    value="${basedir}$
    {fileseparator}myprojects
    {fileseparator}UserGuide
    {fileseparator}userguide.ditaval"
/>
```

This tag tells the build that the ditaval conditions file *dita.input.valfile* is named *userguide.ditaval* and will be found in the *myprojects\UserGuide* folder under the DITA base directory, *C:\DITAOT* for example. The build can now load the filtering conditions from the ditaval file and apply them to the metadata attached to the various project elements.

Finally, although this article includes actual code snippets, I realise that hand-coding is so five minutes ago. Most good authoring tools now include user-friendly interfaces to the nuts and bolts of metadata, build conditions and file locations, so that setting up and implementing conditional processing is relatively easy. But I figure when you're admiring the dashboard, it's still good to know what's under the bonnet.

Summary

DITA is a brilliant implementation of structured authoring, incorporating single-sourcing, content sharing and reuse, and conditional processing as core technological elements.

Conditional processing is at the heart of content specificity, and metadata is its control mechanism. Grasping the relationship between metadata and filtering is one of the *aha!* experiences we have along the road from linear narrative to structured authoring, a little epiphany that suddenly propels us forwards in our efforts to get the most benefit from technology and makes our jobs a little bit easier, a lot more productive, and yes, sometimes even fun.

Dave Gash

Dave Gash owns HyperTrain (www.hypertrain.com), a California firm specialising in training and consulting for hypertext developers. A veteran software professional with over thirty years of programming, documentation and training experience, Dave holds degrees in Business and Computer Science, and is well known in the technical publications community as an engaging and animated technical instructor. Dave is a frequent speaker at user assistance seminars and conferences in the US and around the world. He will be speaking at the AODC 2010 conference in Darwin (May 12–14).

HyperTrain dot Com

HyperTrain dot Com specialises in:

- Online Help
- HTML
- JavaScript
- CSS
- XML
- XSLT and
- DITA.

We also offer a full range of tech pubs services including group or individual training, project jumpstart, technical writing, and end-to-end project development.

Toll free (US): 888-722-0700
International: +1-760-214-0698
Web: www.hypertrain.com
Email: dgash@hypertrain.com

Language: the core skill in technical writing

Geoffrey Marnell¹

Like relativity in physics, usability in documentation is a concept that simply can't be ignored. It colours—or should colour—every decision we make in designing and writing documentation. But like relativity, pinning down a useful definition of usability is no easy matter.

The International Standards Organization describes usability as “the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction”.² This gives some conceptual traction, but it lacks the necessary concreteness to make its application facile.

A more concrete definition, and one more widely discussed, is based on the work of Gretchen Hargis and her colleagues in defining *quality* documentation.³ This view of usability (and quality documentation) has it that the information in documentation must be:

- easy to find
- easy to understand and
- easy to apply.

The linking concept here is provided by the common definition of quality as “fitness for use”. Obviously, if a document is not fit for use, it lacks usability; and if it is fit for use, it has usability—at least to some degree. And it is plausible to judge that degree on how easy the information in it is to find, understand and apply.

For information to be easy to find, there must be sufficient signposts in places where readers are likely to look. The two most likely places are an index and a contents list, with an index arguably the more important of the two in a document of more than a score or so of pages. A contents list, though useful to the occasional browser, simply hasn't the degree of granularity needed to help the typical reader: the time-poor, deadline-harassed person needing to know in a hurry how to do the particular task at hand. For such a reader, a lengthy document without an index would be seriously deficient in usability.

Other aides in helping users find information easily include running headers and footers, cross-

references and hyperlinks, lists of related tasks, breadcrumbs, and a full-text search facility (especially one that enables wildcard searching and Boolean filtering). All these features, to varying degrees, help readers find the information they are after, and thus contribute to the overall usability of documentation.

Once a user has found the information they are after, they need, of course, to be able to understand it. This is where language and usability intersect, and the influence of the former on the latter is the main topic of this paper. I will come back to it shortly.

The other pillar of usability is that the information, once found and understood, must be easy to apply. To that end, it must deliver what it promises. A procedure promising to explain how to set a timer on a video recorder might be easily found and well-written, but if it doesn't fully explain how to set a timer, and under all likely conditions, then it is less than maximally usable. Moreover, it should not cause the reader to back out of the procedure by introducing prerequisites in the steps rather than in the preamble, nor cause the reader to consult other sections of the user guide in order to complete the procedure they are working their way through. Thus the information must be relevant, accurate, comprehensive and self-contained.

But let's return to the second pillar of usability: the information presented in documentation must be *easy to understand*. This is arguably the most important facet of usability in the documentation field. There may be a plethora of signposts directing a reader to the procedures they might need (and thus the information is easy to find) and each procedure may well cover all conditions and be self-contained (and thus score not too badly on the easy-to-apply scale); but if a reader has to struggle to understand the information presented to them, then the usability of the document is undeniably deficient.

But what is meant by *easy to understand*?

Understandability and readability

One often hears the KISS principle extolled in technical writing circles: Keep It Simple, Stupid. Alas, the KISS principle is hoist with its own petard. It is just too simple to be of any use. Still, much effort has gone into providing simple measures of understandability, measures that, unlike the KISS principle, have some *prima facie* claim to scientific rigor. These are the so-called text-based readability formulas, the most well-known of which is the Flesch reading-ease formula (the maths behind the readability scores generated by Microsoft Word).

1. A version of this article appeared in the June 2009 issue of *Intercom*, the monthly magazine of the [Society for Technical Communication](#).
2. International Standards Organization, *Human-centred design processes for interactive systems*, ISO 13407:1999. The definition is repeated in numerous ISO standards directed at technical communicators, such as ISO/IEC 18019:2007 and ISO/IEC 26514:2008.
3. Hargis G, Carey M, Fernandez AK, Hughes P, Longo D, Rouiller S & Wilde E, *Developing quality technical information: a handbook for writers and editors*, Prentice Hall, NJ, 2004.

For a start, readability and understandability are often used interchangeably:

*"Readability means understandability. The more readable a document is, the more easily it can be understood ..."*¹

Hence readability formulas such as the Flesch reading-ease formula can be considered contenders for determining the usability of documentation (or at least that component related to ease-of-understanding).

But the Flesch reading-ease formula errs on the side of KISS-like simplicity. It takes as its input just two features of text: average sentence length and average syllable count. Nothing about the reader is included, such as their familiarity with the concepts discussed. And many features of text that necessarily contribute to, or detract from, understandability are ignored: conventional grammar and punctuation, typographical cueing, contradiction, inconsistency, non sequiturs, ambiguity (especially that resulting from the use of transitional vocabulary), and many more. It is just far too easy to concoct a difficult, or even nonsensical, piece of text that scores well on the Flesch reading-ease formula. (*Type spray in the short thought* scores just as well as *The cat sat on the mat* on the Flesch formula.) Short sentences and monosyllabic words do not understanding make.

To those who accept these limitations but argue that the Flesch reading-ease formula is still the best proxy measure of readability we have², we can retort that *best* does not imply *good*. At one time, the *best* way we had of estimating the number of stars in the universe was to look at the night sky and count them. But that, obviously, was not a very *good* technique.



1. *Editing technical writing*, by Donald C. Samson Jr., Oxford University Press, New York, 1993, p. 58.
2. See, for instance, William H. DuBay, *Smart language: readers, readability, and the grading of text*, Impact Information, Costa Mesa, CA, 2007, p. 79.

Further, numerous studies have failed to reproduce the sort of validation correlation that excited Flesch—the correlation between Flesch scores and scores on independent comprehension tests—and any such correlation is necessarily inflated by ineradicable sampling bias.³

We should not be fooled, then, into thinking that its use in Microsoft Word gives the Flesch reading-ease formula the imprimatur of scientific rigor. The formula is overly simplistic and offers little guidance in determining whether a piece of text meets any likely usability criterion.

Understandability and communicative efficiency

We get closer to an understanding of *understanding* if we reflect on why we write, namely, *to communicate*. We communicate if we get our message across. But our success in getting our message across can be judged in degrees. We might achieve effortless communication: our readers get our message immediately, without any cognitive or emotional struggle. At the other end of the spectrum, we might fail completely: ambiguity, vagueness, conceptual denseness, and a host of other factors might block all attempts at deciphering our intended message. And in between are the readers who eventually work out what we mean, but only after some degree of struggle, or an encounter with more words than were necessary to get the message across.

Communicative efficiency captures the notion of ease-of-understanding far better than sentence length and syllable count. Efficiency entails effectiveness: obviously we need to get our message across if our communication is to be efficient. But it also entails that we get our message across *with the least effort on the part of our readers*. In other words, we should write with maximum economy, using language that is most familiar to our intended audience, and which has the least potential for *distraction* (which might arise, for example, if we engage the emotions of our readers with paternalistic or insensitive language, or if we use language inconsistently).

Usability, words and the flight from technical writing

Ease of understanding, and thus usability, depends, then, on our writing exhibiting clarity, economy, familiarity, neutrality and consistency. And thus it is impossible, in our field, to achieve maximum usability without a pre-eminent respect for language and for the words that are its building blocks. For we risk failing to get our message across if a careless

3. See Geoffrey Marnell, "Measuring readability. Part 2: Validation and its pitfalls", *Southern Communicator*, issue 15, October 2008, pp. 17–21.

choice of words leads to ambiguity, vagueness, bafflement, offence or cognitive overload.

Words, then, should be at the centre of our professional concerns. And yet words and language can often seem of marginal concern to technical communicators. The threads on discussion forums, the articles published in our journals and the marketing materials designed to attract students to our university courses, lean strongly toward tools, methodologies and practices. Issues of language are often missing or downplayed.

Our obsession with broadening our profession's profile—apparent in the number of times we have changed our name—may have contributed to the drift away from appreciating the importance of words. We were once technical *writers*, and when we were, the importance of writing—of words and of language—was explicit. It needed no explaining. But we did, have always done, more than writing, and thus we felt a need to be called something else: technical communicators, content providers, end-user assistance professionals, information designers and so on.

But other professions are not so touchy about their name. Teachers do more than teach. They also act as playground monitors, sports-day referees, mentors, excursion leaders, and curriculum designers. But they still call themselves *teachers*. We do more than write, but, unlike teachers—and many other professionals—we have sought to change our profession's name to make what we do explicit.

In the process, we have ended up achieving the opposite: concocting names of such bland generality as to encompass many clearly distinct professions. (A journalist, graphic designer and musician can all be seen as *content providers*; and a call-centre representative is also an *end-user assistance professional*.) We have failed to identify and differentiate ourselves by adopting names that

drown out our particular, unique contribution. And in doing so we may have lost sight of the fact that *writing* is what most of us do most of the time (just as teaching is what most teachers do most of the time). We may be especially fond of tools and methodologies—and there is no harm in that; indeed some degree of tools expertise is essential—but expertise in XSL transforms, DITA, persona mapping, VBA macros, Framescript, wiki design and the like is of no use if our writing—our particular, unique contribution—fails to achieve its primary purpose: effortless communication. It is writing before all else, and that is so even if some in our profession spend all their working time doing things other than writing.

To its credit, our profession has always prized usability. We may not have always agreed on what it means, nor given due respect to the need to clarify its definition. But a modicum of reflection on why we do what we do, on the ISO definition of usability, and on the work of Hargis and her colleagues, should bring home the fundamental importance of language to our profession. Words are what make or break us. Our technical skills are secondary, and have always been secondary. Their relevance changes from year to year, version to version—unlike that of language. So if we are to continue our commendable respect for usability, we must return language to the spotlight. We must develop a passion for language that matches that of lexicographers. We must put down our prescriptive grammars and become scientists of linguistic flux. We must accept that being a users' advocate—which most of us do—requires immersion in the users' language. For what good is an attractive, well-structured document—even a well-crafted sentence, written once and re-used often—if it fails to deliver its meaning to the audience for which it was intended.

Geoffrey Marnell

Don't miss this great professional development and networking opportunity!

**12th to 14th May 2010
Darwin, NT, Australia**



20 Sessions... 10 Expert Speakers... Pre-Conference Workshops... Social Events... Networking...

Knowledge Management, DITA, eLearning, Help, Structured Documents, Wikis, Case Studies, more...

Keep your skills up-to-date, and **learn new techniques, tips and technologies**. If you missed a previous AODC, don't make the same mistake in 2010!

The biggest annual event for technical communicators from Australia and New Zealand

Register now at www.aodc.com.au

Proudly presented by **HYPERWRITE**

Journals

Rhetoric, Professional Communication and Globalization

Barry Thatcher, Kirk St. Amant

This journal publishes articles on the theory, practice and teaching of technical and professional communication in critical global and intercultural contexts, such as business, manufacturing, environment, information technology and others. As a global initiative, the journal welcomes manuscripts with diverse approaches and contexts of research, but manuscripts are to be submitted in English and be grounded in relevant theory and appropriate research methods. The journal is peer reviewed with an editorial board consisting of experienced researchers and practitioners from over 20 countries. The journal's main objectives are to:

- Improve the research, teaching and practice of technical and professional communication across a range of global and intercultural contexts.
- Develop better theories and research methodologies for global and intercultural professional communication.
- Use professional communication research and practice to address issues of social justice in critical global contexts such as manufacturing, environment, law, health, immigration, energy, economic development and human rights.

- Address how new information and communication technologies influence the forces and processes of globalisation, especially in developing regions.
- Develop better curricula for teaching global professional communication, not only in the United States and Europe, but around the world. Special attention will be given to countries and regions in development, including parts of Latin America, Eastern Europe, Asia, Africa and India.
- Address the influence of English as the *de facto* language of global professional communication.

The journal will be free or *open access*, using PKP open-source software and housed at East Carolina University. The first edition is planned for September 2010, and it will be published quarterly thereafter. For more information visit www.rpcg.org or contact Barry Thatcher (bathatch@nmsu.edu) or Kirk St. Amant (kirk.stamant@gmail.com).

Barry Thatcher

Founder/Editor-in-Chief, New Mexico State University

Kirk St. Amant

Assistant Editor, East Carolina University

Journal of Technical Writing and Communication


Contents of the current issue

Click the heading below to read the editorial, abstracts and reviews, or to purchase an article or subscribe.

Volume 40, Issue 2, 2010

- From the Editor's Desk, Charles H. Sides, Executive Editor
- Relationship between Innovation and Professional Communication in the "Creative" Economy, David Hailey, Matthew Cox, and Emily Loader
- Coverage of Team Science by Public Information Officers: Content Analysis of Press Releases about the National Science Foundation Science and Technology Centers, Marita Graube, Fiona Clark, and Deborah L. Illman
- Introducing China's First Comprehensive Technical Writing Book: *On Technological Subjects* by Song Yingxing, Daniel D. Ding
- Cross-Cultural Blunders in Professional Communication from a Semantic Perspective, Pinfan Zhu
- Writing Degree Binary: An Argument for Interscription, Christopher Berg
- A Dozen Years After Open Source's 1998 Birth, It's Time for *OpenTechComm*, Brian Still

award-winning, authoritative international voice, publishing the latest research by recognized scholars from around the globe



Every article ever published in the JTWC is now available in clear, concise, comprehensive PDF format.

New Online-Only Delivery Options

- Enhanced subscription (all articles from volume 1 through current volume)
- Current volume subscription
- Individual articles on pay-per-view basis

Click here for all the details and ordering information.

- Read all article abstracts free
- Download a complimentary sample issue

BAYWOOD PUBLISHING COMPANY, INC.
<http://baywood.com>

The kitchen sink checklist: Book review

Peter Martin

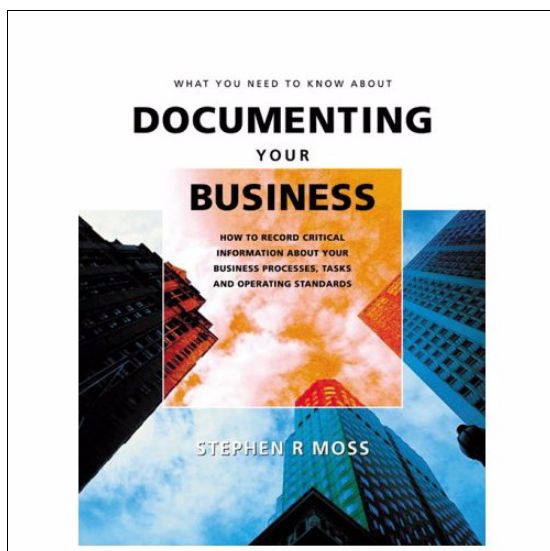
What you need to know about documenting your business: how to record critical information about your business processes, tasks and operating standards by Stephen R Moss (Pearson Education, 2001, 354 pages)

It was an early acquaintance with the joys of QA, and my first documentation job on a piece of hardware. The task: write the installation guide for a newer, smaller version of a remote terminal server, the latest child of four product generations.

In theory, it was mainly a case of getting some good vector graphic drawings together, and then updating the text for a few new features. In practice, it was the most painful experience I can recall in a couple of decades of technical writing.

The thing that made the difference was the so-called *formal review* process. That involved three days with three subject-matter experts (SMEs) and two technical writers in a room for up to one hour or more, each taking turns to read the manual word-for-word, pausing only to note significant errors and issues. At the end of it, I was psychologically demolished, and furious with myself. There were no fewer than 120 basic errors of fact uncovered in about 80 pages of text.

When the embarrassment and self-fury subsided, and corrections had been made, I took the time to find out what had gone wrong. The revelation came with a review of earlier generations of the documentation. Almost 95% of the errors were boilerplate errors: many of them had survived, in error, through all four generations of documentation. I had assumed the previous material was of biblical status, and it wasn't: no-one had bothered to check it thoroughly before.



The formal review can be a painful technique in so many ways, but it is probably far too rare in modern business environments. It should probably be made compulsory in some systems, especially those where life and limb can depend on the accuracy of documentation.

I mention the formal review here because it appears to be about the only process I can think of which Stephen Moss does not specifically deal with in detail in *What You Need to Know About Documenting Your Business*. Despite that omission, this is a book I intend to keep on the shelf above the computer, there to consult for every technical writing job I do from now on. It's a checklist covering just about everything but the proverbial sink. It may well be eminently put-downable, as it is no Peter Temple thriller, but it's something to have handy, just to remind yourself of what needs to be done to cover all bases.

I have a few minor qualifications to raise about this book, but firstly, the basics:

- It's oldish (copyrighted 2001) and in a few areas, shows its age. More on that later.
- It has an almost classical technical writing design, with significant and suitable touches of Information Mapping (but without a slavish demand for purity in that discipline). It's easy to follow its structure.
- It sits flat fairly comfortably on the desk, something I find useful in a workbook.
- In all, there are some 350 pages in letter size (American quarto), broken up into three main parts:
 - ◆ a management overview of business documentation
 - ◆ key areas of document development
 - ◆ detailed methods of documenting processes, tasks and operating standards.

There is a very handy bibliography to guide the reader to other works of value, and relevant notes in the text pointing out where these other sources may be most valuable. There are guides to publishing tools, document-viewing tools and even presentation and binding formats. And there is a comprehensive index—for which, much thanks.

As a side note, I found myself both puzzled and somewhat outraged by a publisher's note on the copyright page. This note, somewhat uselessly, tries to disclaim responsibility for the book's content, claiming that "[it has] not undergone the rigorous editorial and development processes normally afforded" to their titles. I only hope Stephen Moss

was given an increased share of gross receipts in exchange for that egregious and rather useless cop-out from a company only too willing to reserve all rights—but not all responsibility.

On the subject of the age of the work, there were occasional surprises for me as I saw how much things have changed while we've all been having such a good time. For example, in a reference to screen viewing there's mention of a *premium* monitor size of 21 inches. At a time when my neighbours' TV monitor is now about the same width as the car in their garage, this tended to jump out at me a little. How things change, as our horizons and eye sockets expand.

I would also take issue (and have, frequently) with the quick and common assumption that sans serif fonts are better in online documents. (But that was in another country, and besides the wench is dead.)

Perhaps more importantly, I'd argue that a shortcoming of the book is that some more recent changes in business and development methods are not covered. I'd suggest, for example, that *sprints* and *scrums* are changing the way work-flows are organised, at least in IT development. In turn, this is changing the way documentation tasks have to be organised.

But Stephen Moss can hardly be expected to have caught up with all of this much before 2001. But a related subject which I might have expected to see covered is the difference between documentation tasks in small companies compared with those in larger companies. This book is comprehensive in the sense that it covers what you ideally should do in a large company. But not all companies control that ideal world: and many would drown if they bothered to try.

Smaller companies have often found that they simply cannot afford the sheer weight of paperwork involved in, say, full QA in a waterfall development environment. Far too often they find themselves writing specification documents a week or two after product delivery, and are left with top-level design documents that no-one managed to keep up to date after the first draft. We could have done with some

The Words team

- Artwork: Christine Weaver
- Copy-editor: Claire Mahoney
- Adviser: Mark Ward
- Cartoons: Aaron Bacall and Fran
- Editor: Geoffrey Marnell
- Contact: words@abelard.com.au

© Individual contributors or Abelard Consulting Pty Ltd, 2009
[unless otherwise noted]



commercial
translation
centre

For Effective
Translation/Localisation
Of Technical Documentation
In 80 Languages

Australia Wide Service Since 1988
Call 1800 655 224 (Australia Toll Free)
Email: mail@ctc.com.au
Web: www.ctc4.com

commercial translation centre

notes on how to cope when we're not in the best of all possible worlds, but in a world of continuing compromise.

Yet this may, paradoxically, be one reason why this book is a good one to have in your library. Even if you work in a small company, the book can serve as a reminder of the kinds of things you didn't quite get around to doing in full but which might still be useful somewhere as a short segment in one of the few documents you've actually had time to write.

I think there's still a need for technical writers on small projects to consider the implications of new workflows in the profession (for example, those arising from agile development and its so-called sprints). The differences may not seem all that great, but there are subtle differences that call for a re-examination of things like larger-scale project plans and detailed specifications. The world is not full of documentation phases that just flow neatly into maintenance phases: there are greys out there.

And there's also a need to keep up to date with the forms of final documentation presentation, and the newer production methods and tools now being used. In 2001, there was an understandable uncertainty about whether XML was going to affect HTML. Heading into the second decade of the century, we know now that XML is actually the established thing (if only in the form of XHTML). XML does seem to be the basis for continuing development of web markup language. And we know there are a range of new tools and reworked old tools that have been developed to deal with these changes.

On the point of documentation tool choices, I might also question a suggestion or two made about factors that might be given weight in such decisions. Stephen Moss rightly points out that some tools, such as Adobe FrameMaker, carry with them the problem of what to do about document maintenance, as fewer people have FrameMaker skills than Microsoft Word skills. The suggestion given is very much a big-company one: what if you have 80 different sections, each requiring its own document? I don't see that the answer given is really satisfactory: that you use Word because that's the base tool that all sections know about and use. At that stage of the game, it's clear you need a big documentation section with experts in technical writing and sophisticated tools.

Frankly I don't care what reviewers and SMEs are used to. The further they are kept away from any source files the better I like it. Let them cut text. Simple text—or, at its most complex, let them have a simple HTML tool or even a Wiki. But leave the formatting to those who know what they are doing, and keep it all in one place. If taps and pipes are a problem, you get in a plumber. If source maintenance is an issue, get someone trained in maintenance work (and they don't come as expensive as plumbers).

One final note: on the writing styles in this book. I found more than one style early that annoyed me a great deal, but was relieved to find a second style popping up at different points. A great deal of this book is written in the prescriptive style approaching

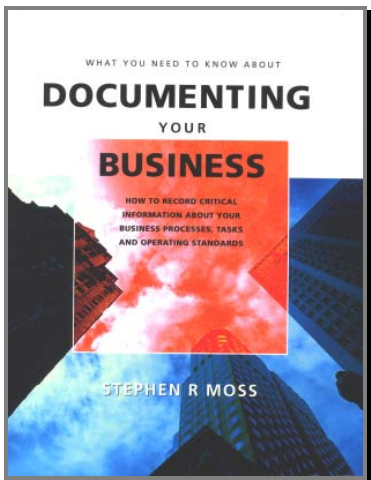
that which is far too common in, for example, ISO standards documents. By *prescriptive* I mean simply the use of phrases and clauses like “this should be done”. This comes up early in the book when almost every conceivable role in the documentation process is being outlined. It's not that Stephen Moss is as bad as those ISO document authors who vainly think (in vain) that they can re-define English usage of the word *shall* by means of a single footnote. But there are moments early in the piece when I prayed for a block of text which used the second person, while shuddering at the gradual accumulation of passive tenses. Fortunately, my prayers were answered, and there are passages of the second kind which are a lot more readable than any ISO standard I've stumbled over: like a cool drink of water after a long run.

In summary: give or take a few minor problems that in no way justify the publisher's disclaimer, this book comes up well. It's worth it for lots of good reasons, none of which, apparently, have anything much to do with the publisher. (Well, that's what it says!). It's just a bit of a pity that something that looks like an internal managerial dispute between arms of a publishing empire should pop up to mar the image. Someone needs to tell a publisher or two that authors are clients too.

Peter Martin

Peter Martin has been contracting in technical writing for 21 years, after a career as a current affairs journalist, a ministerial adviser, a press secretary and a commissioner with the Australian Film Commission.

This is a book I intend to keep on the shelf above the computer, there to consult for every technical writing job I do from now on. It's a checklist covering just about everything but the proverbial sink.



knowledgeDOC
exceeding expectations

To purchase a paperback copy of
Documenting Your Business
by Stephen R Moss

Special for WORDS readers!

Only AUD \$30 + postage and handling (\$9.90 within Australia)

Email [Knowledgedoc](mailto:info@knowledgedoc.com.au) (info at knowledgedoc dot com dot au). Include:

- your name
- full postal address
- number of copies required.

We will send you a PayPal online payment request.

Poetry

In the last issue of *Words*, we noted that back in the 1980s and early 1990s, a number of academic papers were published linking poetry and technical writing. There were claims that both disciplines used similar language structures and communicative techniques, and that academic studies in poetry provided a good basis for teaching technical writing. This led the *Words* team to call, not for papers linking poetry with technical writing (a link somewhat fanciful to most practising technical writers), but a call to poets and would-be poets to submit a poem related, in some way or other, to technical writing. Thankfully, the call did not fall on deaf ears. Our special thanks to the contributors.

qwertea

A compact explanation should fit in the palm of your hand.
Tail curled around your wrist, eyes at rest.
Silent, warm and flawless.
But some days it fidgets like unfinished business.
Clawing and contracting; the right verb guarded in its cheek.
All teeth and appetite.
Proving it's sometimes wiser to dip your qwerty in a hot cup of tea;
and feed your procedure a stale biscuit.

Claire Mahoney

Everyday madness at work

My pencil, a lancet of lead
That skewers the spider-word specimens
As they burst from their mountain cave home
To lie there unmoving, unchanging
On the page in their moment of death

My keyboard, a maze of buttons
A dance-floor for blindfolded fingers
They drunkenly careen and stagger
To the orchestral tune tac-tacking
What must that band-master be thinking?

My desktop, a platform tectonic
I build mountain range river and plain
Rickety cities and ramshackle shanties
Up-ended, tsunamis of flotsam
As I quest for the missing leaf

My chair, a marvel of modern day madness
Levers and spindles, adjustors and tilters
Uppers and downers, sideways and backers
Smooth-bearing swivelers spectacular
Facilitate ice-skater twirls
Technology tow'ring
O'er something that's merely
For sitting

Dan Smith

Aubade for the tech writer

(With apologies to Philip Larkin)

I write all day and check in my files at night
Leaving at five for my other life, I think
In time the doco structure will grow tight
Till then I see what's really always there
The editor's massacre, a whole day nearer now
Making all thought impossible but how
And where and when my grammar will falter
Perfect pagination, yet the dread
Of editing and being edited
Flashes afresh to hold and horrify.
The cursor blinks in the air.
Not in remorse—the style not followed,
The change bars not turned on, the spellchecker left unused
Nor wretchedly because
An edited file can take so long to climb
Clear of its strange idioms, and may never;
But at the total emptiness for ever,
The sure extinction that user guides travel to
And shall be lost in always.
Not to be read, not to be followed,
And soon, nothing more ignored, nothing less known.

Meanwhile, programmers crouch, getting ready to code
In beige cubicles, and all the unedited
Bloated software begins to grow.
The screen is as blue as death, with no Windows.
Restarts have to be done.
Viruses, like rats, go from desk to desk.

Suchitra Govindarajan

"There are two ways of disliking poetry: one way is to dislike it, the other is to read Pope."

Oscar Wilde

"Poetry is an echo, asking a shadow to dance."

Carl Sandburg

Youse

[Dashed-off doggerel for John Wilson]

How odd that such a simple sound
Can rouse in some
Revulsion almost visceral—
A shuddering of the soul.

Its parts are nothing new but
Well established in the alphabet
Of sounds—*juz*—a soft concatenation
Yet together and from certain lips
In certain ears revolt.

“Illiterate”, the complaint is heard
Although the utterer of the word
May never have been schooled
In all the fineries of palatial speech.
But whose fault is that?

Do we scorn those poor sods
Whose numerical dexterity
Leaves a lot to be desired
Whose sums and long division
Could so easily be decried?

No. And nor do we descry those
For whom the quark is a concept
Weird and dark, as is ergs, as is bosons
Gravitons and photons.

It is just a matter of schooling, surely?
And ignorance is no laughing matter.

So why is it different with language,
Why does ignorance of words
But not of atoms and of surds
Cause so many of us to choke
In ridicule and belittlement?

Much of what we accept today
Came from the common folk
So should we rewind all of English
And strip out all its vulgar links?
Or is there something else at play
In your choice of what is good
And what stinks?

Perhaps in language we see a way
To brand and mark, a simple way—
No fee required—to place ourselves
Above another, the them
That is not us. A snobbery so few
Admit, but so common as to
Make one think it springs
From sources somewhat deep, perhaps
A canny twist of DNA
That gives the utterer of palatial speech
An advantage I regret not having—
Considerably more sex?

But here’s another twist:
This urchin word does not
Dilute nor does it blur
Our mighty language
Unlike so much in
The swamp of daily verbiage.
Indeed, it adds.
It fills a gap that’s
Yawned for many centuries:
No second-person plural!
How could English have become so powerful
When it lacks what lesser tongues
Have had since birth?
And there lies the irony, that so
Useful and mellifluous a word
Could be so thunderously rejected.

It’s not its freshness that offends:
New words are cuddled every day.
Who sniggers at blog?
Who frowns at Blu-Ray?
But these drop from the tongues
Of today’s High Priests
The technocrats and their caucus
Drop one instead from the hoi polloi
And the pedants’ shrieks are raucous.

Stop this bigotry at once!
It stinks so much of class.
Exclusion based on origin
Is nothing short of farce.

Do youse understand?

Geoffrey Marnell

Miscellany

Word of the Year: *shovel-ready*

The Macquarie Dictionary Word of the Year Committee has announced the adjective *shovel-ready* as the 2009 Word of the Year. Its definition: *of a building or infrastructure project, capable of being initiated immediately as soon as funding is assured*. The committee comprises Dr Michael Spence (Vice-Chancellor of the University of Sydney), Professor Stephen Garton (Dean of the Faculty of Arts at the University of Sydney), Les Murray (renowned Australian poet), Peter Fray (editor of *The Sydney Morning Herald*) and Susan Butler (publisher of the Macquarie Dictionary). The People's Choice Award was *tweet* (in both its verb and noun forms).

Other additions to the lexicon given an honourable mention by the committee include *roar factor* (the influence that a home crowd has on a referee or umpire in making adjudications), *head-nodder* (supporter of a politician or other media figure who stands beside them in the frame of a television shot and nods his or her head in agreement with what the speaker is saying) and *heritage media* (media, as print newspapers, television, etc., which, although strong and influential in the past, are thought to be losing viability in the face of changing methods of communication).

See www.macquariedictionary.com.au for more new words that have made it into the dictionary.

Percent or per cent?

Many people—even folk who make their living from writing—are under the impression that *per cent* is a shortened form of *percentage* and, on that ground, justify spelling it as one word: *percent*. However, *per cent* is a shortened form of *per centum* (a Latin expression meaning “for or in every hundred”). And just as we write *per annum* and not *perannum*, and *per hour* and not *perhour*, logic would dictate that we write *per cent* and not *percent*.

But logic, of course, is a poor guide to language. The Americans have been writing *percent* for quite some time, and so used are some Australians to reading American books and magazines that, even when admitting to reading Australian newspapers—where *per cent* is ubiquitous—still contend that the two-word spelling is rare and odd.

More emetic writing

What does it take to win a Pulitzer Prize for writing? The sort of talent, it seems, for producing writing of the following quality:

“America’s scientific eminence is one of our greatest sources of strength ... [so] let’s cosset the scientists of today and let’s home-grow the dreamers of tomorrow, the next generation of scientists. For by fostering a more science-friendly atmosphere,

surely we would encourage more young people to pursue science careers, and keep us in fighting trim against the ambitious and far more populous upstarts India and China.”¹

One has to wonder whether Richard Dawkins read the book before commenting that “every sentence sparkles with wit and charm”, an endorsement the publishers splash across the front cover. To call someone an upstart is hardly a witty or charming comment. (*Upstart*: a parvenu; someone who has risen above their class or to a position above their qualifications. Well put, Macquarie.)

It is also a sad reflection on contemporary science if its principal purpose is not to advance the well-being of living beings and their habitats (as was once thought) but to advance the interests of some particular country or other (a view Angier appears to be promoting). Perhaps she’s not alone. Try getting a grant to undertake *pure* research these days.

Regional variations in Australian English

By and large, Australian English is homogenous, but there are some regional differences. For example, in most states of Australia, the pole that carries electricity and telephone cables is called a *power pole*. But in South Australia it is more commonly known as a *stobie pole* (after the engineer who designed them). If you were writing a user guide for, say, the line-persons who work for South Australia Electricity, *stobie pole* would be the term you would have to use.

Another regional variation is the name for German sausage which, depending on where you live, might be called *fritz*, *devon* or *strasbourg*. We also have *flake* in Victoria, but *shark* in West Australia, and *potato scallop* in New South Wales but *potato cake* in Victoria.

Further, in Victoria, a corner store or corner shop is called a *milk bar* (an expression unheard of by many residents of New South Wales).

South Australia has a few more oddities up its sleeve. A *pint* of beer in South Australia does not have the same capacity as a pint of beer elsewhere in Australia. And one expression that seems to be exclusively South Australian is *double-cut*. Order, say, a salad roll in Adelaide and you may be asked if you want it double cut, that is, cut horizontally twice, giving room for two layers of filling. A bit like a club sandwich, but for a roll.

And a Brisbane taxi driver reminded me recently that I had two *ports* in the boot. Now there’s a old synonym for *suitcase* you don’t hear very often, if at all, in our more southerly states.

1. Natalie Angier, *The canon: the beautiful basics of science*, Scribe, Carlton North (Australia), 2008, pp. 9–10.

Mindstretchers

Geoffrey Marnell

Self-centred sentences

In what climatic conditions are each of the following sentences, taken in turn, true: When it is raining? When it is fine? When it is either wet or dry? Or under no circumstances at all?

1. It is raining and this sentence is false.
2. Either it is raining or this sentence is false.
3. If it is raining, then this sentence is false.
4. It is raining and at least two of these four sentences are true.

Solutions will appear in the next issue of *Words*.

Solutions to the last puzzles

Puzzle 1

The first part of puzzle one asked:

What number comes next in the following sequence?

1 4 9 16 ?

The expected approach to solving puzzles of this type is to find some mathematical rule that generates the numbers that are given and use that rule to generate the next number in the sequence. Such puzzles appear frequently on tests of numeracy and of IQ.

Many readers would have noticed that the difference between consecutive numbers in the given sequence forms the set {3, 5, 7 ...} and would have assumed that the next number in the set must be 9, thus making the answer to the puzzle 25. But this is only one of an *infinite* number of possible answers.

Such puzzles are fundamentally flawed. Any finite sequence of numbers can be continued *in any way at all* and the sequence, as continued, will be the product of some mathematical rule (a *polynomial*, to be precise).¹ For example, suppose I continued the given sequence with 13 instead of 25:

1 4 9 16 13

On the face of it, this looks rather bizarre, and yet it is generated by plugging, in turn, the values {0, 1, 2, 3, 4 ...} into the following equation:

$$y = -\frac{1}{2} \cdot x^4 + 3x^3 - \frac{9}{2} \cdot x^2 + 5x + 1$$

A little playing around with polynomials in general will quickly prove that any four-numeral sequence can be continued in any way at all and still be the product of some fourth-order polynomial.

1. Contact the editor (geoffrey@abelard.com.au) if you would like to receive a proof of this claim.

Consider the second sequence from the last issue of *Mindstretchers*:

3 -112 69 1003 -47 ?

This looks impossibly unruly, and yet it too is the product of a polynomial:

$$y = -\frac{1597}{12} \cdot x^4 + \frac{2624}{3} \cdot x^3 - \frac{18533}{12} \cdot x^2 + \frac{4127}{6} \cdot x + 3$$

Plugging 5 into this equation gives -9012 as a *logical* continuation of the sequence: but just one of many.

The only strictly correct answer to sequence puzzles of this type is “any number”. Alas, if you gave that answer you would almost certainly be marked wrong—and possibly miss out on that scholarship to someone with less developed mathematical skills. (To retort that these puzzles are looking for the *simplest* answer fails to acknowledge (a) that such a constraint is never explicitly made and (b) how impossibly difficult *simple* is to define.)

Puzzle 2

A first suspicion is that the woman was born on 29 February 1896. Otherwise she would have had 24 birthdays by 1 April 19 20, not 5. And yet, with a birthday every four years—given that leap years occur every four years—she should have had 6 birthdays by 1 April 1920.

The error, here, is an error of fact: leap years do not occur every four years. By the Gregorian calendar, which has been adopted by most of the Western world, century-years (e.g. 1700, 1800, 1900, 2000 and so on) are leap years only if they are exactly divisible by 400. Since 1900 is not exactly divisible by 400, it was not a leap year and so did not contain a 29 February. Thus the woman in question did not have a birthday in 1900. Her first birthday, then, would have been in 1904, after which she had a birthday every four years, giving her just 5 birthdays by 1 April 1920. Incidentally, the old Julian calendar (which the Gregorian calendar replaced) did make every fourth year a leap year.

AUSTRALIAN STYLE

A NATIONAL BULLETIN ON ISSUES IN AUSTRALIAN STYLE AND ENGLISH IN AUSTRALIA

Announcing the online rebirth of the biannual newsletter published at Macquarie University, featuring:

- o articles on language usage and research
- o the Feedback questionnaire
- o the exclusive crossword Rubicon by David Astle
- o cartoons by Judy Dunn
- o book reviews, a new word column, and much more

Website: http://www.ling.mq.edu.au/news/australian_style.htm

Contact: adam.smith@mq.edu.au