

Conditionals vs. Composition

Eric Armstrong
2 Sep 2008

Summary:

This article is part of a series that describes the 20-some decisions that face every DITA project. The goal is to identify the pros and cons for each decision and, where warranted, record the known "best practices" around each decision point. (Most of them can be covered in a single article. But a couple, like this one, are intricate enough to require an article of their own.) The series concludes by considering whether DITA would benefit from the creation of a specialization for a "Decision Guide".

Acknowledgments:

This series of articles was motivated, in part, by a round table discussion at the July 2008 meeting of the Silicon Valley DITA Interest Group (SVDIG). It records many of the thoughts that surfaced during the meeting about the kinds of decisions you need to make when starting a DITA project. Those thoughts combined nicely with information gathered in conjunction with the lead DITA architect for JavaSE docs, Sowmya Kannan. In the process, we leaned heavily on information gleaned from Alfresco's documentation manager, Briana Wherry, and her lead architect, Janys Kobernick.

Contents

- Introduction
- Using Conditionals
- Using Composition
- Comparing the Two Strategies
- A Hybrid Alternative
- Compound Conditions
- Simulating Metadata Hierarchy
- Improving the Review Process
- Automating the Composition Process

Introduction

The goal is to reuse a topic in different settings, where the topic has only minor differences in each setting. There are two strategies you can use to achieve that goal:

- **Conditionals:** Tag elements with conditional metadata.
- **Composition:**
 - Conref all 'conditional' elements from a definition file.
 - Substitute a different definition file at production time, which means that the conrefs are, in effect, variables.

I became intrigued in the subject of composition after hearing from people on the DITA-user mailing list. People who use it swore by it, considering conditionals as "old school" and "archaic". Those messages simply begged the question, "What's so good about composition?"

This article describes the two approaches, contrasts them, and attempts to answer that question.

MetaNote:

This article provides a basic template for a *decision guide*--an introduction that lists the choices, explanations that describe the options, and a section that compares the advantages and disadvantages of each choice, optionally followed by an explanation of "hybrid" alternatives that provide some combination of the advantages and disadvantages of the individual options. (In the last article of this series, I intend to explore the possibility of a decision-guide specialization for DITA.) Of the remaining sections of this article, some should be subheads of the composition section, and others should be independent topics that expand on entries in the comparison table. But the luxury of the article format is that I'm free to take liberties with the template, both for readability and for improved authoring speed--especially since that template hasn't been fully defined, as yet.)

Using Conditionals

Here's an example of a task step with conditionals:

```
<step><cmd>Go to the installation directory at
  <ph platform="solaris">/opt/product</ph>
  <ph platform="linux">/usr/product</ph>
  <ph platform="windows">C:\Program Files\product</ph>.
</cmd>
</step>
```

The good news when you're reviewing this topic is that everything is right there in front of you. The bad news is that when you need to make a change, you need to find all such occurrences, scattered throughout your topic set. And while you may know the location for one of the entries (say, the Solaris location), you may not have values for the other two, which leaves unknowns scattered throughout your topics.

Using Composition

Here is an example of the same task step, using composition

```
<step><cmd>Go to the installation directory at
  <ph conref="metadata_platform.dita#install_dir"/>.
</cmd>
</step>
```

It's easier to see the punctuation and spacing here. With conditionals, it's harder to make sure you've got those details right.

Composition works by substituting different files for `metadata_platform.dita`, all of which have an element in them that has the ID `install_dir`.

In this case, there would be 4 files. When writing, `metadata_platform.dita` would be a synonym for the variables file:

```
metadata_platform_variables.dita
  <ph id="install_dir">Installation Directory</ph>
```

(The synonym would be created using a symlink or by renaming files, depending on operating system characteristics.)

The writer then sees "Installation Directory" in the text (highlighted as a conref). They don't even need to know the location at that time, they just need a variable to transclude at that point in the text.

At production time, one of the values file is substituted to provide the appropriate text:

```
metadata_platform_solaris.dita
  <ph id="install_dir"/>/opt/product</ph>

metadata_platform_linux.dita
  <ph id="install_dir"/>/usr/product</ph>

metadata_platform_windows.dita
  <ph id="install_dir"/>C:\Program Files\product</ph>
```

Notes:

- A definition file is generally a generic topic, so you can put anything in it. (Anything with an ID is a "variable".)
- Anything that doesn't have grammatical variations, like a file path, can be in a <ph> element by itself. But anything that has grammatical variations (like product name) should be a complete sentence in a <ph> element, because only complete sentences translate well. (Even in English, when the plurality or part of speech changes, the change wreaks havoc on possessives and sentence structure.)
- When the specifics aren't yet known, values files can be populated with something like this:

```
<ph id="install_dir"/>__NOT YET KNOWN__</ph>
```

The values file then provides a checklist of important information that needs to be determined.

- In general, you will have one file set per metadata dimension--one for the variables and one for each of the possible values. You can then create combinations of metadata by varying the set of values files you supply.

For example, these two files would be used to create the JDK installation instructions for Solaris:

```
metadata_platform_solaris.dita
metadata_product_jdk.dita
```

- To be clear about the terminology I'm using, a *definition file* is either a *variable file*, or a *values file*. In this article, I use the term *values file* to mean, "a definition file that substitutes a set of values for a set of variables". But that choice overloads the term "values file", which also refers to a `.ditaval` file, in DITA. To disambiguate the term, I propose to call a `.ditaval` file what it really is: a *control file*.

That nomenclature has the advantage of maximum accuracy, because the prototypical definition of a "process" is something that has inputs, outputs, and controls--and the `.ditaval` file certainly acts as a process control. In addition, in a suitably clever implementation of the production system, the control file can be used to automatically select the values files to use during the production run. (That subject is covered in the final section of this article.)

Comparing the Two Strategies

Originally, the idea of using conditionals seemed like a "no brainer" decision. But after more reflection, I realized that with composition, there is no need to extend the metadata attribute set, no need to worry about metadata hierarchies, and no need to worry about boolean combinations of metadata. So after a closer inspection, composition seemed to have a lot going for it. The table below summarizes the differences I have been able to discern between the two approaches.

Characteristic	Conditionals	Composition
Authoring	All variations exist side by side, so they are easily compared, but multiple variations make a topic harder to read. Readability requires a sophisticated editor that does smart color coding.	There is only ever one conref'd value. A topic is easier to read. But it's harder to verify values are correct by inspection, as to how files are authored separately.
Review	Possible to create a version for review that shows all conditional variants--but identifying which variant is which is more difficult (metadata values need to be displayed).	Must review all variants of final publication for accuracy. (But a tool can be constructed for reviews. More on that subject coming.)
Version-based differences	Processing mechanisms for version-numbered conditionals let you select for all "version 6 and later" content.	No equivalent capability using composition.
Metadata Design	Metadata needs to be designed correctly at the outset to minimize the need for change.	Anywhere metadata would have been needed, it would have been created. Different substitution-sets (conrefs) are created to reflect different combinations of metadata. So you have one set of files for browser-specific values, one set for browser-specific values, and so on.
Metadata Change	When metadata changes, elements tagged with old metadata values need to be modified to use the new tags. If old conditionals remain in the topics, the content they tag may be quietly ignored or erroneously included.	Most changes restricted to definition files, not changing the source text. When the metadata changes, existing text can be updated with conrefs. If the change is such that variables are reorganized, then references to old files will break, which helps to ensure that the new files are used.
New Metadata Values	If a new platform needs to be supported (say, a Mac), then all places where text is tagged with "platform=X" must be found, and a new variant of the text added for "platform=mac". (Locations are naturally scattered through the doc set, and can only be found with a sophisticated metadata search.)	If a new platform needs to be supported, a copy of the closest definition file, and all affected variables--all of which are in one file.

Content Maintenance	When values change in the next release (say, the platform-specific names of installation files), then you have to find all text that refers to them and make the appropriate changes. If they are buried in multiple topics, it can be easy to overlook some.	Having all values in one file makes it easy to change. It also gives you a checklist, so you have gotten all the information you need for the next version.
Compound (boolean) Metadata	Conditional metadata does not support boolean combinations (e.g. platform=windows <i>and</i> browser=firefox).	Can simulate boolean logic using conditional files (more on that subject, later on).
Metadata Hierarchy	Standard metadata lacks hierarchy, but it can be simulated.	Some situations can be readily simulated, but others cannot (more later).
Training	Writers need to understand the metadata model to tag things correctly. They need to understand the metadata hierarchy, understand inclusion vs. exclusion, and know what substitutions to use for boolean combinations. --taken from Andrea Leszek's slides	Writers need to know location of definitions to know when to reference it in lieu of a definition.
Production	Create a single ditaval file for each combination of metadata attributes you need. Specify that file when doing production.	Specify a different set of definitions for each combination of metadata--a process that is prone to error, unless you automate it (more later).
Semantic Web and Automation	DITA-OT has been enhanced to allow metadata to "pass through" as class attributes, which produces HTML <i>microtags</i> . (Generally called "microformats"--a designation that fosters the misconception that the added attributes are somehow related to presentation format, and which doesn't reflect their true value as semantic tags.)	Joe Gelb comments: With composition, you can break up chunks of information into deliverable units, which are then processed by tools for indexing and fielded search or knowledge bases. It also gives the advantage of interoperating with standards and technologies developed for the semantic web.

Table: Comparison of Conditionals and Composition.

A Hybrid Alternative

Upon reviewing the original version of the comparison table above, IBM's Megan Bock offered this alternative:

"Use conditional metadata for topicrefs in maps, but use composition in topics, so there are no embedded conditionals in your topics."

With that strategy, you would still need to create specialized metadata, which negates some of composition's advantages. But you are left with maps that are more readable, since they don't contain references to "a topic or sub-map to be named at a later date". So the advantage is that your maps are less abstract and less complex. The disadvantage is that you still have to define the metadata, you just use it less.

There is one more significant advantage for this approach:

Conditionals let you solve the one kind of problem that composition can't touch:
The problem of multiple locations.

Consider this real-world example. The outline below comes from a map for an installation guide that covers both the Java Development Kit (JDK) and Java Runtime Environment (JRE). There is only one way to install the 64-bit supplement for the Java Runtime Environment (with an executable), but there are two ways to install it for the JDK, depending on how the JDK was installed. The same topic is therefore referenced in two different locations. In one location, it is nested. In the other, it isn't:

```
...
JRE: Installing the 64-bit Supplement (executable)
JDK: Installation Options
  Installing the 64-bit Supplement (executable)
  Installing the 64-bit Supplement (packages)
...
```

Here, the same topic is used in two different locations. Conditionals solved that problem handily.

Of course, it may be that you don't really *need* conditionalized maps. It's quite possible to create a different map for each deliverable you intend to produce. You give up some of the advantages of single-sourcing, but you do so only at a very high level, where it doesn't hurt very much.

This is an area that needs a razor--a way to decide between the two approaches. Conditionals in topics certainly appear to be more trouble than their worth, but when does it make sense to use them in maps, and when does it make more sense to create duplicate maps? (In the example above, there were about a dozen topics in the map, so it didn't seem to make sense to have two copies.)

Compound Conditions

Most of the time, one file per metadata dimension will be sufficient. So you might have sets of files like this:

- metadata_platform_variables.dita
- metadata_platform_solaris.dita
- metadata_platform_linux.dita
- ...
- metadata_browser_variables.dita
- metadata_browser_firefox.dita
- metadata_browser_opera.dita
- ...

Then, at production time, you do the appropriate substitutions to create the document appropriate for Firefox users on Linux, for example.

But every once in a while, you may find that you some boolean combination of metadata--a *compound condition*--where, for example, the substitution value for Solaris and Firefox differs from the value for Solaris and Opera.

Conditionals don't allow for that kind of capability, but composition does. To get that behavior, you would create a set of dual-dimension files like these:

- metadata_platform_browser_variables.dita

- `metadata_platform_browser_windows_firefox.dita`
...etc...

Authors would then need to know to look for some variables in the dual-dimension file, rather than in one of the single-dimension files.

Simulating Metadata Hierarchy

Ideally, it would be nice to have hierarchical metadata that looks like this: `solaris`, `solaris:32`, and `solaris:64`

where:

- Content elements are tagged with one of the three
- When producing a document for 32-bit Solaris, `solaris:32` is specified, but all items tagged `solaris` are automatically included.
- When producing a generic Solaris document, `solaris` is specified, and all items tagged `solaris:32` and `solaris:64` are included.

With conditional metadata, the closest we can come is to create metadata that looks like this: `solaris`, `solaris_32`, `solaris_64`. It looks similar, but it's not actually a hierarchy. Then:

- To keep authoring simple, content elements are tagged with one of the three, as before.
- When producing a document for 32-bit Solaris, both `solaris` and `solaris_32` are specified for inclusion.
- When producing a generic Solaris document, `solaris`, `solaris_32`, and `solaris_64` are all specified.

It's not exactly the same as a true hierarchy, but it works pretty much the same. The authoring task is no more difficult than it was before and as long as you get the production scripts right, you get the expected results.

With composition, the effect is somewhat harder to achieve--and some things can't be done at all. Imagine a topic that looks like this:

```
<step><cmd>Go to the installation directory at
  <ph conref="metadata_platform.dita#install_dir"/>.
</cmd>
</step>

<step conref="metadata_platform.dita#install_additional_64_bit_package"/>

<step>
  <cmd conref="metadata_platform.dita#platform_specific_install"/>
</step>
```

The idea is for that middle step to be included when you're doing a 64-bit install, but to be left out for a 32-bit install. That's easily done with conditionals. But it won't work with composition. To resolve the reference, you'll have to include an empty step, which won't read very well. So the best you could do here would be to create a step that says "Install additional packages, if any", and then include the phrase "No additional packages". It's ugly, but it would solve the problem, more or less. The alternatives are to restructure things so that the additional installation step is in a topic of its own (so the conditionals are confined to the map, as in the hybrid alternative), or else consider the situation an exception in which in-topic conditionals are required.

The situation represented by the third step can be handled with greater accuracy, but it still takes a bit of work. The idea is that the nature of the third step depends on whether you are doing a 32-bit or a 64-bit install. So your file substitution-set could look like this:

- `metadata_platform_variables.dita`
- `metadata_platform_solaris_32.dita`
- `metadata_platform_solaris_64.dita`

With that implementation, common `solaris` values would be duplicated in the two values files. Of course, if you had your heart set on avoiding duplication, it would be possible to do so. You would have the same variables file, and then divide the values into files for `solaris`, `solaris_32`, and `solaris_64`. You would then manufacture the substitution-file at production time by combining the values in the `solaris` file with the values from one of the other two files. (In practical terms, that's a lot of additional effort for results that are more difficult to predict. In most every case, it will make more sense to live with the duplication in return for a single, easily-reviewed values file. But it's an interesting thought-experiment to imagine how the problem *could* be solved, if you needed to.)

Improving the Review Process

The goal is to display all possible values of a conref, to provide the same kind of review capability for topic composition that you get with conditional metadata (the ability to see all possible values in one place). To do that, we can use the DITA-OT to generate an output that shows all possible values for each conref, tagged with the metadata that produces it.

In the three platform example given above, the copy published for review would look like this:

1. Go to the installation directory at
 - solaris:** `/opt/product`
 - linux:** `/usr/product`
 - windows:** `C:\Program Files\product.`

To produce that kind of output, we need to combine all elements with the same ID from multiple values files into a single `<ph>` element, coupled with an identifier (`solaris`, `linux`, etc.) for each value. Substituting that file at production time displays all possible values, nullifying the advantage that conditionals would otherwise have in this area.

Here is an outline of the procedure:

- Given a file named `metadata_platform_variables.dita`:
 - Start a new file called `metadata_platform_tagged_values.dita`
 - For each `<ph>` element in the variables file with an ID, add to a new `<ph>` element to the target file that looks like this:

```
<ph id="xyz">
  </ph>
```

where the ID is the same as the ID in the variables file
- For all files named `metadata_platform_*.dita`:
 - Extract the metadata `METADATA_VALUE` from the name of the file (represented by the wildcard, `"*"`).
 - For each `<ph>` element with an ID in the file

- Extract the CONTENT from the element
- Add the tagged value to the <ph> element in the tagged_values file:


```
<ph id="xyz">
  <!-- break !--> <b>#{METADATA_VALUE}</b> #{CONTENT}
</ph>
```

where:

- The processing instruction identifies a place to create a line break
 - "`#{X}`" is the Ruby syntax for string interpolation. It says to insert the value of variable X into the string.
- To produce a copy for review:
 - Generate HTML, substituting `metadata_platform_tagged_values.dita` in the processing stream
 - Convert the processing instructions to `
` tags in post-processing.

Automating the Composition Process

If multiple values files need to be substituted at production time, it makes sense to automate the process. That way, you can ensure that the correct files are substituted every time you produce a given deliverable.

In an ANT script, the substitutions could be made at the start of a task. When invoking the OT from the command line, a wrapper script can be created that does the substitutions before invoking the OT.

In either case, the substitutions need to match the data specified in the `.ditaval` file that drives the production (if there is one). It would even be possible to create the substitution set by examining the `.ditaval` file.

To do that, it's necessary to have a naming convention for the values files. Using the naming conventions described in this article, the process would need to:

1. Examine ditaval file, extracting the metadata property name and associated value for all entries that specify "include"
2. Look for the corresponding metadata files.
3. Do the appropriate renames.

Given this `.ditaval` specification:

```
<val>
  <prop att="platform" val="opensolaris" action="include"/>
  <prop att="browser" val="firefox" action="include"/>
</val>
```

The script would substitute files of the form `metadata_<property>_<value>` for each entry. So it would look for `metadata_platform_solaris.dita` and `metadata_browser_firefox.dita`.

Although it is somewhat more difficult to do so, the script also need look for compound metadata files of the form `metadata_<property1>_<property2>_<value1>_<value2>.dita`. In this case, it might need to substitute a file named `metadata_platform_browser_opensolaris_firefox.dita`

Of course, while that level of automation is interesting to contemplate, it is probably overkill in nearly every

case. A few lines in a script or ANT task that does the substitutions is all that is really necessary, most of the time.