DITA Configuration and Specialization Tutorials

Contents

DITA Configuration and Specialization Tutorials

This information set provides a set of tutorials on DITA 1.2 configuration and extension. These tutorials replace the DITA 1.1 "Specialization Tutorials" originally published in 2007.

These tutorials are excerpted from the upcoming book *DITA for Practitioners*, to be published by *XML Press* in 2011.

You can find the working results of all the tutorials at *http://www.xiruss.org/tutorials/materials/dita-tutorial-materials.zip*.

This tutorial is published in the following versions in addition to the HTML version served at *xirus.org/tutorials/dita-specialization*:

- PDF: *http://xiruss.org/tutorials/dita-specialization/dita-specialization-tutorial.pdf*
- EPUB: *http://xiruss.org/tutorials/dita-specialization/dita-specialization-tutorial.epub*
- Kindle (Mobipocket): *http://xiruss.org/tutorials/dita-specialization/dita-specialization-tutorial.mobi*

The tutorials assume you have a copy of the DITA Open Toolkit and an XML-aware editing environment of some sort. I generally recommend *OxygenXML* as a high-value XML and DITA development environment but you can of course use any comparable tool set.

The tutorials are arranged in order from least-involved to most-involved. If all you need to do is create a local document type shell, you can jump right to *Document Type Shell Tutorial* on page 27.

About These Tutorials

These tutorials are authored as a set of DITA 1.2-conforming maps and topics using topic types from the *DITA for Publishers* project as configured for use in the *DITA for Practitioners* book from which these tutorials are drawn. I use *OxygenXML* as my day-to-day authoring environment. The HTML, PDF, and EPUB versions were produced using the *DITA Open Toolkit*. The EPUB plugin is currently available through the DITA for Publishers project.

Please report any comments, bug reports, or technical inaccuracies to me at *drmacro@yahoo.com*. Please include "DITA Tutorial" in the subject line so I know it's probably not spam. I'm also happy to discuss any aspect of the tutorial on the DITA users Yahoo! group, *dita-users@groups.yahoo.com*.

Setting Up Your Development Environment

There are many ways to set up a productive DITA development environment. This chapter describes a set of tools that work well for the author and that the various tutorials and how-to sections are defined in terms of.

The development and maintenance of DITA systems requires the following types of tools:

- An XML-aware editing environment for working with DITA documents, DTDs, XSDs, XSLT, XQuery, and other XML-specific artifacts. This book recommends the OxygenXML product.
- A Java development environment for compiling and deploying Java code (optional). Many DITA systems never require the implementation of custom Java code but it can still be useful to be able to examine or modify the Java code that underlies the Open Toolkit. If you are integrating with a content management system you will likely need to write custom Java code. This book specifies the Eclipse IDE, although similar systems, such as NetBeans, are comparable. You will need Eclipse if you need to deploy Eclipse Infocenters.

- A Web browser.
- Output-specific viewers, including PDF readers, EPUB readers, Windows help viewers, etc.
- A source code control system, such as Subversion, GIT, or VCC, for managing your project's source components. Can also be used to manage DITA content quite effectively.

  Note: You should not use CVS with DITA content because it cannot handle Unicode documents properly (CVS is ASCII only).

General DITA development and the tutorials in this book require the following specific tools:

- A Java development kit (SDK), which is required by the DITA Open Toolkit and Java development. At the time of writing the latest Java version is Java 7 but Java 6 is the version required by most of the tools you will be using. You should use Sun (Oracle) Java, as some tool components will not work with other Java distributions.
- The DITA Open Toolkit, for testing and validating DITA processing, even if your system does not use the Toolkit itself.
- Apache Ant. Ant is a general-purpose scripting system used to manage the building and deployment of software projects. It is used by the DITA Open Toolkit and is generally useful for lots of tasks. While Ant is included in the main Open Toolkit package it is useful to have a separate standalone copy.

XML-Aware and DITA-Aware Editing Environment

At the time of writing I use the OxygenXML editor as my primary XML development environment and heads-down authoring tool. As of the time of writing, OxygenXML provides unmatched support for DITA 1.2 features. It integrates with the DITA Open Toolkit and is as easy to configure for new DITA vocabulary modules as it is possible for a DITA-aware tool to be. For those reasons I strongly recommend OxygenXML as an XML and DITA development environment. While OxygenXML is a commercial product and is not free, it offers a tremendous value in terms of the features provided. OxygenXML is provided as a standalone tool and as an Eclipse plugin. The two versions of OxygenXML are functionally equivalent and the same license will work for either. I personally use the standalone version of OxygenXML for largely historical reasons and also to avoid having to close down my OxygenXML session when I have to close Eclipse, which is fairly often due to the type of development I typically do.

You can also use Eclipse alone for DITA-aware XML development using Eclipse's various XML-related support components and available plugins. Eclipse can be integrated with the Open Toolkit so that you can run the Toolkit from within Eclipse (see *Setting Up Eclipse to Work With the Open Toolkit* on page 8).

Other XML development environments, such as XML Spy, can of course be used to develop DITA-related components but they do not, at the time of writing, offer the same degree of built-in DITA support or integration with the Open Toolkit as provided by OxygenXML.

Configuring OxygenXML For DITA Development

To make OxygenXML ready for DITA development you need only do one thing after installing the product: Turn on use of external DTDs for determination of document type association. You also have a choice in how you use OxygenXML in relation to the DITA Open Toolkit.

OxygenXML has a general feature by which it can determine the document type associated with a given document, and thereby determine the set of features to use for that document. OxygenXML has built-in DITA-specific features and it will turn those on for any document that it recognizes as being a DITA document, regardless of the specific vocabulary the document uses. It does this by looking for the `ditaarch:DITAArchVersion=` attribute, which all conforming DITA documents must have and which serves to unambiguously identify a document as being a DITA document.

However, most DITA documents do not literally specify the `ditaarch:DITAArchVersion=` attribute because it is defaulted in the DTDs and schemas. Thus, in order to recognize a document as a DITA document the document must be parsed with respect to its DTD or schema.

OxygenXML can do this but it does not do it by default, because there is a performance cost in parsing documents in order to then determine what general type of docuemnt they are. So you must turn this feature on in

order for Oxygen to automatically process your specialized DITA documents, or even documents that use local document type shells, as DITA document.

To turn this feature on go to Preferences->Document Type Association to bring up the Document Type Association panel. On that panel make sure that the "Enable DTD/XML Schema processing in document type detection" check box is checked and that the "Only for local DTDs/XML schemas" checkbox is checked. Save the new preference settings.

Once you have made this change, OxygenXML will automatically apply to DITA-specific editor features to all DITA documents regardless of specialization.

OxygenXML includes a copy of the Open Toolkit, usually the latest released version current at the time the OxygenXML version is released. The OxygenXML-provided Toolkit includes some small patches to the Toolkit that improve it's interaction with OxygenXML but that are not essential for use of the Toolkit with Oxygen.

OxygenXML uses, or can use, the master entity resolution catalog that is maintained by the Open Toolkit to resolve DITA DTDs and schemas. If you deploy your local shells and specialized vocabulary modules to Oxygen's built-in Toolkit then your DITA documents that use those shells and modules will just work with no additional configuration actions required. Because OxygenXML is fully specialization aware it provides all its out-of-the-box DITA features to all DITA documents once it is able to parse them, which it can do if the DTDs and schemas are resolvable.

OxygenXML's built-in Open Toolkit is in `frameworks/dita/DITA-OT` below the OxygenXML installation directory.

To ensure that the master Toolkit catalog is being used, go to Preferences->Document Type Association to bring up the Document Type Association panel. Select the entry for "DITA" and select the "Edit" button to edit the document type association. Select the "Catalogs" tab to see the list of associated catalogs. You should see an entry for "${frameworks}/dita/catalog.xml". The string "${frameworks}" is a reference to the OxygenXML-defined variable that resolves to the location of the OxygenXML `frameworks` directory. If you want Oxygen to use a Toolkit installed at a different location on your computer you can change this setting to reflect the location of that Toolkit.

Depending on your Toolkit needs you have three choices for how to use OxygenXML with the DITA Open Toolkit:

1. You can use the OxygenXML-provided Toolkit and simply deploy new plugins to it as needed. This ensures that you get full advantage of OxygenXML's patches to the Toolkit. This limits you to the use of the Toolkit version OxygenXML provides out of the box.
2. You can replace the OxygenXML-provided Toolkit with another Toolkit version in the same location on the file system. This loses the OxygenXML Toolkit patches but means all existing Toolkit-related transformation scenarios and catalog configurations will work without modification.
3. Use a Toolkit installed elsewhere on your computer and set the value of the dita.dir parameter used in OxygenXML transformation scenarios as well as updating the catalog configuration for the DITA document type association. This preserves the OxygenXML-provided Toolkit but requires changes to any transformation scenarios you want to use with the non-Oxygen Toolkit.

I use option (2) most of the time because I find it easiest and most reliable to have my Toolkit be in an invariant location and use Ant scripts or manual moving and copying to configure that Toolkit instance with whatever I need at the moment. This reflects in part the fact that my daily job involves working with many different Toolkit configurations and versions for specific clients. I find it easier to have a consistent process and supporting scripts (Ant scripts in my case) that swap into the that one location the Toolkit versions and plugins I need for a specific project. It allows me to have exactly one Toolkit location that all my Toolkit-related activities use, removing the chance of accidentally doing things in the wrong location.

If you are working with just one Toolkit version or configuration most or all of the time but you can't use the OxygenXML-provided version for whatever reason (for example, you need to use an older version or newer version of the Toolkit) then option (3) is probably the best practice since you only have to configure the DITA document type association and each transformation scenario once to reflect the Toolkit location. It does mean that you have to remember to change the default value for the dita.dir parameter whenever you create a new DITA transformation scenario.

Setting Up Eclipse to Work With the Open Toolkit

Eclipse is a general development environment that offers many handy features for doing DITA development, including the ability to develop and run Ant scripts. Eclipse includes both an Ant editor for editing Ant scripts and an Ant "view" that makes it easy to run different Ant scripts.

I use Ant scripts from Eclipse mostly for development related activities, such as scripts to manage source code compilation, packaging, and deployment of project components to my local working environment. For example, for all my DITA-related projects I have a standard Ant target that deploys that project's Toolkit plugins to my local Toolkit instance (which is normally the Toolkit that Oxygen uses).

Setting up Eclipse so it can run the Open Toolkit's Ant scripts you can also use Eclipse to run Ant scripts that run the Toolkit. For example, you can create Ant scripts that will run multiple Toolkit transformations on a given input DITA map or process a whole set of documents and then do something with the generated results, such as publish them to a Web site.

To set this up you must add the Toolkit's custom Java JAR files to the list of Jar files that Ant uses when it is run by Eclipse. This does essentially the same thing that the Toolkit's startup.sh and startup.bat scripts do, namely set up the appropriate Java class path so the custom Toolkit Ant processing will work.

You must also configure an ant property that specifies the location of the Open Toolkit on your machine. By convention I call this property *dita-ot-dir* and that's the name used in all the samples in this book. While you can hard-code this property in the various Ant scripts you use, it's much better to set up your Ant scripts to get the property from a separate configuration file so that you can change it one place. This also makes it easier to do collaborative development of DITA projects where each developer needs to have a different value for the Toolkit location but the scripts are managed in a common code repository.

To set up a property file for defining the *dita-ot-dir* Ant property, do the following:

1. Create a file named `build.properties` or `.build.properties` your home directory. If you are on a Unix or Linux system, using `.build.properties` offers more security because the file will be hidden by default, making it safer to hold sensitive values like passwords.
2. Edit the `build.properties` file and add a line like this:

   ```
   dita-ot-dir=/Applications/oxygen/frameworks/dita/DITA-OT
   ```

   Where the value to the right of the "=" is the actual location of the Open Toolkit you want to use on your machine.
3. In your Ant scripts include these three lines before any other property definitions:

   ```
   <property file="build.properties"/>
   <property file="${user.home}/.build.properties"/>
   <property file="${user.home}/build.properties"/>
   ```

   These three property file inclusions will look for `build.properties` in the same directory as the Ant script itself, `.build.properties` in your home directory, and `build.properties` in your home directory, in that order. Whichever of these files has the first definition of a given property will set the value of that property. This organization allows you to have global defaults for properties in your user-specific properties file and override those defaults in a project-specific properties file.

To add the Toolkit-related JAR files to Eclipse's Ant configuration, do the following:

1. Go to Preferences->Ant->Runtime to bring up the Ant runtime settings. Select the Classpath tab.
2. Select "Global Entries" item in the classpath list. Expand the item to see what JAR files are already listed.
3. Assuming that the Toolkit's JAR files are not listed, with "Global Entries" selected in the tree view, select "Add external JARs" and navigate to the `lib` directory underneath the Toolkit installation you want to use (e.g., the Toolkit installed with OxygenXML).
4. Select all the .JAR files in the lib directory add them to the list under "Global Entries".
5. Select "Add external JARs" again and navigate down into the `saxon` directory under the `lib` directory.
6. Select all the .JAR files in the `saxon` directory and add those to the list under "Global Entries".

You may also need to add the two jars "xercesImpl.jar" and "xml-apis.jar", which are part of the Apache Xerces2 Java project. If you don't already have them on your system somewhere, you can download them from *http://xerces.apache.org* and put them in the `lib` directory of your Toolkit installation.

With this Ant configuration in place you should be able to run Ant scripts that use the Open Toolkit from within Eclipse.

This Ant script tests the ability to run the Toolkit from Ant:

```
<?xml version="1.0" encoding="UTF-8" ?>
<project name="dita.build.demo" default="demo.book" basedir=".">
  <!-- Simple Build script to test the ability to call the
       DITA Open Toolkit from Ant.
    -->

  <property file="build.properties"/>
  <property file="${user.home}/.build.properties"/>
  <property file="${user.home}/build.properties"/>

  <property name="dita-ot-dir" location="c:\DITA-OT1.5"/>
  <property name="dita.demo.book.dir"
      value="${dita-ot-dir}${file.separator}samples"/>

  <property name="dita.output.demo.book.dir" location="${basedir}/temp/
demo"/>

  <target name="demo.book"
    description="build the book demo">
    <mkdir dir="${dita.output.demo.book.dir}"/>
    <ant antfile="${dita-ot-dir}/build.xml"
      target="dita2xhtml">
      <property name="args.input"
        value="${dita.demo.book.dir}${file.separator}taskbook.ditamap"/>
      <property name="output.dir"
        value="${dita.output.demo.book.dir}"/>
      <property name="transtype" value="xhtml"/>
    </ant>
  </target>

</project>
```

To run this script yourself you will need to either create a `build.properties` file that sets the value of the *dita-ot-dir* Ant property or change the value in the script itself. (The first definition of a property is the effective value in Ant, so a value for *dita-ot-dir* set in one of the included `build.properties` file would take precedence over the value specified in the Ant script itself.

If you want to be able to run this type of Ant script from the command line outside of Eclipse you must either do so from a command window created using the Toolkit-provided `startup.sh` or `startup.cmd` script or you must configure Ant's classpath using normal Ant configuration facilities. For this type of configuration there are many possible approaches, all of which are beyond the scope of this book.

Setting Up OxygenXML for DITA Editing

OxygenXML can automatically detect that a document is a DITA document as long as you have turned on Oxygen's ability to look inside DTDs and schemas to determine document type association. This setting is not turned on by default through Oxygen version 11.

To turn this feature on, do the following:

1. Go to Preferences -> Document Type Association
2. Check the box "Enable DTD/XML Schema processing in document type detection"
3. Check the box "Only for local DTDs/XML schemas"

With DTD processing turned on for document type detection, Oxygen will recognize any DITA document as a DITA document regardless of what document type shell it uses as long as the `dita:DITAArchVersion=`

attribute is present. This will cause Oxygen to turn on it's built-in DITA-specific editing features, allowing you to edit any DITA map or topic document with full features.

Understanding Configuration and Specialization

Before you can design and implement your own configurations and specializations you must have a basic understanding of what configuration and specialization are and do.

DITA is designed specifically to allow the definition of new markup and new document types in a way that preserves the ability for any general-purpose DITA processor to usefully process documents that use the new markup. This in turn enables blind interchange of DITA documents, because any DITA user knows that they can use and process, at least minimally, any other conforming DITA documents they get from any source. In particular, in the context of a map, you can combine together topics of any type and know that you can get useable, if not optimal, results from any general-purpose DITA processor.[1]

This feature of DITA, the specialization feature, the ability to have your own markup design while still ensuring blind interchange of your content with other DITA users, is unique among all currently-existing standard XML applications (and most, if not all, private XML applications).

Specialization is the one truly unique and distinguishing aspect of DITA. No other aspect of DITA is exclusive to DITA. All of DITA's modularity features—maps, topics, key-based addressing, etc.—can either be found to one degree or another in other XML applications or could be added to those applications without too much trouble. This is not to discount the value of these features of DITA—they represent very deep thought and years of practical experience and are quite valuable in themselves, but they are not distinguishing in the way that specialization is.

Even if you have no use for any aspect of DITA having to do with modularity or reuse you still have a use for specialization simply because it enables reliable interchange in a way that no other XML application does. Even if your only interchange partner is your future self, DITA still offers dramatic and compelling advantages.

In short, one can see DITA as an architecture for the management of XML vocabularies.

These tutorials show you how to apply the architecture to specific types of requirements.

XML Vocabularies and their Management

In general XML parlance a "vocabulary" is a set of element types and attributes designed to be used together for some purpose. A given vocabulary may be "encompassing", meaning that it is intended to be used as the main or only vocabulary for a given document, or "enabling", meaning it is intended to be integrated into and used with encompassing vocabularies.

Examples of encompassing vocabularies are DocBook, XHTML, and NLM. Examples of enabling vocabularies are MathML and Dublin Core metadata.

A challenge historically with managing XML (and SGML) vocabularies is that, while it's easy to define enabling vocabularies like MathML and possible to define "extensible" encompassing vocabularies like DocBook, there was no standard-defined mechanism for managing how encompassing and enabling vocabularies are combined or extended in a way that ensured understandability and interchange.[2]

---

[1] I have to say "general purpose" processor because it's possible to have conforming DITA processors that only understand specific markup vocabularies and are not specialization aware. Such processors are probably rare but they are explicitly allowed by the DITA 1.2 conformance clause. Most DITA-aware tools you will find are both general-purpose and specialization-aware, meaning that they can handle, to at least some minimal degree, all conforming DITA documents, specialized or not.

[2] The HyTime standard, ISO/IEC 10744, provided a mechanism for vocabulary management but was not widely adopted before XML made it obsolete. You can view DITA's specialization feature as the XML reincarnation of the architectural form facility of HyTime. DITA's specialization feature was directly influenced by HyTime and how we had applied it within IBM to the IBMIDDoc SGML application.

In particular, before DITA, all mechanisms for combining or extending vocabularies were entirely syntactic—they provided no way to examine a *document* and know how that document's vocabulary (it's document type) related to any known vocabulary so that you could know, for sure, whether your processing environment could handle it or if you could share content from that document with your documents.

For example, DocBook provides for extension by providing the syntactic hooks needed to allow local modification to content models (e.g., parameter entities in DTDs). This allows you to define your own element types and use them in a nominally "DocBook" document. However, having defined your new tags, there's nothing *in the markup* that tells a processor or an observer how your new element type relates to any known element types (that is, to the element types defined in the DocBook standard or defined in any other DocBook-based document type). Thus, there's no reliable way for a general-purpose DocBook processor to know what to do with your document. Thus, to say that your document is "DocBook" is not accurate or useful. Rather, at best your document is "DocBook based". But knowing that doesn't tell you anything particularly useful. In particular, it doesn't tell you what you'd need to know in order to process it reliably (because you have no way to know what to do with the non-DocBook-defined elements in the document without actually talking to the developer of the custom markup).

By the same token, there are no DocBook-defined constraints on *how* you can extend DocBook so there's no way to predict what sort of changes a given "DocBook" document might reflect so that, for example, a general-purpose DocBook processor could provide useful fallback processing or so that a human observer can understand the nature of the extensions even if they don't understand the details of the new markup (for which they would need some form of documentation).

That is, using only the syntactic tools provided by DTDs or XSD schemas (or any other available form of XML document constraint, such as RelaxNG) extension and customization of XML vocabularies is inherently *unmanageable* because there is no machine-processable mechanism for communicating or understanding the relationship between any two vocabularies.

If customization is not manageable then the only way to ensure interchange is to disallow customization. This was the approach taken by "interchange" document types, such as ATA 2100. But of course invariant vocabularies suffer a number of serious and fatal problems. They tend to become very large because they must reflect a union of the requirements of all current and expected interchange partners. They tend to not satisfy key requirements of individual interchange partners because you can never put everything in or because local requirements are at odds with interchange requirements (for example, markup that is specific to a given company's internal business processes, which might be trade secrets). I think it's fair to say that, as an industry, we've conclusively proved over the last 20 years or so that monolithic interchange document types do not work.

DITA Vocabulary Management: Modules

DITA addresses this lack of manageability by providing features that make vocabularies manageable while avoiding the inherent problems of monolithic interchange document types.

DITA essentially turns the problem on its head. Rather than having invariant monolithic document types, it provides invariant vocabulary "modules" that can be combined together to form an infinite variety of specific document types and that can themselves be extended, in a controlled fashion, to create new vocabulary modules.

Likewise, vocabulary modules can be locally configured through "constraint" modules, which enable the customization of content models and attribute lists but only in ways that guarantee interchangeability and processability.

DITA defines a markup-based declaration mechanism that makes the nature of any configuration or extension machine-understandable. That is, a processor, looking at any conforming DITA document, can know exactly what vocabulary modules it uses and, if constraints have been applied, what those constraints are. For any given DITA element, a processor can know what standard-defined DITA elements it is based on, and thus how to apply at least some minimal DITA-defined processing to those elements.

These declaration mechanisms are the `class=` and `domains=` attributes. These attributes are the "magic" of DITA that make everything work. These attributes, along with some essential rules for vocabulary composition, make it possible for any general-purpose DITA processor to reliably process any DITA document no matter how it has been configured or extended. Likewise, a human observer of the document can know how its markup

relates to other DITA markup and can look at any given DITA vocabulary module and know how it relates to other modules.

Thus you can think of DITA as an unbounded set of vocabulary modules and a set of tools for combining those modules into specific document types suited to specific requirements.

DITA Document Types, Configuration, and Specialization

In DITA a "*DITA document type*" is nothing more or less than a unique set of vocabulary and constraint modules used together in a document. For example, a `<concept>` document that uses the highlight and indexing domain modules (and no others) reflects the DITA document type consisting of the concept topic type module and the highlight and indexing domain modules. This combination can be expressed by the string "topic concept hi-d indexing-d", read as "the concept topic type, which extends the topic topic type, integrated with the highlight and indexing domains".

This simple list of module names tells you everything you need to know in order to know what the processing requirements for the document are and whether or not elements from another DITA document are or are not consistent with the elements in this document.

In DITA documents this declaration of the set of modules used is specified by the required `domains=` attribute. e.g.:

```
<concept id="topic-id"
  domains="(topic concept hi-d indexing-d)"
>
 <title>My Concept</title>
</concept>
```

Note that you don't need the actual DTD or XSD declarations for the modules, you only need to know the module names.

One implication of this is that DITA documents do not need to have literal DOCTYPE declarations or XSD schema associations *as long as* they specify the set of vocabulary modules they use. Likewise, when a document does have a DOCTYPE or schema assocation, it doesn't matter *what* DTD file or XSD document it uses as long as that DTD or XSD accurately reflects the set of modules the document declares it uses.

This means that DITA processors should *never* depend on the use of a specific DTD or XSD file because the use of a specific file means nothing. Two DTD or XSD document type shells that reflect the same set of modules define *identical* DITA document types. This is a fundamental difference between DITA and traditional XML and SGML applications, where the *only thing* you could know for sure was the specific DTD or XSD file a document used.

For this reason, any system that claims to be a general DITA-aware processor that also requires or expects the use of specific DTD or XSD files is fundamentally broken because it demonstrates a lack of understanding of how DITA document types work.

(But do keep in mind that the DITA way of viewing document types is so different from traditional XML practice that it's no surprise that tools and many practitioners would get it wrong, especially tools that reflect an SGML heritage, where the DTD was everything. Unfortunately, some of these tools reflect unfortunate architectural decisions made decades ago that are difficult or impossible to undo in order to fully support DITA's way of thinking about document types. That doesn't mean those tools are not useful or even compelling, just that they will be harder to adapt to locally-defined document types and non-standard-defined vocabulary modules.)

There are three types of modules that can be used to define a DITA document type:

- Structural modules, which define map types or topics types ("map", "bookmap", "topic", "concept", etc.)
- Domain modules, which define sets of elements usable across map or topic types (the highlighting domain, the programming domain, etc.)
- Constraint modules, which restrict the content models or attribute lists of specific element types within a specific structural or domain module, for example, the strict task constraint module that takes the DITA 1.2 general task topic type and restricts it to match the rules of the DITA 1.1 task topic type.

In this module-based approach to vocabulary management there are two things you can do to create DITA document types: *configuration* and *specialization*.

The DITA standard defines specific structural, naming, and coding requirements for document type shells and modules that help ensure consistency of design and implementation and make it easy to combine modules into new document types. While these patterns are not strictly needed technically (they have no bearing on the syntactic validity or processability of DITA documents), they make it easier to use and re-use modules and generally keep things consistent. Once you understand the patterns and how the pieces fit together, you will see that creating new specializations and configurations is remarkably easy.

DITA is about interchange and that includes interchange of knowledge and interchange of implementation components, as well as interchange of content. DITA's modular vocabulary approach is designed in part to make the interchange of vocabulary as reliable as the interchange of content. A large part of this is simply standardizing implementation details so that having learned how DITA vocabulary implementation works you should be able to quickly apply that knowledge to any conforming DITA vocabulary, no matter how specialized.

Configuration

Configuration is the task of taking existing vocabulary and constraint modules and combining them together to define a specific DITA document type.

You do configuration by creating new document type shells, that is, DTD or XSD files that serve essentially as a manifests of the vocabulary modules that make up the DITA document types.

Configuration can also involve the creation of new constraint modules.

As an implementation activity, the creation of new document type shells is an entirely mechanical process that anyone can perform even if they have no knowledge of DTD or XSD syntax. These tutorials demonstrate the mechanical process. Likewise, because the process is entirely mechanical (meaning it requires no creative thought or invention), it can be automated, as it has been by Jarno Elovirta and his DITA DTD Generator (*http://dita-generator.appspot.com/*).

The development of constraint modules requires a bit more DTD or XSD knowledge, but it is also a largely mechanical process because it is always about removing or constraining existing things, not adding new things, so it does not require invention, only analysis of requirements and modification of existing declarations.

Specialization

Specialization is the process of creating new structural or domain vocabulary modules that provide new markup for specific requirements.

The essential aspect of specialization is that every element type or attribute defined in a vocabulary module must be based on and consistent with an element type or attribute defined in a more-general vocabulary module or in the base topic or map type.

This requirement ensures that any element, no matter how specialized, can always be mapped back to some known type and therefore understood and processed in terms of that known type. This ensures that all DITA documents, no matter how specialized, can always be processed in some way. That is, new markup should never break existing specialization-aware DITA processing.

Every element type exists in a *specialization hierarchy*, which goes from the base module (topic or map) through any intermediate modules to the element itself.

For example, if you defined a specialization of `<concept>` called `<myConcept>` it's specialization hierarchy would be `<topic>` -> `<concept>` -> `<myConcept>`. A processor given a `<myConcept>` document would be able to process it either as a concept topic or as a generic topic, as appropriate.

The magic of specialization is the `class=` attribute.

Every DITA element must have a `class=` attribute. The value of the class attribute is the specification of the specialization hierarchy for the element. The syntax of the `class=` attribute is:

- A leading "-" or "+" character: "-" for structural types, "+" for domain types.
- One or more space-separated module/element-type pairs: "topic/p", "topic/body", "hi-d/i", etc.

- A trailing space character, which ensures accurate string matching on the last term in the hierarchy

For the <myConcept> topic type the class= value would be

```
"- topic/topic concept/concept myConcept/myConcept "
```

Which you read right to left as:

> The <myConcept> element in the "myConcept" module, which specializes <concept> from the "concept" module, which in turn specializes <topic> from the "topic" module.

If the <myConcept> topic type defined a specialized body element, say <myConceptBody>, then it's class= value would be:

```
"- topic/body concept/conbody myConcept/myConceptBody "
```

Looking at an instance of the <myConcept> element you would find these class= attributes:

```
<myConcept id="topicid"
  class="- topic/topic concept/concept myConcept/myConcept "
>
  <title>My Concept</title>
  <myConceptBody
    class="- topic/body concept/conbody myConcept/myConceptBody "
  >
</myConcept>
```

Note that these are attributes of element instances. While we tend to think of the class= attribute as something that is set in DTDs or XSDs, that is merely a convenience. What's really important is that the attributes are available to XML processors, which will be the case whether they are defaulted in DTDs or specified explicitly in instances—the two are identical to XML processors.

The magic of the class= attribute is that specialized DITA documents can "just work" when processed by general-purpose specialization-aware processors, such as the DITA Open Toolkit.

One implication of this magic is that you can define new markup without the need to also implement all the different forms of processing that might be applied to that markup—it will just work. To the degree that your specialized markup doesn't require any specialized processing, then you will *never* need to implement any new processing for it.

If your specialized markup does require specific processing, DITA-aware tools will tend to make adding that processing easier because they tend themselves to be modular. For example, the DITA Open Toolkit provides a general plugin mechanism that makes it easy to implement and deploy specialization-specific processing that extends the out-of-the-box processing using the smallest amount of custom code possible.

The Cost of Interchange Enablement and the Overall Value of DITA

There is of course no free lunch. DITA's interchange features do have a cost, namely the imposition of some general constraints and rules that are necessary to ensure the system works.

In particular, a given DITA element must be at least as constrained as its immediate base type. This means, for example, that if the base type requires an "A" element followed by a "B" element then any specialization of the base type must provide "A" or a specialization of "A" and must require it to be followed by "B" or a specialization of B.

For example, because the content model for <topic> requires <title>, all specialized topic types must also require <title> or a specialization of <title> and must require the title element to be the first child of the topic element.

This means that markup designers do not have completely free reign to structure new markup designs however they might want to. It also means that it will not always be possible to make an existing non-DITA vocabulary into a DITA vocabulary simply by adding the appropriate class= attributes, because the existing vocabulary may not align structurally with the appropriate base DITA type.

For example, the general DocBook model for titled divisions does not include an element type that corresponds to the DITA `<body>` element, which DITA requires be used to hold the direct content of topics. Thus DocBook divisions are not structurally compatible with DITA topics.

The "at least as constrained" requirement also means that elements designed to be the basis for further specialization need to allow appropriate options in their own content models so that they don't prevent reasonable specialized designs. This is why the content models for most of the topic elements are so loose: they have to allow a wide range of possible specializations. It is important to remember that the base standard types were *not* intended to be directly used for authoring. They were intended to be specialized or constrained as appropriate for specific authoring use cases.

For example, while `<body>` within `<topic>` allows an unconstrained mix of block elements and `<section>` elements, `<conbody>` within `<concept>` only allows block elements before any `<section>` elements. But another specialization of `<topic>` might need to continue to allow a mix of block elements and `<section>` elements, so the constraint in `<conbody>` would be inappropriate in `<body>` because it would prevent other legitimate ways of organizing topic content in other specializations.

It's also important to remember that DITA, while developed originally for technical documentation, is not specific to technical documentation and therefore should not reflect markup design decisions that reflect editorial practice specific to technical documentation. DITA is a completely general XML application framework and must therefore accommodate all legitimate requirements within the general constraints of enabling interchange and interoperation.

To experienced XML practitioners the "at least as constrained" requirement may seem like a particularly onerous, or at least annoying, constraint, and it can be frustrating to realize that the way you would have designed a particular bit of markup in the past simply cannot work in a DITA context. It means that you cannot always create a blindly-literal mapping from legacy non-XML structures into DITA structures but will have to work out some amount of structural reordering and a bit of retraining of the people transitioning from the old system to the new. It means you will have to learn the basic DITA structural patterns in order to know how to translate specific requirements into conforming DITA markup designs.

But the important question is not about cost but about value.

The value of DITA is that it *dramatically* lowers the overall cost of system implementation, use, and maintenance while increasing the inherent value of content by enabling blind interchange over the broadest possible scope. This lower overall cost far outweighs the direct cost of specialization, while the increase in value of DITA-based content increases the value of the overall system. Even if a DITA-based implementation were no less expensive to initially implement than the equivalent non-DITA-based system, it would still have greater value because the content would have greater value and the ongoing cost of ownership of the DITA system will be lower. In fact, even if initial DITA implementation were more expensive than non-DITA options the added long-term value would still justify the DITA-based solution. But DITA-based systems are demonstrably less expensive to acquire and implement than any possible non-DITA solution that satisfies the same set of requirements.

That is, by accepting a few constraints on your freedom to define arbitrary markup structures and by taking the effort to learn the basics of DITA, you gain huge leverage that enables implementing very sophisticated XML systems with a minimum cost of both startup and ownership compared to any other way you could satisfy the same requirements using a non-DITA solution.

Part of the point of these tutorials, and of *DITA for Practitioners* in general, is to make the necessary knowledge available so that the cost of learning what you need to learn is lowered as well.

So while there is a cost to the specialization feature in terms of design constraints and increased complexity for general-purpose, specialization-aware processors and some learning of new technical details and concepts for practitioners, the value returned making the investment of those costs is remarkable indeed. And the value will continue to increase as network effects serve to make more and more DITA-aware knowledge and processing available, which means it's available to all. That suggests that an investment in DITA-based systems is the safest XML system investment you can make today.

As a practitioner myself I would refuse to do a from-scratch XML implementation that was not DITA-based for the simple reason that it would be a disservice to the client to do anything else, because anything else would be more expensive, both in the short term and the long term, than a DITA-based solution. The only non-DITA-based

systems I work on any more are legacy systems that, for business reasons, cannot be (immediately) replaced with DITA-based equivalents. And it is painful for me to do so, because everything that DITA makes easy is hard in these systems.

Hopefully I've made my point, but in case I haven't, here's my position in a nutshell:

> DITA is a compelling technology not because it does cool stuff (although it does) but because through the specialization feature it *dramatically* lowers the initial and recurring costs of doing *anything* with XML for documents intended primarily for human consumption. Thus, even XML applications with the most basic requirements will benefit from a DITA-based solution simply because it will be cheaper, probably much cheaper, than any other XML-based alternative. And when you realize that you actually do need some of the really cool stuff DITA does, it's there waiting for you.

Introduction to the Configuration and Extension Tutorials

This section provides tutorial examples of creating the different types of DITA configuration and extension components, with a focus on the mechanics, not the concepts.

DITA defines a specific set of DTD and XSD coding requirements that all conforming vocabulary modules should follow. These requirements serve several important purposes:

- Consistency of implementation across modules
- Smooth interchange of knowledge
- Smooth interchange of vocabulary components
- Ease of implementation

Some the DITA-imposed requirements are required by the syntax and semantics of DTDs and XSDs, others are simply arbitrary decisions that had to be made and, having been made, make replication easy.

In short, once you learn the basic rules for how to organize the components of a vocabulary module, constraint module, or document type shell, you will be able to quickly understand the markup details of any conforming DITA module. You will also be able to quickly create new components because it is largely an exercise in copying, pasting, renaming, and deleting what you don't need.

What you don't have to do is worry about the details of how you'll structure and organize your DTD declarations or schema components. You won't have to work out clever schemes for modularity or conditionality.

You do need to have a basic working understanding of DTD or XSD syntax, depending on what technology you choose for your vocabulary modules. These tutorials tell you exactly what to type, so you can do them even if you don't have a working understanding of DTD or XSD syntax, but you will eventually need to know why you're doing what you're doing.

You should also, at some point, read the "Configuration, specialization, and constraints" section of the *DITA Architectural Specification*, which defines the implementation requirements reflected in these tutorials.

These tutorials are not intended to give you a deep conceptual understanding of how vocabulary modules work, they're intended to make it possible for you to create your own before you have a full understanding of why they work they way they do. One of the cool parts of DITA is that you can in fact do that.

If you follow these tutorials you should be able apply the processes demonstrated to your own configuration and extension requirements, assuming you've already worked out the markup design itself. A lot of specializations are either simple addition of new mention elements (specializations of `<keyword>` or `<term>`) or metadata elements. The easiest specialization to implement is an attribute domain that adds a new `props=` specialization.

General Guidance for Developing New Shells and Modules

While the creation of new document shells and modules is a largely mechanical process it is one that involves a lot of moving parts and fiddly bits. There are many opportunities for error and many of these errors can be difficult to track down because of all the pointing and indirection going on in the files involved.

To ensure success you must work carefully and methodically. The tutorials presented here reflect the methodical approach that I depend on.

In general, the methodical approach involves the following principles and practices:

- *Test*

  You must test from the very beginning. This is generally referred to as "test-driven development". The general practice is to create test cases first, verify that they fail (e.g., documents don't validate, transforms produce no output, etc.), then implement until the test cases pass. When they pass you know you're done.
- *Test at every step*

  You must ensure that you are in a known working state before making any change. If you do that, then you know that the last thing you did caused the breakage when something stops working. It means you only have to back out one change in order to get back to a good starting state.
- *Check your assumptions*

  Sometimes things aren't working because something doesn't work the way you thought it did. If you're getting an inexplicable failure, test your basic assumptions to ensure things work the way you think they should be working. For example, when debugging references to DTD components through catalogs, you can test your assumption that the catalog is correct by tracing down through a chain of references. The OxygenXML editor's "open file at cursor" feature makes this easy, as you can start with the root map and just chain down through the catalog-to-catalog and catalog-to-file references to make sure everything there is hooked up correctly (other editors have similar features). Likewise, you can use search and replace to verify that strings match between DOCTYPE declarations or schema location values and catalog entries.

  It's also good to verify you're changing the file you think you are. With the Toolkit there are often two or three copies of files: the copy you develop against in your source tree, the copy deployed to the Toolkit instance, and, for "template" files, the copy generated by the Toolkit's integration process. It's easy to accidentally open the wrong copy and then wonder what happened to your changes, either because you forgot to deploy them or because you modified the copy in the Toolkit and then redeployed over your changes from your source tree.

  If you have multiple Toolkits installed you should verify that you're running the code you think you are, since it's easy to run against the wrong Toolkit.
- *Start simple and work up*

  Implement in small increments. For example, start with all the new files for a related set of shells and vocabulary modules in one directory so you don't have to worry about setting up catalogs initially. Once everything works in that context, then reorganize the files to reflect the desired organization structure, creating and testing the necessary catalogs.

  Likewise, if you are creating several new document type shells, implement one completely before implementing the others, to ensure that you're not copying any mistakes.
- *Watch for cut-and-paste errors*

  A lot of the work in creating new document shells and modules is cutting and pasting from existing files to create new ones. It's part of what makes it so fast to create new modules. But it also has the potential for insidious cut-and-paste errors because you copy something you shouldn't have or inadvertantly copy the same mistake multiple times.
- *Use code control*

  Use a code control system like Subversion or VCC and commit your code frequently. You do not want to let uncommitted changes sit too long because it risks data loss and time loss. By testing early and often you know that you can commit code that isn't broken, even if it's not complete. If things subsequently go totally wrong you can simply restore from your last commit and start over. As a general rule you never want to be at

risk of losing more than an hour or two's worth of work, certainly not more than a day's work. Even if you are simply supporting yourself as a lone author you should use code control to manage your code and your authoring work. There are low-cost and free Subversion services or you can just set up a respository on your work machine (just make sure you back up the repository itself regularly).

DTD or XSD?

DITA vocabulary modules can be implemented using DTDs or XML schema documents (XSD). Which should you use?

At the time of writing, most DITA users use DTD-based vocabulary modules. However, you can use XSD-based modules if you want to or must in order to accommodate the tools you're using or the demands of a particular user community. For example, the Syntext Serna editor only uses XSDs to drive in-editor tag awareness (it can also use DTDs for validation, but not for in-editor tag awareness).

Which raises the question: should I use DTDs, XSDs, or both?

The short answer is "use DTDs unless you absolutely have to use XSDs" for now. This answer will hopefully change in the future.

The reason for this is simple: DITA's XSDs currently depend on the redefine feature of XSD. Unfortunately, the definition of redefine in the XSD 1.0 specifications is ambiguous to the point that different conforming XSD processors will produce different results for the same set of XSD documents. For this reason, the redefine feature is deprecated in XSD 1.1 (under development at the time of writing). Unfortunately, as currently formulated, DITA's XSDs depend on one particular interpretation of redefine, the one implemented by the Xerces 2.x parser.

This means that some conforming XSD processors will consider DITA XSDs invalid and will be unable to process them. This means in turn that you cannot reliably interchange XSD-based vocabulary modules and document type shells except to the degree that all the interchange partners are using compatible XSD processors.

However, because the Xerces parser does the right thing and because so many tools use the Xerces parser (or can use it), including the Open Toolkit, and because all the major commercial DITA-aware editors support DITA XSDs, you can create environments where XSD-based DITA documents can be processed reliably. But you cannot expect that all schema-aware XML processors that are not specifically DITA-aware will be able to process XSD-based DITA documents.

I find this frustrating because except for the redefine issue, I prefer XSD over DTDs for the following main reasons:

1. XSDs are fully namespace aware and namespaces are good.
2. XSDs are more expressive than DTDs.
3. As XML documents, you can embed complete XML-based documentation in your XSD in a way you cannot in DTDs.
4. As XML documents, XSDs are more convenient to process than DTDs (however, there are available DTD parsers that make processing DTDs almost as convenient as XSDs)

DITA 1.x can't use namespaces so that obviates the namespace advantage of XSDs (DITA's modularity features are the functional equivalent of how one can use namespace-based XSDs to create true document type modules, they just don't use namespaces to do it).

The XSD 1.1 spec, currently at working draft stage as of December 2009, defines a new feature, "override", that is intended to functionally replace the redefine feature and provide a more flexible extension mechanism, one that is a better match to what DITA needs. If the override feature works as we want it to (the DITA TC has provided input to the XSD Working Group on the override feature on DITA's specific requirements) and it is implemented by most or all XSD-aware processors, then it will be possible to rework the DITA XSDs to use override rather than redefine, at which point XSDs can become the obvious better choice for vocabulary module implementation.

Until that time, however, the easiest and safest route is to stick with DTD-based shells and vocabulary modules.

Schema-Aware Parsing and Saxon

If you are using XSD for your DITA documents you will very likely want to process them outside of the Toolkit or an IDE like OxygenXML. Doing this requires a little bit of one-time setup.

At the time of writing, the Saxon XSLT engine is packaged in three versions: Home Edition, Professional Edition, and Enterprise Edition. Of these three packages, only Enterprise Edition provides schema-aware XSLT processing directly.

However, because Saxon can use any JAXP parser, you can configure it to use a schema-aware processor. This in turn is simply a matter of turning on a couple of options on the Apache Xerces parser.

You do this by creating a simple Java class that wraps the Xerces parser in order to set its configuration and then use that class as the parser class used by Saxon, which you can specify on the command line.

This configuration is done for you automatically by the Open Toolkit but if you want to run Saxon outside of the Toolkit you may need to set this up.

The Java class looks like this:

```
package org.example.xerces;

import org.apache.xml.resolver.CatalogManager;
import org.apache.xml.resolver.tools.ResolvingXMLReader;
import org.xml.sax.SAXNotRecognizedException;

/**
 *
 */
public class SchemaValidatingCatalogResolvingXMLReader extends
        ResolvingXMLReader {

    /**
     * @throws Throwable
     * @throws SAXNotRecognizedException
     *
     */
    public SchemaValidatingCatalogResolvingXMLReader() throws
SAXNotRecognizedException, Throwable {
        super();
        init();
    }

    /**
     * @throws Throwable
     * @throws SAXNotRecognizedException
     *
     */
    private void init() throws SAXNotRecognizedException, Throwable {
        // System.err.println(" + INFO: Using " +
this.getClass().getName());
        this.setFeature("http://xml.org/sax/features/validation", false);
        this.setFeature("http://apache.org/xml/features/validation/
schema", true);
        this.setFeature("http://apache.org/xml/features/validation/
dynamic", true);
    }

    /**
     * @param catalogManager
     * @throws Throwable
     * @throws SAXNotRecognizedException
     */
    public SchemaValidatingCatalogResolvingXMLReader(CatalogManager
```

```
catalogManager) throws SAXNotRecognizedException, Throwable {
        super(catalogManager);
        init();
    }

}
```

This class as shown depends on the Apache resolver.jar library, which does the catalog resolution you need.

To use it you would normally package this class into a jar, e.g., "mySchemaParser.jar", include it in the Java class path you use to run Saxon, and specify the class name on the "-x" parameter to Saxon, e.g.:

```
set CLASSPATH=%CLASSPATH%;%PROJECT_PATH%\java\lib\resolver.jar;
set CLASSPATH=%CLASSPATH%;%PROJECT_PATH%\java\lib\saxon.jar;
set CLASSPATH=%CLASSPATH%;%PROJECT_PATH%\java\lib\mySchemaParser.jar;

java -cp %CLASSPATH% net.sf.saxon.Transform -x
org.example.xerces.SchemaValidatingCatalogResolvingXMLReader $1 $2 $3
```

Packaging Document Type Shells and Vocabulary Modules as Toolkit Plugins

You can simplify deployment and testing of document type shells and vocabulary modules by packaging them as Open Toolkit plugins.

The DITA Open Toolkit provides a general plugin facility that makes it easy to integrate local shells and new vocabulary modules into the Toolkit's master entity resolution catalog. This makes the shells and modules immediately available to all processors that use the Toolkit's catalog, including, of course, the Toolkit itself.

If your DITA-aware editor uses the Toolkit's catalog to resolve DTD and XSD references then by deploying your modules to the Toolkit your editor uses, you should be able to immediately start validating and editing with your shells and modules. For example, the OxygenXML editor is configured by default to use the master catalog of the Open Toolkit provided with the Oxygen editor. By deploying your shells and modules as Toolkit plugins Oxygen becomes immediately able to use them with no additional configuration required. This makes it very quick to develop and test new shells and vocabulary modules.

Entity resolution catalogs are an OASIS standard and are supported by most XML-aware tools. A catalog provides a mapping from public identifiers, system identifiers (such as URNs) and URIs, to files on a local system.

The Open Toolkit's plugin mechanism works through pre-defined extension points that plugins can plug into [see general section on creating Toolkit plugins]. One extension point is in the master entity resolution catalog, catalog-dita_template.xml.

Neither the Toolkit nor the DITA standard say how you have to organize your document type shells and modules. The practice I use is to package all the document type shells and modules for a given project (or that otherwise would be expected to work together or that are developed and deployed as a unit) into a single plugin, named "*unique-package-prefix*.doctypes", where *unique-package-prefix* is a Java-style reverse Internet domain name, e.g. "com.planetsizedbrains", resulting in a plugin named "com.planetsizedbrains.doctypes". The point of the Java-style name is to ensure that the plugin's name will be unique in any Toolkit instance it's deployed to.

As deployed to the Toolkit a plugin is just a directory containing the files that make up the plugin.

My normal practice is to have within the plugin a directory named `doctypes` that then contains one subdirectory for each distinct vocabulary module or document type shell. For example, if I have a shell for each of the base topic types plus a new attribute domain module, the directory structure in my plugin would be:

```
com.planetsizedbrains.doctypes/
  doctypes/
    concept/
    phase-of-moon-AttDomain/
    reference/
    task/
    topic
```

At the root of the plugin are two files:

- `plugin.xml`, which defines the plugin to the Toolkit and controls how it is integrated with the appropriate extension point.
- `catalog.xml`, which is the file that will be integrated with the extension point in the main `catalog-dita_template.xml` file.

The `doctypes/` directory under the main plugin directory contains a master catalog file that then includes the catalogs from each module-specific directory. This organization provides a general-purpose root directory for your doctypes regardless of how they might be packaged for different tools.

Within each subdirectory are the working DTD or XSD files and a `catalog.xml` file providing the appropriate catalog entries for the files in that subdirectory. If I expect to have both DTD and XSD versions of my shells or modules, I create another level of subdirectory, one for DTDs and one for XSDs, like so:

```
com.planetsizedbrains.doctypes/
  doctypes/
    concept/
      dtd/
      xsd/
    phase-of-moon-AttDomain/
      dtd/
      xsd/
    reference/
      dtd/
      xsd/
    task/
      dtd/
      xsd/
    topic
      dtd/
      xsd/
```

With this organization, the `catalog.xml` files that provide the actual mapping from public IDs or schema location urns to files go in the `dtd` or `xsd` directories. Each module's main directory just contains a `catalog.xml` file that simply includes the catalog files from each of the subdirectories. This approach keeps everything self contained at each level. If you add or remove a module from your plugin you simply update the top-level catalog file in the `doctypes/` directory to add or remove a reference to that module's top-level catalog file and everything just works.

For this example, the `catalog.xml` file under the `com.planetsizedbrains.doctypes` directory would look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
prefer="public">
  <!-- NOTE: The following catalog entries will be added to the Toolkit's
             top-level catalog-dita.xml without modification, so they
             need to work as they will be in that catalog, not as they
             exist within the plug-in's directory.
    -->

  <nextCatalog catalog="doctypes/catalog.xml"/>

</catalog>
```

The catalog file in the `com.planetsizedbrains.doctypes/doctypes/` directory looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
prefer="public">

  <nextCatalog catalog="topic/catalog.xml"/>
  <nextCatalog catalog="concept/catalog.xml"/>
  <nextCatalog catalog="reference/catalog.xml"/>
  <nextCatalog catalog="task/catalog.xml"/>
```

```
  <nextCatalog catalog="phase-of-moon-AttDomain/catalog.xml"/>

</catalog>
```

This is the catalog that will be included by the `<nextCatalog>` entry add to the main `catalog-dita.xml` file.

Within each of the module directories the catalog looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
prefer="public">

  <nextCatalog catalog="dtd/catalog.xml"/>
  <nextCatalog catalog="xsd/catalog.xml"/>

</catalog>
```

Within the `dtd` or `xsd` directory for a document type shell, the catalog would look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
  prefer="public">

  <public publicId="urn:pubid:planetsizedbrains.com:doctypes:dita:topic"
    uri="topic.dtd"
  />
  <system
systemId="urn:pubid:planetsizedbrains.com:doctypes:dita:topic.dtd"
    uri="topic.dtd"
  />
</catalog>
```

The general pattern here is that there is a catalog in each directory that points down into the next directory. Only the top-level catalog and leaf catalogs vary—the intermediate catalogs are always the same. This makes it easy to copy an existing module or shell's directory tree as the starting point for a new module or shell. This also makes it easier to reorganize the directories if necessary, since no single catalog points down more than one directory level.

The `plugin.xml` file goes in the top-level directory. For a document type plugin it looks like this:

```
<plugin id="com.planetsizedbrains.doctypes">
  <feature extension="dita.specialization.catalog.relative"
value="catalog.xml" type="file"/>
</plugin>
```

The bit in bold is the plugin identifier and is the only part you must change for your own module. The `<feature>` element is always the same for a doctype plugin. The plugin name must be unique across all plugins in your Toolkit, so the easiest and most reliable thing is to use the same Java-style name for the plugin ID as you used for the plugin's directory.

To deploy the plugin, simply copy the directory into the `plugins/` directory of your Toolkit and run the `integrator.xml` Ant script, e.g., from the root directory of the Toolkit:

```
ant -f integrator.xml
```

Your shells and modules should be ready to use. You can verify the integration by opening the Toolkit's `catalog-dita.xml` file and looking for a reference your plugin's top-level catalog file. If you are using an editor like OxygenXML that lets you follow file references (in Oxygen you put your cursor on a line that contains a reference to a file and press "ctrl+enter") you can use that feature to follow the chain of references from catalog to catalog to make sure you have everything hooked up correctly. But if you followed the file organization pattern shown here, it should be good.

[Need a reference to a general topic on troubleshooting entity resolution issues.]

Deploying Toolkit Plugins

All of the tutorials that follow ultimately result in components that can or must be packaged as Open Toolkit plugins.

A plugin for the Open Toolkit is simply a directory containing the files that make up the plugin.

To use a plugin with a Toolkit instance you "deploy" the plugin by copying the directory to the `plugins` directory and running the integrator.xml Ant script included with the Toolkit (see *Packaging Document Type Shells and Vocabulary Modules as Toolkit Plugins* on page 20).

This raises the question of how to actually do the copying, especially during development, where you will be making many changes and wanting to continuously re-deploy your plugin as you develop and test it.

As a general practice you do not want the source code for your plugin to be managed in the directory structure of the Toolkit itself. You want to manage the source in a separate work area and then copy it to the Toolkit as you go.

Another reason to keep your plugin source separate from the Toolkit is that the source file organization may not match how the files need to be organized for use in a plugin. For example, you may need to support one or more editors that expect a specific organizational structure that is different from what you would use in a plugin. Thus you may need to produce several different packagings of the same document type files for use by different tools.

There are, of course, many ways to deploy plugins: you can simply copy the files manually using Windows Explorer or Finder or whatever. You can use a batch or a shell script. Or you can use Ant.

The Open Toolkit is based on Apache Ant, which is a general build process scripting facility. Ant is used very heavily in Java projects to manage compiling and packaging Java code, but Ant is general purpose and can be used for lots of things.

One thing you can do easily with Ant is copy files from one place to another. Thus you can create an Ant script that will copy the source files for your plugins to the appropriate Toolkit and run the integrator.xml Ant script in one go.

The main challenge here is telling the Ant script where the Toolkit is on your local machine.

My practice is to have a file named `build.properties` (Windows) or `.build.properties` (Linux or OSX) in my home directory that defines a property named "dita-ot-dir" and sets it to the location of the Toolkit I want to deploy to. In my project-specific build scripts I then include the `build.properties` file and use the *dita-ot-dir* property in copy tasks that deploy my Toolkit plugins. Note that the script below defines the property "dita-ot-dir". This definition of the property will be used *only* if *dita-ot-dir* is not defined in eitehr `.build.properties` or `build.properties`. By the property definition precedence rules in Ant, the first definition of a property wins, so the definition in the main Ant script is ignored if *dita-ot-dir* is defined in either of the included properties files.

Here is a simple Ant build script (named `build.xml` by Ant convention) that will deploy a set of Toolkit plugins where the plugin source files are in a directory named `toolkit_plugins` under the main project directory:

```
<project name="DITA Tutorials" basedir="." default="deploy-toolkit-
plugins">

  <property file="build.properties"/>
  <property file="${user.home}/.build.properties"/>
  <property file="${user.home}/build.properties"/>


  <property name="dist" location="${basedir}/dist"/>
  <property name="plugin.dist" location="${dist}/plugins"/>
  <property name="dita-ot-dir" location="c:\DITA-OT1.5"/>
  <property name="plugin-src" location="${basedir}/toolkit_plugins"/>
  <property name="plugin-deploy_target" location="${dita-ot-dir}/plugins"/>
  <property name="ot-plugins-base-name" value="com.example.plugins"/>
```

```
  <import file="${dita-ot-dir}${file.separator}integrator.xml"
optional="yes"/>


  <target name="init">
  <tstamp/>
  </target>

  <tstamp>
    <format property="package.date" pattern="yyyy-MM-dd"/>
  </tstamp>

  <target name="clean">
  </target>


  <target name="dist-toolkit-plugins"
      description="Packages the DITA Open Toolkit plugins for deployment
to a working Toolkit instance"
      depends="dist-init"
  >
    <delete includeemptydirs="true" failonerror="false">
      <fileset dir="${plugin.dist}"/>
    </delete>
    <copy todir="${plugin.dist}">
        <fileset dir="${plugin-src}">
        </fileset>
    </copy>
  </target>

  <target name="deploy-toolkit-plugins" depends="dist-toolkit-plugins"
     description="Deploy plugins to local DITA Open Toolkit">
    <delete failonerror="true" includeemptydirs="true">
      <fileset dir="${plugin-deploy_target}" includes="${ot-plugins-base-
name}.*/**"/>
    </delete>
    <mkdir dir="${plugin-deploy_target}"/>
    <copy todir="${plugin-deploy_target}">
      <fileset dir="${plugin.dist}">
        <include name="**/*"/>
      </fileset>
    </copy>
    <!-- Itegrate the deployed plugins: -->
    <antcall target="integrate"/>

  </target>

  <target name="dist-init">
    <delete failonerror="false" includeemptydirs="true">
      <fileset dir="${dist}" includes="*/**"/>
    </delete>
    <mkdir dir="${dist}"/>
  </target>

</project>
```

If you're not familiar with Ant this may look a lot more complicated than it really is.

Each `<target>` element represents a separate callable part of the script. In this script the main target is named "deploy-toolkit-plugins". It's also set as the default target so that if you simply run the script without specifying a target it will run the deploy-toolkit-plugins target automatically. Likewise, the Ant command looks for a file

named `build.xml` by default, so from a command line, if you are in the directory containing the `build.xml` file, you can just type "ant" and it should work (assuming the "ant" command is on your path):

```
c:\workspace\myproject> ant
Buildfile: build.xml

dist-init:

dist-toolkit-plugins:

...

BUILD SUCCEEDED
```

You can see what targets are available in the script by using the -projecthelp parameter:

```
c:\workspace\myproject> ant -projecthelp
Buildfile: build.xml

Main targets:

 deploy-toolkit-plugins  Deploy plugins to local DITA Open Toolkit
 dist-toolkit-plugins    Packages the DITA Open Toolkit plugins for
deployment to a working Toolkit instance
Default target: deploy-toolkit-plugins
c:\workspace\myproject>
```

The targets listed are those with `description=` attributes.

This script breaks the deployment into two steps:

1. dist-toolkit-plugins copies all the files to a "distribution" directory
2. deploy-toolkit-plugins then copies the files from the distribution directory to the Toolkit `plugins` directory. It also runs the `integrator.xml` script that is part of the Toolkit itself.

This two-step process allows you the opportunity to pull files together from different source locations into a single set of files to be deployed. For example, my normal practice is to manage all my vocabulary modules and document type shells in a source directory called `doctypes` and all my Toolkit plugins in a directory called `toolkit_plugins`. In the `toolkit_plugins` directory I have directories for each distinct document type plugin with just the files that are Toolkit-specific (usually just the `plugin.xml` file). I then have my dist-toolkit-plugins Ant target merge the files from the `toolkit_plugins` directory with the files from the `doctypes` directories to create complete Toolkit plugins.

The Ant property *ot-plugins-base-name* holds the common directory name prefix for all of the plugins managed by this Ant script, which makes it easy to delete existing deployed plugins and otherwise copy only the files you want. You would set this property to match whatever you've used for your plugins (I recommend the Java-style reverse domain name convention, e.g. "com.example.myproject").

One you get this sort of pattern set up it becomes easy to replicate.

Once you have the Ant script working you can run it in a couple of different ways.

An easy way is to start a Toolkit command-line shell using the `startcmd.bat` or `startcmd.sh` scripts that are part of the full Toolkit installation. These scripts set up a command line environment with everything set up correctly so you can run the Toolkit Ant scripts.

For example, to run your Ant script to deploy your plugins, you might do something like this:

```
c:\> cd c:\DITA-OT
c:\DITA-OT> startcmd
{stuff happens here}
c:\DITA-OT> cd c:\workspace\myproject
c:\workspace\myproject> ant
Buildfile: build.xml

dist-init:
    [mkdir] Created dir: /workspace/myproject/dist
```

```
dist-toolkit-plugins:

 ...

integrate:
[integrate] Using XERCES.

BUILD SUCCESSFUL
Total time: 2 seconds
```

Once you've run `startcmd` you can come back to the command window and rerun the deploy script just by hitting the up arrow and enter.

If you use Eclipse as your development environment you can run the Ant script from within Eclipse once you do a one-time setup of Ant with the Open Toolkit's java libraries. See *Setting Up Your Development Environment* on page 5.

You can also set up a standalone batch or shell script that sets up the execution environment as `startcmd` does and then runs your Ant script. That may be the most convenient approach if you expect other people to run this process with minimal setup.

Note also that you can set Ant properties on the command line. So for example if you had two different Toolkit instances that you wanted to deploy to you could set the value of the *dita-ot-dir* Ant property on the command line like so:

```
ant -Ddita-ot-dir=/Applications/oxygen_12_beta/frameworks/dita/DITA-OT/
```

Public Identifiers

You may have noticed in my examples that the public identifiers I use are URNs, not SGML-style public identifiers as used for the standard DITA modules.

This is because public identifiers are nothing more than magic strings, so it absolutely doesn't matter what syntax you use as long as it's reasonably likely to be globally unique. The only XML-defined requirement is that the public ID consist of the characters allowed by the production "PubidChar" in the XML standard (essentially characters allowed in URIs).

I prefer URNs over SGML-style public IDs for two reasons:

- It's the 21st century and DITA and XML are Web standards. URIs (and thus URNs) are the Web way of giving globally-unique names to resources.
- For XSDs you have to use URIs, so why not be consistent in your global naming syntax?

The use of public identifiers is pretty standard practice in XML and in the DITA community especially. However, in XML, public identifiers are *completely pointless*.

In XML, you must always have a system identifier. Even if you have a public identifier, you must also have a system identifier. Which immediately raises the question of why have a public identifier at all?

Why indeed?

I used to argue exactly that: that public IDs were pointless, that there was no useful difference between having a public ID and using a URN as your system ID because neither can be resolved directly and thus both require some sort of mapping and entity resolution catalogs can map both public and system IDs with equal facility. This is all true.

In addition, in an environment where document type shells and modules will be deployed to many locations (many different Toolkit instances) it is absolutely necessary that all references to shells and module components be indirect and that everything be properly mapped. Thus having directly-resolvable system IDs would be counter productive—you want system IDs that *cannot* be resolved directly so that any mapping configuration bugs cause early and immediate failure in your development environment. (This is why I make a point of ensuring that the system IDs in all of my shell document types consist of just the filename of the target module, regardless of where it might be relative to the using module—this ensures it won't be resolvable and thus mask a catalog mapping bug.)

Yet, you will notice that in all the examples in this book I use public identifiers? Why?

The answer is simply that the use of public identifiers is so ingrained and, in some cases, required by tools even when it shouldn't be (especially tools with an SGML legacy), that it simply proved too quixotic to stick to my principle and not provide or use public IDs. So I use them even though they are totally pointless. But I use URNs partly to subtly make the point that they are pointless because its more obvious that a public ID that is a URN is functionally identical to a system ID that is a URN (because they both use the same syntax and both require mapping in order to be resolved).

Whatever you do **do not** use URLs for public identifiers. It runs the risk of systems trying to resolve them. Always use URNs or SGML-style public IDs.

And please remember that in DITA the public ID for a document type shell or module means nothing. The only thing that matters is the value of a document's `domains=` attribute. DTDs and XSDs are just a convenience for authoring and (weak) validation and nothing more.

Any DITA tool (or, for that matter, any XML system generally) that puts too much emphasis on public IDs, and especially on the public IDs of document types, is fundamentally broken because it reflects a misunderstanding of what DTDs do and don't represent and, in DITA especially, a misunderstanding of what constitutes a *DITA document type*.

Document Type Shell Tutorial

Goal: Define a document type shell that omits domains you don't want and integrates a third-party domain you do want.

Most, if not all, production uses of DITA require the creation of document type shells. Fortunately, it's easy to do.

Document type shells serve to *integrate* vocabulary and constraint modules into a working DTD or XSD document that can be used to validate documents that should conform to the document type.

Often the only thing you need to do to configure your DITA environment is create document type shells, at least initially. For example, it is likely your writers do not need all the domains defined by the DITA standard and included in the OASIS-provied document type shells. You may also want to allow the nesting of different topic types or similar configurations that do not, themselves, require new specialization or constraint modules.

In a production DITA environment (meaning one where you will be doing real work as opposed to simply evaluating DITA technology) you should *always* create local shells even if you have no immediate need to impose constraints, adjust domain usage, or do specialization.

The reason for this requirement to create local document type shells is that as soon as you *do* need to do any of these things (and you will, sooner rather than later), you *must* have local document type shells. If you have not created local shells in advance, then any existing documents will have to be modified to point to the new shells you must now create. That could be a very disruptive change depending on how your documents are managed and which tools you are using to manage and author them.

Much better to set up your local shells first, get the configuration and deployment details worked out, and then you don't have to worry about future requirements that will require changes to the shells because you can modify the shells themselves without the need to modify existing documents (because the DTD or XSD pointers in the documents don't need to change in that case).

The overall process for creating a document type shell is:

1. Find an existing shell that is close to what you want to end up with and copy it to a new location.
2. Add or remove references to constraint or vocabulary modules as needed.
3. Assign an appropriate public identifier or URN for the new shell.
4. Deploy the new shell so that it is available to authors and processing components

The details of public ID or URN assignment and deployment depend on what specific tools you are using. In the case of the Open Toolkit, the best thing to do is to package the shell as part of a Toolkit plugin that contains an entity resolution catalog entry for the shell (which serves to assign the public identifier) and integrates the catalog

into the Toolkit's master catalog (which serves to deploy the shell, making it available to anything that uses the Toolkit, including the Toolkit itself).

(There is an implicit message here, which is that tools that depend on the Toolkit tend to simplify the overall task of DITA system configuration and maintenance by limiting the number of system components that have to be created or updated to reflect a change to vocabulary components. This is one reason that OxygenXML, in particular, serves as an effective DITA development and authoring environment: it uses the Toolkit directly to access all DITA-related vocabulary components. This, coupled with Oxygen's built-in specialization awareness, means that you can deploy your local shells, constraint modules, and vocabulary modules to one place and all DITA editing and processing through Oxygen just works.)

DTD Topic Type Shell Creation Tutorial

This tutorial demonstrates the process of creating a DTD-syntax document type shell.

In this tutuorial you will create a document type shell for `<topic>` that removes the technical-documentation-specific domains and adds a reference to the XML markup domain (defined in *Element Domain Specialization Tutorial* on page 54).

The other tutorials also include instructions for creating shells that integrate the components created in those tutorials. This tutorial shows you the basic process for shell creation, showing both removing existing references and adding new references.

DTD Topic Type Shell Tutorial Step 1: Copy Existing Shell

Create a directory in your workspace to hold the new document type shell, e.g., "`workspace/myTopicShell`". In the `myTopicShell` directory create the directory `dtd`.

Find the file `topic.dtd` in the technical content area of the standard DITA DTD distribution (`dtd/technicalContent/dtd/topic.dtd` in the files packaged with the Open Toolkit) and copy it into the `workspace/myTopicShell/dtd` working directory as file `myTopic.dtd`.

Note: The DITA 1.2 DTD distribution includes two document type shells for `<topic>`: `topic.dtd`, in the technical documentation area, and `basetopic.dtd`, in the base area. The `topic.dtd` shell integrates all the different topic domains that are part of the standard DITA vocabulary. The `basetopic.dtd` only integrates the highlighting, indexing, and utility domains. For thus tutorial I'm having you use topic.dtd so that you get experience removing things.

In the `myTopicShell` directory, create a new XML document named `mytopic-test-dtd.xml` that uses `myTopic.dtd` as its document type with this content:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE topic SYSTEM "dtd/myTopic.dtd">
<topic id="topic-1">
  <title>Topic title</title>
  <body>
    <p>Topic paragraph</p>
  </body>
</topic>
```

This document serves as your test document to verify that you haven't made any syntax errors in the new shell.

Open this document in your XML editor (e.g., in OxygenXML) and validate it. It should be valid.

DTD Topic Type Shell Tutorial Step 2: Delete Unwanted Domain Modules

A domain module is integrated into a DTD shell by the following bits, in order of occurrence in the DTD file:

- A reference to the domain's .ent file
- References to the domain's extension parameter entities (one or more) in the domain extension parameter entities section of the shell.
- A reference to the domain's `domains=` attribute component text entity in the `&included-domains;` entity.
- A reference to the domain's .mod file.

Thus, deleting a domain module means removing all four of these bits for a given domain.

To remove the reference to the programming domain from `myTopic.dtd`, perform these steps:

1. Find the declaration and reference for the `%pr-d-dec;` parameter entity and delete it:

```
...

<!ENTITY % abbrev-d-dec
  PUBLIC "-//OASIS//ENTITIES DITA 1.2 Abbreviated Form Domain//EN"
         "abbreviateDomain.ent"
>%abbrev-d-dec;

<!ENTITY % pr-d-dec
  PUBLIC "-//OASIS//ENTITIES DITA 1.2 Programming Domain//EN"

"programmingDomain.ent"
>%pr-d-dec;

<!ENTITY % sw-d-dec
  PUBLIC "-//OASIS//ENTITIES DITA 1.2 Software Domain//EN"

"softwareDomain.ent"
>%sw-d-dec;

...
```

2. Find the comment "DOMAIN EXTENSIONS", which is the section that contains the domain extension parameter entities. Find all the parameter entity references that start with "%pr-d-" and delete them, along with the leading or trailing "|" character:

```
...

<!-- ============================================================= -->
<!--                    DOMAIN EXTENSIONS                          -->
<!-- ============================================================= -->
<!--                    One for each extended base element, with
                        the name of the domain(s) in which the
                        extension was declared                     -->

<!ENTITY % pre          "pre |
                         %pr-d-pre; |
                         %sw-d-pre; |
                         %ui-d-pre;
                        ">
<!ENTITY % keyword      "keyword |
                         %pr-d-keyword; |
                         %sw-d-keyword; |
                         %ui-d-keyword;
                        ">
<!ENTITY % ph           "ph |
                         %hi-d-ph; |
                         %pr-d-ph; |
                         %sw-d-ph; |
                         %ui-d-ph;
                        ">
<!ENTITY % term         "term |
                         %abbrev-d-term;
                        ">
<!ENTITY % fig          "fig |
                         %pr-d-fig; |
                         %ut-d-fig;
                        ">
<!ENTITY % dl           "dl |
                         %pr-d-dl;
                        ">
<!ENTITY % index-base   "index-base |
                         %indexing-d-index-base;
```

```
                          ">
<!ENTITY % note           "note |
                           %hazard-d-note;
                          ">

...
```

The result of these deletions should look like this:

```
...

<!-- ============================================================= -->
<!--                    DOMAIN EXTENSIONS                          -->
<!-- ============================================================= -->
<!--                    One for each extended base element, with
                        the name of the domain(s) in which the
                        extension was declared                     -->

<!ENTITY % pre            "pre |
                           %sw-d-pre; |
                           %ui-d-pre;
                          ">
<!ENTITY % keyword        "keyword |
                           %sw-d-keyword; |
                           %ui-d-keyword;
                          ">
<!ENTITY % ph             "ph |
                           %hi-d-ph; |
                           %sw-d-ph; |
                           %ui-d-ph;
                          ">
<!ENTITY % term           "term |
                           %abbrev-d-term;
                          ">
<!ENTITY % fig            "fig |
                           %ut-d-fig;
                          ">
<!ENTITY % dl             "dl
                          ">
<!ENTITY % index-base     "index-base |
                           %indexing-d-index-base;
                          ">
<!ENTITY % note           "note |
                           %hazard-d-note;
                          ">
```

3. Find the comment "DOMAINS ATTRIBUTE OVERRIDE", which is the section that contains the declaration for the `&included-domains;` entity. Delete the reference to the `&pr-d-att;` entity:

```
...

<!-- ============================================================= -->
<!--                    DOMAINS ATTRIBUTE OVERRIDE                 -->
<!-- ============================================================= -->
<!--                    Must be declared ahead of the DTDs, which
                        puts @domains first in order               -->

<!ENTITY included-domains
                            "&hi-d-att;
                             &ut-d-att;
                             &indexing-d-att;
                             &hazard-d-att;
                             &abbrev-d-att;
                             &pr-d-att;
                             &sw-d-att;
                             &ui-d-att;
```

```
  "
>

...
```

The result should be:

```
...

<!-- ============================================================= -->
<!--                    DOMAINS ATTRIBUTE OVERRIDE                 -->
<!-- ============================================================= -->
<!--                    Must be declared ahead of the DTDs, which
                        puts @domains first in order               -->

<!ENTITY included-domains
                          "&hi-d-att;
                           &ut-d-att;
                           &indexing-d-att;
                           &hazard-d-att;
                           &abbrev-d-att;
                           &sw-d-att;
                           &ui-d-att;
  "
>

...
```

4. Find the declaration and reference to the `%pr-d-def;` parameter entity and delete it:

```
...

<!ENTITY % ui-d-def
  PUBLIC "-//OASIS//ELEMENTS DITA 1.2 User Interface Domain//EN"
         "uiDomain.mod"
>%ui-d-def;

<!ENTITY % pr-d-def
  PUBLIC "-//OASIS//ELEMENTS DITA 1.2 Programming Domain//EN"
         "programmingDomain.mod"
>%pr-d-def;

<!ENTITY % sw-d-def
  PUBLIC "-//OASIS//ELEMENTS DITA 1.2 Software Domain//EN"
         "softwareDomain.mod"
>%sw-d-def;

...
```

5. Validate your test document. It should be valid if you've deleted everything correctly. Because the test document points directly to the `myTopic.dtd` file, any validation errors must be the result of something in the DTD file itself.

Repeat this process for each of the other domains you don't want.

Notice that the naming convention used for domain-related components makes it easy to find the bits for a given domain: just search on "*domainShortName*-d" where *domainShortName* is the short name for the domain, e.g., "pr", "ui", "sw", etc.

If you remove the programming (pr), software (sw), user interface (ui), and hazard statement (hazard) domains from this shell, the final result should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ============================================================

     My Topic Shell

     Sample topic type shell demonstrating removing some domains
```

```
     and adding others.


     ============================================================= -->

<!-- ============================================================= -->
<!--                    TOPIC ENTITY DECLARATIONS                  -->
<!-- ============================================================= -->

<!-- ============================================================= -->
<!--                    DOMAIN ENTITY DECLARATIONS                 -->
<!-- ============================================================= -->


<!ENTITY % hi-d-dec
  PUBLIC "-//OASIS//ENTITIES DITA 1.2 Highlight Domain//EN"
         "../../base/dtd/highlightDomain.ent"
>%hi-d-dec;

<!ENTITY % xml-d-dec
  PUBLIC "urn:pubid:example.org:doctypes:dita:modules:xml:entities"
         "xmlDomain.ent"
>%xml-d-dec;

<!ENTITY % ut-d-dec
  PUBLIC "-//OASIS//ENTITIES DITA 1.2 Utilities Domain//EN"
         "../../base/dtd/
utilitiesDomain.ent"
>%ut-d-dec;

<!ENTITY % indexing-d-dec
  PUBLIC "-//OASIS//ENTITIES DITA 1.2 Indexing Domain//EN"
         "../../base/dtd/
indexingDomain.ent"
>%indexing-d-dec;

<!ENTITY % abbrev-d-dec
  PUBLIC "-//OASIS//ENTITIES DITA 1.2 Abbreviated Form Domain//EN"
         "abbreviateDomain.ent"
>%abbrev-d-dec;

<!-- ============================================================= -->
<!--                    DOMAIN ATTRIBUTE DECLARATIONS              -->
<!-- ============================================================= -->


<!-- ============================================================= -->
<!--                    DOMAIN EXTENSIONS                          -->
<!-- ============================================================= -->
<!--                    One for each extended base element, with
                        the name of the domain(s) in which the
                        extension was declared                     -->

<!ENTITY % pre          "pre
                        ">
<!ENTITY % keyword      "keyword |
                         %xml-d-keyword;
                        ">
<!ENTITY % ph           "ph |
                         %hi-d-ph;
                        ">
<!ENTITY % term         "term |
                         %abbrev-d-term;
                        ">
```

```
<!ENTITY % fig          "fig |
                         %ut-d-fig;
                        ">
<!ENTITY % dl           "dl
                        ">
<!ENTITY % index-base   "index-base |
                         %indexing-d-index-base;
                        ">
<!ENTITY % note         "note
                        ">

<!-- ============================================================= -->
<!--                    DOMAIN ATTRIBUTE EXTENSIONS                -->
<!-- ============================================================= -->
<!ENTITY % props-attribute-extensions
  ""
>
<!ENTITY % base-attribute-extensions
  ""
>


<!-- ============================================================= -->
<!--                    TOPIC NESTING OVERRIDE                     -->
<!-- ============================================================= -->

<!--                    Redefine the infotype entity to exclude
                        other topic types and disallow nesting     -->
<!ENTITY % topic-info-types
  "topic
  "
>


<!-- ============================================================= -->
<!--                    DOMAINS ATTRIBUTE OVERRIDE                 -->
<!-- ============================================================= -->
<!--                    Must be declared ahead of the DTDs, which
                        puts @domains first in order               -->

<!ENTITY included-domains
                          "&hi-d-att;
                           &ut-d-att;
                           &indexing-d-att;
                           &abbrev-d-att;
                           &xml-d-att;
  "
>


<!-- ============================================================= -->
<!--                    TOPIC ELEMENT INTEGRATION                  -->
<!-- ============================================================= -->

<!--                    Embed topic to get generic elements        -->
<!ENTITY % topic-type
  PUBLIC "-//OASIS//ELEMENTS DITA 1.2 Topic//EN"
         "../../base/dtd/topic.mod"
>
%topic-type;


<!-- ============================================================= -->
<!--                    DOMAIN ELEMENT INTEGRATION                 -->
```

```
<!-- ============================================================= -->

<!ENTITY % hi-d-def
  PUBLIC "-//OASIS//ELEMENTS DITA 1.2 Highlight Domain//EN"
         "../../base/dtd/highlightDomain.mod"
>%hi-d-def;

<!ENTITY % xml-d-def
  PUBLIC "urn:pubid:example.org:doctypes:dita:modules:xml:declarations"
         "xmlDomain.mod"
>%xml-d-def;

<!ENTITY % ut-d-def
  PUBLIC "-//OASIS//ELEMENTS DITA 1.2 Utilities Domain//EN"
         "../../base/dtd/utilitiesDomain.mod"
>%ut-d-def;

<!ENTITY % indexing-d-def
  PUBLIC "-//OASIS//ELEMENTS DITA 1.2 Indexing Domain//EN"
         "../../base/dtd/indexingDomain.mod"
>%indexing-d-def;

<!ENTITY % abbrev-d-def
  PUBLIC "-//OASIS//ELEMENTS DITA 1.2 Abbreviated Form Domain//EN"
         "abbreviateDomain.mod"
>%abbrev-d-def;


<!-- ================== End My Topic DTD  ====================== -->
```

Note that in the domain extension area, there are now declarations like this:

```
<!ENTITY % pre           "pre
                         ">
```

Which is just defining `%pre;` to be "pre", which is effectively doing nothing to the original definition of `%pre;`. However, it doesn't hurt to have it here and it gives you a ready place to add in any new domains you might integrate later. You can remove this declaration entirely if you want.

DTD Topic Type Shell Tutorial Step 3: Add Reference to a New Domain

For this tutorial we will to integrate the XML domain defined in *Element Domain Specialization Tutorial* on page 54.

Note: You can get the worked result of that tutorial from the tutorial materials.

To integrate a new domain, we add the following four components:

- A declaration and reference to the domain's .ent file
- A reference to the domain's domain extension parameter entities, one for each different base element type the domain extends.
- A reference to the domain's `domains=` attribute text entity in the `&included-domains;` text entity declaration.
- A declaration and reference to the domain's .mod file.

For a given domain to be integrated you need to know the following:

- The public IDs of the .ent and .mod files. This should be in the entity resolution catalog for the domain. If the domain is packaged using the pattern described here, then the catalog file should be in the same directory as the .ent and .mod files for the domain. If you are integrating a standard DITA domain, you can look in the DITA standard itself or in the `catalog_dita.xml` file that is in the Open Toolkit's top-level directory.

- The set of domain extension parameter entities the module defines. These should be defined in the .ent file for the domain. For example, the file `softwareDomain.ent` has these declarations:

```
<!ENTITY % sw-d-pre
  "msgblock
  "
>
<!ENTITY % sw-d-ph
  "filepath |
   msgph |
   systemoutput  |
   userinput
  "
>
<!ENTITY % sw-d-keyword
  "cmdname  |
   msgnum |
   varname
  "
>
```

  Reflecting the fact that the software domain specializes from `<pre>`, `<ph>`, and `<keyword>`.
- The name of the `domains=` attribute text entity, which should be named "*domainShortName-d-att*", e.g., "sw-d-att". This should also be declared in the .ent file for the domain. For the software domain it looks like this:

```
<!-- ============================================================= -->
<!--                    DOMAIN ENTITY DECLARATION                  -->
<!-- ============================================================= -->

<!ENTITY sw-d-att
  "(topic sw-d)"
>
```

We want to integrate the XML domain defined in the Element Domain Specialization tutorial. The information we need for the XML domain is:

.ent file public ID	"urn:pubid:example.org:doctypes:dita:modules:xml:entities"
.mod file public ID	"urn:pubid:example.org:doctypes:dita:modules:xml:declarations"
domain extension parameter entities	`%xml-d-keyword;`
domains attribute text entity	`&xml-d-att;`

Armed with this knowledge, the process of integrating the domain is as follows:

1. Find the declaration and reference for the `%hi-d-dec;` parameter entity. Copy it and paste a new copy into `myTopic.dtd` immediately after the original:

```
...

<!-- ============================================================= -->
<!--                    DOMAIN ENTITY DECLARATIONS                 -->
<!-- ============================================================= -->


<!ENTITY % hi-d-dec
  PUBLIC "-//OASIS//ENTITIES DITA 1.2 Highlight Domain//EN"
         "../../base/dtd/highlightDomain.ent"
>%hi-d-dec;

<!ENTITY % hi-d-dec
  PUBLIC "-//OASIS//ENTITIES DITA 1.2 Highlight Domain//EN"
         "../../base/dtd/highlightDomain.ent"
>%hi-d-dec;
```

```
...
```

2. In the copy, change "hi-d-dec" to "xml-d-dec" in both places where it occurs:

```
<!ENTITY % xml-d-dec
  PUBLIC "-//OASIS//ENTITIES DITA 1.2 Highlight Domain//EN"
         "../../base/dtd/highlightDomain.ent"
>%xml-d-dec;
```

3. Replace the public identifier string with the URN for the XML domain:

```
<!ENTITY % xml-d-dec
  PUBLIC "urn:pubid:example.org:doctypes:dita:modules:xml:entities"
         "../../base/dtd/highlightDomain.ent"
>%xml-d-dec;
```

4. Replace the system identifier with "xmlDomain.ent":

```
<!ENTITY % xml-d-dec
  PUBLIC "urn:pubid:example.org:doctypes:dita:modules:xml:entities"
         "xmlDomain.ent"
>%xml-d-dec;
```

5. Find the comment "DOMAIN EXTENSIONS" and add a reference to the `%xml-d-keyword;` parameter entity to the declaration of the `%keyword;` parameter entity:

```
...

<!-- ============================================================= -->
<!--                    DOMAIN EXTENSIONS                          -->
<!-- ============================================================= -->
<!--                    One for each extended base element, with
                        the name of the domain(s) in which the
                        extension was declared                     -->

<!ENTITY % pre          "pre
                        ">
<!ENTITY % keyword      "keyword |
                         %xml-d-keyword;
                        ">

...
```

Note the leading "|" character added after "keyword " in the original declaration. This parameter entity is used to build up an OR group of element type names, so you need to add the "|" (OR) connector between "keyword" and "%xml-d-keyword;".

Tip: Common error: If you forget the "|" character or accidentally leave one out you will get validation errors that may be cryptic or misleading, because they will often point to the declarations for elements somewhere deep in the DITA declaration sets, usually the first element type encountered that happens to use the parameter entity you just modified. When you get this type of error, the first thing to check is your domain extension parameter entities—that is likely the cause of the error, especially if you are integrating pre-existing domains that you know are otherwise valid.

6. Find the comment "DOMAINS ATTRIBUTE OVERRIDE" and then the declaration for the `&included-domains;` text entity. Add a reference to the `&xml-d-att;` text entity:

```
...

<!-- ============================================================= -->
<!--                    DOMAINS ATTRIBUTE OVERRIDE                 -->
<!-- ============================================================= -->
<!--                    Must be declared ahead of the DTDs, which
                        puts @domains first in order               -->

<!ENTITY included-domains
                          "&hi-d-att;
                           &ut-d-att;
                           &indexing-d-att;
```

```
                              &abbrev-d-att;
                              &xml-d-att;
  "
>

...
```

7. Find the declaration and reference for the `%hi-d-def;` parameter entity. Copy it and paste a new copy immediately after the original:

```
...

<!-- ============================================================= -->
<!--                    DOMAIN ELEMENT INTEGRATION                 -->
<!-- ============================================================= -->

<!ENTITY % hi-d-def
  PUBLIC "-//OASIS//ELEMENTS DITA 1.2 Highlight Domain//EN"
         "../../base/dtd/highlightDomain.mod"
>%hi-d-def;

<!ENTITY % hi-d-def
  PUBLIC "-//OASIS//ELEMENTS DITA 1.2 Highlight Domain//EN"
         "../../base/dtd/highlightDomain.mod"
>%hi-d-def;

...
```

8. Change "hi-d-def" to "xml-d-def" everywhere it occurs:

```
<!ENTITY % xml-d-def
  PUBLIC "-//OASIS//ELEMENTS DITA 1.2 Highlight Domain//EN"
         "../../base/dtd/highlightDomain.mod"
>%xml-d-def;
```

9. Change the public identifier to the public ID for `xmlDomain.mod`:

```
<!ENTITY % xml-d-def
  PUBLIC "urn:pubid:example.org:doctypes:dita:modules:xml:declarations"
         "../../base/dtd/highlightDomain.mod"
>%xml-d-def;
```

Tip: Common error: If you are copy the public ID or URN from a catalog file, it is easy to accidently copy the public ID for the .ent file instead of for the .mod file and visa versa. If you get mysterious errors, like duplicate declaration messages or "element type note declared" messages even though you know you included the module, check the public ID value to make sure you didn't mix up the .ent and .mod public IDs.

10. Change the system identifier to "xmlDomain.mod":

```
<!ENTITY % xml-d-def
  PUBLIC "urn:pubid:example.org:doctypes:dita:modules:xml:declarations"
         "xmlDomain.mod"
>%xml-d-def;
```

11. Validate your test document. Assuming that the XML domain module is already deployed (you can find it in the materials package for the tutorials), then your document should validate if you haven't made any syntax errors in the shell DTD.

    If the document validates, see if you can enter any of the element types from the domain, such as `<xmlelem>` or `<xmlatt>`, into a paragraph. You should be able to.

That's it, you're done. You've successfully created a new topic type shell that removes domains you don't want and adds a new domain you do want.

It should be clear from following this tutorial that creating document type shells is an entirely mechanical process and that once you've done it once or twice it becomes very easy and quick.

This tutorial did not show how to package your new shell as a Toolkit plugin. For that, see *Packaging Document Type Shells and Vocabulary Modules as Toolkit Plugins* on page 20.

This tutorial shows how to create a topic type shell. Creating map type shells is exactly the same.

XSD Topic Type Shell Tutorial

This tutorial demonstrates the process of creating an XSD-syntax document type shell.

XSD document type shells use two XSD features to create complete XSD-based DITA document types: `<xs:include>` and `<xs:redefine>`. Includes are used to include components that are not being modified through redefine. Redefine is used to include components for which one or more groups defined in the component are being redefined, either to extend base groups with domain-provided element types or to restrict groups through constraint modules.

XSD Topic Type Shell Tutorial Step 1: Copy Existing Shell

Create a directory in your workspace to hold the new document type shell, e.g., "`workspace/myTopicShell/xsd`".

Find the file `topic.xsd` in the technical content area of the standard DITA XSD distribution (`schema/technicalContent/xsd/topic.xsd` in the files packaged with the Open Toolkit) and copy it into the `workspace/myTopicShell/xsd` directory as file `myTopic.xsd`.

The DITA 1.2 schema distribution includes two document type shells for `<topic>`: `topic.xsd` in the technical documentation area, and `basetopic.xsd` in the base area. The `topic.xsd` shell integrates all the different topic domains that are part of the standard DITA vocabulary. The `basetopic.xsd` only integrates the highlighting, indexing, and utility domains.

In the `myTopicShell` directory, create a new XML document named `mytopic-test-xsd.xml` that uses `myTopic.xsd` as its document type with this content:

```
<?xml version='1.0' encoding='UTF-8'?>
<topic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="xsd/myTopic.xsd"
 id="topic-1">
  <title>Topic title</title>
  <body>
    <p>Topic paragraph</p>
  </body>
</topic>
```

This document serves as your test document to verify that you haven't made any syntax errors in the new shell.

Open this document in your XML editor (e.g., in OxygenXML) and validate it. It should be valid.

XSD Topic Type Shell Tutorial Step 2: Remove References to Unwanted Domain Modules

Within an XSD document type shell, a given domain module is integrated by the following bits:

- An include of the domain's XSD file (XSD domain modules consist of a single XSD file, rather than as two files as for DTD domain modules or topic and map XSD modules).
- References to the domain's domain extension groups within the redefine for the commonElementMod.xsd document (or the module that contains the declaration of the base type being extended).
- The domains attribute value component for the domain, added to the definition of the `domains=` attribute value in the XSD shell (unlike in DTDs, there is no way to parameterize the value of an attribute in XSD schemas).

To remove the use of the programming domain from the `myTopic.xsd` shell, do the following:

1. Find the `<xs:include>` element for the `programmingDomain.xsd` file and delete it or comment it out:

```
  ...

  <!--  ================ TOPIC DOMAINS ====================  -->
  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:uiDomain.xsd:
1.2"/>
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:softwareDomain.xsd:1.2"/>
```

```
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:highlightDomain.xsd:1.2"/>
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:programmingDomain.xsd:1.2"/>
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:utilitiesDomain.xsd:1.2"/>
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:indexingDomain.xsd:1.2"/>
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:hazardstatementDomain.xsd:
1.2"/>
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:abbreviateDomain.xsd:1.2" />

  ...
```

2. Find all the <xs:group> elements that refer to a group starting with "pr-d-" and delete them:

```
  ...

  <xs:redefine
schemaLocation="urn:oasis:names:tc:dita:xsd:commonElementGrp.xsd:1.2">
    <xs:group name="keyword">
      <xs:choice>
        <xs:group ref="keyword"/>
        <xs:group ref="pr-d-keyword" />
        <xs:group ref="ui-d-keyword" />
        <xs:group ref="sw-d-keyword" />
      </xs:choice>
    </xs:group>

    <xs:group name="ph">
      <xs:choice>
        <xs:group ref="ph"/>
        <xs:group ref="pr-d-ph" />
        <xs:group ref="ui-d-ph" />
        <xs:group ref="hi-d-ph" />
        <xs:group ref="sw-d-ph" />
      </xs:choice>
    </xs:group>

    <xs:group name="pre">
      <xs:choice>
        <xs:group ref="pre"/>
        <xs:group ref="pr-d-pre" />
        <xs:group ref="ui-d-pre" />
        <xs:group ref="sw-d-pre" />
      </xs:choice>
    </xs:group>


    <xs:group name="dl">
      <xs:choice>
        <xs:group ref="dl"/>
        <xs:group ref="pr-d-dl"/>
      </xs:choice >
    </xs:group >


    <xs:group name="fig">
      <xs:choice>
        <xs:group ref="fig"/>
        <xs:group ref="pr-d-fig"/>
        <xs:group ref="ut-d-fig" />
      </xs:choice>
```

```
    </xs:group >

    <xs:group name="index-base">
      <xs:choice>
        <xs:group ref="index-base"/>
        <xs:group ref="indexing-d-index-base"/>
      </xs:choice>
    </xs:group >

    <xs:group name="note">
      <xs:choice>
        <xs:group ref="note"/>
        <xs:group ref="hazard-d-note"/>
      </xs:choice>
    </xs:group >

    <xs:group name="term">
      <xs:choice>
        <xs:group ref="term"/>
        <xs:group ref="abbrev-d-term"/>
      </xs:choice>
    </xs:group >

  </xs:redefine>

  ...
```

3. Find the declaration for the "domains-att" attribute group and delete "(topic pr-d)" from the value:

```
  <xs:attributeGroup name="domains-att">
    <xs:attribute name="domains" type="xs:string"
      default="(topic ui-d) (topic hi-d) (topic sw-d)
      (topic pr-d) (topic ut-d) (topic indexing-d)
      (topic hazard-d) (topic abbrev-d)"/>
  </xs:attributeGroup>
```

4. Validate your test document. It should still be valid.

   Because the XSD uses normal XML markup it's harder to introduce syntax errors than when modifying DTDs so it's unlikely you would have an invalid shell after following this set of steps. The most likely error would be failing to remove a reference to a domain extension group, but an XSD-aware editor like OxygenXML will flag in the editor a reference to a group that isn't defined (in this case, because you would have already deleted the reference to the domain's XSD file, which defines domain extension groups).

Repeat this process for the software, user interface, and hazard domains.

The resulting myTopic.xsd file should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ============================================================
     My Topic Shell

     Shell that demonstrates adding and removing domains.
     ============================================================ -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:ditaarch="http://dita.oasis-open.org/architecture/2005/">

  <!-- Add the domains to the base topic XML Schema -->
  <!--  ================ TOPIC DOMAINS ====================  -->
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:highlightDomain.xsd:1.2"/>
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:utilitiesDomain.xsd:1.2"/>
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:indexingDomain.xsd:1.2"/>
  <xs:include
```

```
schemaLocation="urn:oasis:names:tc:dita:xsd:abbreviateDomain.xsd:1.2" />


  <!--  ================ GROUP DEFINITIONS ====================  -->
  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:metaDeclGrp.xsd:
1.2"/>
  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:tblDeclGrp.xsd:
1.2"/>
  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:topicGrp.xsd:
1.2"/>

  <!-- =================  MODULE INLCUDE DEFINITION  ==================  --
>
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:commonElementMod.xsd:1.2"/>
  <!-- ======== Table elements ======== -->
  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:tblDeclMod.xsd:
1.2"/>
  <!-- ======= MetaData elements, plus keyword and indexterm ======= -->
  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:metaDeclMod.xsd:
1.2"/>

  <xs:redefine
schemaLocation="urn:oasis:names:tc:dita:xsd:commonElementGrp.xsd:1.2">
    <xs:group name="keyword">
      <xs:choice>
        <xs:group ref="keyword"/>
      </xs:choice>
    </xs:group>

    <xs:group name="ph">
      <xs:choice>
        <xs:group ref="ph"/>
        <xs:group ref="hi-d-ph" />
      </xs:choice>
    </xs:group>

    <xs:group name="pre">
      <xs:choice>
        <xs:group ref="pre"/>
      </xs:choice>
    </xs:group>


    <xs:group name="dl">
      <xs:choice>
        <xs:group ref="dl"/>
      </xs:choice >
    </xs:group >


    <xs:group name="fig">
      <xs:choice>
        <xs:group ref="fig"/>
        <xs:group ref="ut-d-fig" />
      </xs:choice>
    </xs:group >

    <xs:group name="index-base">
      <xs:choice>
        <xs:group ref="index-base"/>
        <xs:group ref="indexing-d-index-base"/>
      </xs:choice>
    </xs:group >
```

```
    <xs:group name="note">
      <xs:choice>
        <xs:group ref="note"/>
      </xs:choice>
    </xs:group >

    <xs:group name="term">
      <xs:choice>
        <xs:group ref="term"/>
        <xs:group ref="abbrev-d-term"/>
      </xs:choice>
    </xs:group >

    </xs:redefine>


  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:topicMod.xsd:
1.2"/>


  <xs:group name="info-types">
    <xs:sequence/>
  </xs:group>

  <xs:attributeGroup name="domains-att">
    <xs:attribute name="domains" type="xs:string"
      default="(topic abbrev-d)
      (topic hi-d)
      (topic indexing-d)
      (topic ut-d)
      "/>
  </xs:attributeGroup>

</xs:schema>
```

XSD Topic Type Shell Tutorial Step 3: Add Reference to a New Domain

To integrate a new domain, we add the following three components:

- An `<xs:include>` of the domain's XSD module file (XSD domain modules consist of a single XSD file).
- A reference to the domain's domain extension groups, one for each different base element type the domain extends.
- The domain's `domains=` attribute contribution in the "domains-att" attribute group declaration.

For a given domain to be integrated you need to know the following:

- The URN or URL of the module's XSD file. This should be in the entity resolution catalog for the domain. If the domain is packaged using the pattern described here, then the catalog file should be in the same directory as the .xsd file for the domain. If you are integrating a standard DITA domain, you can look in the DITA standard itself or in the `catalog_dita.xml` file that is in the Open Toolkit's top-level directory.
- The set of domain extension groups the module defines. These are defined in the domain's XSD document. For example, the file `softwareDomain.xsd` has these declarations:

```
  <xs:group name="sw-d-ph">
    <xs:choice>
      <xs:element ref="msgph" />
      <xs:element ref="filepath" />
      <xs:element ref="userinput" />
      <xs:element ref="systemoutput" />
    </xs:choice>
  </xs:group>

  <xs:group name="sw-d-keyword">
    <xs:choice>
```

```
      <xs:element ref="msgnum" />
      <xs:element ref="varname" />
      <xs:element ref="cmdname" />
    </xs:choice>
  </xs:group>

  <xs:group name="sw-d-pre">
    <xs:choice>
      <xs:element ref="msgblock" />
    </xs:choice>
  </xs:group>
```

Reflecting the fact that the software domain specializes from `<pre>`, `<ph>`, and `<keyword>`.

- The `domains=` attribute contribution is "(topic *domainShortName*-d)" for a domain that specializes directly from topic and not from another domain. The `domains=` attribute value for the domain should also be documented in an XSD annotation at the start of the domain's XSD file, as in this example from `softwareDomain.xsd`:

```
  <xs:annotation>
    <xs:appinfo>
      <dita:domainsModule
         xmlns:dita="http://dita.oasis-open.org/architecture/2005/"
         >(topic sw-d)</dita:domainsModule>
    </xs:appinfo>
    <xs:documentation>

    </xs:documentation>
  </xs:annotation>
```

For this tutorial we want to integrate the XML domain defined in the topic element domain tutorial. The information we need for the XML domain is:

.xsd file URN:	"urn:pubid:exmple.org:doctypes:dita:modules:xmlDomain"
domain extension groups	"xml-d-keyword"
domains attribute contribution:	"(topic xml-d)"

Armed with this knowledge, the process of integrating the domain is as follows:

1. Find the `<xs:include>` element for the highlight domain. Copy it and paste a new copy into `myTopic.xsd` immediately after the original:

```
  ...

  <!--  ================ TOPIC DOMAINS ====================  -->
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:highlightDomain.xsd:1.2"/>
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:highlightDomain.xsd:1.2"/>

  ...
```

2. Replace the `schemaLocation=` value with the URN for the XML domain:

```
  ...

  <!--  ================ TOPIC DOMAINS ====================  -->
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:highlightDomain.xsd:1.2"/>
  <xs:include
schemaLocation="urn:pubid:exmple.org:doctypes:dita:modules:xmlDomain"/>

  ...
```

3. Find the `<xs:redefine>` for commonElementGrp.xsd and then the group named "keyword". Add a reference to the "xml-d-keyword" group:

```
...
```

```
  <xs:redefine
schemaLocation="urn:oasis:names:tc:dita:xsd:commonElementGrp.xsd:1.2">
    <xs:group name="keyword">
      <xs:choice>
        <xs:group ref="keyword"/>
        <xs:group ref="xml-d-keyword"/>
      </xs:choice>
    </xs:group>
...
```

4. Find the group named "domains-att" and add the string "(topic xml-d) " to the value of the `default=` attribute:

```
...

  <xs:attributeGroup name="domains-att">
    <xs:attribute name="domains" type="xs:string"
      default="(topic abbrev-d)
      (topic hi-d)
      (topic indexing-d)
      (topic ut-d)
      (topic xml-d)
      "/>
  </xs:attributeGroup>

</xs:schema>
```

5. Validate your test document. Assuming that the XML domain module is already deployed (you can find it in the materials package for the tutorials), then your document should validate if you haven't made any syntax errors in the shell XSD. As for deleting domains, it is pretty hard to make a syntax error, especially if you are editing the XSD document in an XML editor like OxygenXML.

   If the document validates, see if you can enter any of the element types from the domain into a paragraph, such as `<xmlelem>` or `<xmlatt>`. You should be able to.

That's it, you're done. You've successfully created a new topic type shell that removes domains you don't want and adds a new domain you do want.

It should be clear from following this tutorial that creating document type shells is an entirely mechanical process and that once you've done it once or twice it becomes very easy and quick.

This tutorial did not show how to package your new shell as a Toolkit plugin. For that, see *Packaging Document Type Shells and Vocabulary Modules as Toolkit Plugins* on page 20.

This tutorial shows how to create a topic type shell. Creating map type shells is exactly the same.

Topic Constraint Module Tutorial

Goal: Define a topic constraint module that limits the content of <p> to just character data and a few inline element types.

Constraint modules allow you to adjust the content models and attribute lists of individual elements in any vocabulary module used by a given document type. The primary rule is that your changes must be at least as constrained as the base content model or attribute list, meaning that your constraints cannot allow anything that is not allowed by the base element type or attribute.

This means you can do any of the following:

- Remove optional elements
- Change repeating groups into sequences or (with XSD schemas) limit the number of repeats.
- Remove optional attributes
- Limit the set of values a keyword-valued element may take (as long as the list allows any required values).

In general the base DITA content models are very loose in order to allow maximum flexibility in specialization. This leaves you a lot of room for constraint.

In this tutorial the constraint module limits the content of <p> elements within topics to a small set of inline elements: <b> (bold), <i> (italic), and <u> (Underline).[3]

For both DTD and XSD the task is the same: find the appropriate parameter entity (DTD) or group (XSD) that defines the base content model for <p> and redefine it to reflect the constrained content model.

The constraint module is then integrated into the document type shells that need to impose these constraints.

Topic Constraint Module Step 1: Create The Constraint Module File

A DTD-syntax constraint module consists of a single file that contains the overrides for the base elements or attributes.

Constraint modules should be named "*qualifiertagname*Constraints.mod" where *qualifier* is descriptive of the constraints applied and *tagname* is the topic type, map type, or domain to which the constraint is applied.

Note that a given constraint module can only define constraints for a single topic type, map type, or domain. This ensures that constraint modules map one-to-one with the vocabulary modules they constrain. Likewise, for a given element type or attribute you can have at most one constraint module.

For this tutorial the constraint domain should be called "`highlightOnlyTopicConstraint.mod`".

Create a directory named "highlightOnlyTopicConstraint" and in that directory create the file `highlightOnlyTopicConstraint.mod`.

Edit the file `highlightOnlyTopicConstraint.mod` and add a descriptive header comment:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ====================================
     Constraint Domain: Paragraphs with
     only highlight domain elements.

     Copyright (c) 2010 Your Name Here

     ==================================== -->

<!-- ============ End of constraint module -->
```

Topic Constraint Module Step 2: Declare The `domains=` Attribute Text Entity

Like vocabulary modules, constraint modules must be declared in the `domains=` attribute of each top-level map or topic file that uses the constraint.

Constraint modules are indicated by names of the form "*moduleName*-c", e.g., "highlightOnlyTopic-c". As for vocabulary modules, you declare a text entity that holds the `domains=` attribute contribution. This entity is named "*tagname*-constraints", reflecting the fact that there can be at most one constraint module for a given element type in a given document type shell.

For the highlight-only constraint module, add the declaration shown to the highlightOnlyTopicConstraint.mod file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ====================================
     Constraint Domain: Paragraphs with
     only highlight domain elements.

     Copyright (c) 2010 Your Name Here

     ==================================== -->

<!ENTITY topic-constraints "(topic highlightOnlyTopic-c)" >
```

[3] This example was suggested by Casey Jordan who asked on the DITA Users' list about how to implement this type of constraint using XSD schema.

```
<!-- ============ End of constraint module -->
```

What do you do if you have multiple constraint modules for a given element type that you want to use together?

In that case you must create a new constraint module that combines the multiple constraint modules together. For example, if someone gave you another topic constraint module that constrained a different element type, you would combine those declarations with the declarations in the highlight-only topic constraint module.

When the two constraints constrain different element types then combining them is easy: just copy all the declarations from one constraint module into the other and save the result as a new file with a new module name.

If the constraints constrain the same element types then you must work out how the constraints should be combined in order to create a new constraint module.

Topic Constraint Module Step 3: Define the New Content Models

The base content model for the <p> element is defined in a parameter entity named `%p.content;` (by the general DITA 1.2 rules for constructing DTD-syntax element type declarations). Thus we need to declare our own version of `%p.content;` with the content model we want.

We want to allow only #PCDATA and the element types <b>, <i>, and <u>, or any specializations of those types that might be integrated with a given document type that uses this constraint.

Thus, the content model should be defined as:

```
(#PCDATA |
 %b; |
 %i; |
 %u;)*
```

To allow specializations of these element types we need to reference the tagname parameter entities for them, not the bare element type names. That is, we want to use "`%b;`" not just "`b`" in the content models. Because of the way parameter entity declaration precedence works in DTDs we have to declare the tagname parameter entities in the constraint module to ensure they are declared before they are used in the content model.

Given this knowledge, add the following parameter entity declarations to the constraint module file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ====================================
     Constraint Domain: Paragraphs with
     only highlight domain elements.

     Copyright (c) 2010 Your Name Here

     ==================================== -->

<!ENTITY topic-constraints "(topic highlightOnlyTopic-c)" >

<!ENTITY % b           "b"           >
<!ENTITY % i           "i"           >
<!ENTITY % u           "u"           >

<!ENTITY % p.content
  "
  (#PCDATA |
   %b; |
   %i; |
   %u;)*
  ">


<!-- ============ End of constraint module -->
```

That's the entire constraint module.

It may see odd that we have to declare tagname entities that are already declared in the base module. However, there is an order of inclusion problem that can't be worked around any other way.

In DTDs, the first declaration of a parameter entity name wins. This is why the module files are included *last* in document type shells: they have to give domain modules and constraint modules the opportunity to override any parameter entities declared in the base module.

Likewise, constraint modules are included *after* any domain entity files and after the domain integration parameter entities in the document type shell so that they will use any element-type-specific integrations in the constrained content models they define. But because the tagname parameter entities used in the constraint module may not be overridden by domains in a given shell, the constraint module has to declare the tagname parameter entities locally just to be sure.

Topic Constraint Module Step 4: Integrate the Constraint Module in a Document Type Shell

For testing purposes, copy the standard `topic.dtd` file to a convenient location, either in the same directory as the constraint module file or in a nearby directory. To make its purpose clear you can name it something like `topic-with-highlight-constraint.dtd` if you want.

Edit the copied topic DTD file, search for the comment "DOMAINS ATTRIBUTE OVERRIDE" and update the declaration of the `&included-domains;` text entity to include a reference to the `&topic-constraints;` text entity:

```
<!-- ============================================================= -->
<!--                    DOMAINS ATTRIBUTE OVERRIDE                 -->
<!-- ============================================================= -->
<!--                    Must be declared ahead of the DTDs, which
                        puts @domains first in order               -->

<!ENTITY included-domains
                          "&hi-d-att;
                           &ut-d-att;
                           &indexing-d-att;
                           &hazard-d-att;
                           &abbrev-d-att;
                           &pr-d-att;
                           &sw-d-att;
                           &ui-d-att;
                           &topic-constraints;
  "
>
```

This adds the declaration of the topic-element-type constraint module to the `domains=` attribute for the `<topic>` element as configured in this document type shell.

Now search for the comment "TOPIC ELEMENT INTEGRATION" and add this parameter entity declaration and reference:

```
...

<!-- ============================================================= -->
<!--                    CONTENT CONSTRAINT INTEGRATION             -->
<!-- ============================================================= -->

<!ENTITY % hiOnlyTopic-c-def
  SYSTEM "highlightOnlyTopicConstraint.mod">
%hiOnlyTopic-c-def;

<!-- ============================================================= -->
<!--                    TOPIC ELEMENT INTEGRATION                  -->
<!-- ============================================================= -->
```

```
...
```

The topic.dtd file you started with may or may not have the "CONTENT CONSTRAINT INTEGRATION" comment. If it's not there, add it.

Topic Constraint Module Step 5: Test the Document Type Shell

Create a new <topic> document in the same directory as the document type shell DTD with this content:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE topic SYSTEM "topic-with-highlight-constraint.dtd">
<topic id="topic-1">
  <title>Test of Paragraph Constraint</title>
  <body>
    <p>Allowed: <i>i</i>, <b>b</b>, and <u>u</u>. Not allowed:
<keyword>keyword</keyword>.</p>
  </body>
</topic>
```

Validate this document. The validator should report that <keyword> is not allowed within <p>, that only <b>, <i>, and <u> are allowed. In a DTD-aware editor you should also be able to inspect the content model to see that it only allows <b>, <i>, and <u>. Remove the <keyword> element and verify that the topic is then valid.

In an XML-aware editor you should be able to inspect the effective value of the domains= attribute to verify that the constraint module is properly declared. If you apply the Toolkit to the topic and keep the temp directory you can look at the intermediate topic file generated by the Toolkit to see what it produced for the domains= attribute.

If you have made any typing or copying errors in the DTD declarations the parser will tell you, although the errors are not always obvious from the messages provided by the parser.

You can now package the constraint as a Toolkit plugin and integrate it with your other shell document types, if you want. See *Packaging Document Type Shells and Vocabulary Modules as Toolkit Plugins* on page 20.

Topic Constraint Module: XSD-Syntax Version

XSD constraint modules consist of one or two XSD documents that redefine the content model or attribute list to be constrained. Whether the constraint requires one document two depends on the details of the constraint being applied and the group being constrained. This particular constraint requires an intermediate XSD file in order to work around limitations in the way the redefine feature works. Essentially, we have to clear the content of <p> and then extend it, which requires two levels of redefinition.

To create an XSD version of the highlight-only topic constraint module, follow these steps:

1. Create a new directory to hold the constraint module and an XSD document type shell that uses it. This directory will eventually be the directory for the constraint module, so name it in a way that reflects the name of the constraint module, e.g. "highlightOnlyPConstraints".
2. Copy topic.xsd from the technicalContent directory of the OASIS-provided DITA XSD distribution to this new directory. Name it something like "topic-with-highlight-constraint.xsd" to make it clear what this shell is for.

   Eventually you will move this shell schema to a different directory, but for now putting all the pieces in one directory makes setup and testing easier. Once everything is working in this location you can reorganize the files and know when you've got everything hooked up right again.
3. In the same directory, create a new XML document that uses the shell XSD. This document is used to test as you go. Name it something like "constraint-test.xsd". Verify that the document is valid. As you haven't yet modified the shell schema document, it should be valid. The document should look something like this:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <topic
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="topic-with-highlight-constraint.xsd"
     id="topicid">
   ```

```
  <title>Test of Simple Paragraph Restriction</title>
  <body>
    <p><b>b</b>, <i>i</i>, <u>u</u>. Not allowed <keyword>keyword</
keyword>.</p>
    <p></p>
  </body>
</topic>
```

4. In the same directory, create a new empty XSD document named "highlightOnlyPConstraints.xsd. This is the constraint module file.
5. For this domain you are constraining the content model of the <p> element, so the first task is to find where p's content model is defined in the XSD files that make up the base `<topic>` vocabulary module. One way to do this is to search for 'name="p.content"' across all the standard XSD modules (for example, using OxygenXML's "Find in files" feature or using Search in Windows Explorer"). You should find the group definition for "p.content" in `commonElementMod.xsd`. It looks like this:

```
    <xs:group name="p.content">
        <xs:sequence>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:group ref="para.cnt" minOccurs="0"/>
            </xs:choice>
        </xs:sequence>
    </xs:group>
```

6. Create the intermediate redefinition file `highlightOnlyPConstraintsInt.xsd` and give it this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:annotation>
    <xs:documentation>Intermediate level of redefine to work around XSD
redefine limitations
    </xs:documentation>
  </xs:annotation>

  <xs:redefine
schemaLocation="urn:oasis:names:tc:dita:xsd:commonElementMod.xsd:1.2">
    <!-- constrain content of <p> element -->
    <xs:group name="p.content">
      <xs:choice>
        <!-- "clear" p.content so we can then override it again in the
next level of redefine. -->
      </xs:choice>
    </xs:group>
  </xs:redefine>

  <xs:group name="p-highlight-only.content">
    <xs:choice>
      <xs:sequence maxOccurs="unbounded">
        <xs:group ref="b" minOccurs="0"/>
        <xs:group ref="i" minOccurs="0"/>
        <xs:group ref="u" minOccurs="0"/>
      </xs:sequence>
    </xs:choice>
  </xs:group>
</xs:schema>
```

This schema document redefines <p> as declared in `commonElementMod.xsd` to be an empty group. This document also defines a new group containing the new content model we want for <p>.

7. In the new constraint module XSD file, create an xs:redefine element that points `highlightOnlyPConstraintsInt.xsd` and defines "p.content" to use the "p-highlight-only.content" content group also defined in `highlightOnlyPConstraintsInt.xsd`:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:redefine schemaLocation="highlightOnlyPConstraintsInt.xsd">
    <!-- constrain content of <p> element -->
    <xs:group name="p.content">
      <xs:choice>
        <xs:group ref="p-highlight-only.content"/>
      </xs:choice>
    </xs:group>
  </xs:redefine>


</xs:schema>
```

8. To the topic document type shell XSD add the statement in bold between the two statements shown:

```
  ...

  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:topicMod.xsd:
1.2"/>

  <xs:include schemaLocation="highlightOnlyTopicConstraints.xsd" />

  <xs:group name="info-types">
    <xs:sequence/>
  </xs:group>

  ...
```

   This includes the constraint module in the XSD document type shell.

9. In the topic document type shell, remove or comment out the include of commonElementMod.xsd, since it is now included by the redefine statement in the intermediate XSD document:

```
  <!-- =================  MODULE INLCUDE DEFINITION  ==================
-->
 <!-- <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:commonElementMod.xsd:1.2"/>
-->
```

10. Test your work so far by validating your test document. It should validate (meaning all the schema components are hooked up). Make sure your test document has a paragraph with something other than `<b>`, `<i>`, or `<u>` in it (e.g., `<keyword>`).

11. Update the domains attribute declaration in the document type shell to reflect the use of the constraint module:

```
  ...

  <xs:attributeGroup name="domains-att">
    <xs:attribute name="domains" type="xs:string"
      default="(topic highlightOnlyP-c)
          (topic ui-d) (topic hi-d) (topic sw-d) (topic pr-d)
          (topic ut-d) (topic indexing-d) (topic hazard-d)
          (topic abbrev-d)"/>
  </xs:attributeGroup>

  ...
```

12. Test the constraint by validating your test document. It should be valid and should only allow `<b>`, `<i>`, and `<u>` within `<p>`.

At this point you could package the constraint module and the shell document type into Toolkit plugins.

Attribute Specialization Tutorial

Goal: Declare an attribute domain vocabulary module that provides a new conditional (property) attribute.

You can specialize from the `base=` and `props=` attributes. For conditional processing (filtering and flagging), this lets you add your own attributes rather than using `otherprops=`, which can be clearer to authors and implementors.

For this tutorial, the goal is to declare an attribute domain module that provides a new conditional attribute, in this case the "phase of the moon" condition, useful for medical information for werewolves and other lycanthropes.[4]

Attribute domains are the easiest form of specialization. It's so easy that you should never complain about missing conditional attributes in the base DITA specification, just declare your own.

The specialization requires two things:

1. For each specialization of `props=`, a .ent file that defines the attribute and a corresponding domain declaration. This is the "attribute domain vocabulary module".

2. Modification of any document type shells that need to reflect the specialized attribute (e.g., topic.dtd, reference.dtd, or your own specialized topic types' document type shells). You integrate the specialization attribute domain through the shell DTDs.

For this tutorial we want to create a specialization of `props=` called "phase-of-moon" that takes as its value one or more moon phase names (e.g., "full", "new", "waning", "waxing", etc.).

Note that an attribute domain always defines exactly one attribute. You cannot declare multiple attributes in a single attribute domain module.

Attribute Specialization Step 1: Create Domain Module Files

Step 1 is to create the attribute domain declaration as follows.

First, create a file named "`phase-of-moonAttDomain.ent`".

In "`moonPhasePropsDomain.ent`", create these two declarations:

```
<!ENTITY % phase-of-moon-d-attribute
  "phase-of-moon
    CDATA
    #IMPLIED
  "
>
<!ENTITY phase-of-moon-d-att
   "a(props  phase-of-moon)"
>
```

The first declaration declares the `phase-of-moon=` attribute and puts it in a parameter entity so we can add it to the DITA-defined `%selection-atts;` parameter entity via the `%props-attribute-extensions;` configuration parameter entity in document type shells.

The second declaration is the domain declaration string for the attribute domain. It will be added to the value of the `domains=` attribute declared for each topic or map element type. In the value, the "props" keyword indicates that the attribute is a `props=` attribute specialization and not a `base=` attribute specialization.

You should of course add an appropriate descriptive header to the file as well as a little documentation for the attribute itself.

---

[4] For a more practically useful example, see the d4p_renditionTarget attribute domain in the DITA For Publishers vocabulary.

This is all that is required for the attribute domain module (there is no separate .mod file, as there is for element domains).

Attribute Specialization Step 2: Integrate With Document Type Shell

Step 2 is to integrate the vocabulary module into your local copy of each of your document type shells. The pattern is the same for each shell.

For this tutorial, use a copy of the concept.dtd shell, as for the element domain specialization tutorial.

Edit the copy of `concept.dtd` and find the comment that reads "DOMAIN ATTRIBUTE DECLARATIONS". Following that comment, add this declaration:

```
<!ENTITY % phase-of-moon-d-dec
  SYSTEM "phase-of-
moonAttDomain.ent"
>
%phase-of-moon-d-dec;
```

This pulls in the attribute domain module.

Find the comment that reads "DOMAIN ATTRIBUTE EXTENSIONS". Following that comment you should see a declaration for the `%props-attribute-extensions;` parameter entity. It will probably be declared as an empty string.

Modify the entity replacement text (the bit inside the double quotes) to include a reference to the `%phase-of-moon-d-attribute;` parameter entity:

```
<!ENTITY % props-attribute-extensions
  "%phase-of-moon-d-attribute;"
>
```

This adds the `phase-of-moon=` attribute to the `%selection-atts;` parameter entity which is then included in the `%univ-atts;` parameter entity, making this new attribute available on most elements (some elements, such as title, are not selection candidates).

Find the comment that reads "DOMAINS ATTRIBUTE OVERRIDE". Following that you should see the declaration of the text entity `&included-domains;` and it should include references to a number of "*x*-d" text entities.

To this entity add a reference to the `&phase-of-moon-d-att;` text entity:

```
<!ENTITY included-domains
   "&hi-d-att;
    &ut-d-att;
    &phase-of-moon-d-att;
   "
>
```

This formally declares your `props=` attribute specialization so that DITA processors will know that `phase-of-moon=` is in fact a conditional attribute and that they should filter on it as appropriate.

Note: Note the difference between *parameter* entities, which start with "%", and *general* entities, which start with "&". Parameter entities are used within DTD declarations. Text entities are used within document content. Because the "included-domains" entity is contributing to the value of an attribute it is a general entity, not a parameter entity, and the entities used to build up its value are also general entities, not parameter entities.

That's all there is to it. Now just repeat Step 2 for each document type shell you use and you're done.

Tip: Common mistakes to watch out for:

- Forgetting the closing ';' (semicolon) on entity references.
- Using "&" for parameter entities or or "%" for general entities. A parameter entity reference within an attribute value is parsed as literal text, so it won't be reported by XML parsers.

Attribute Specialization Step 3: Test the Declarations

Step 3 is to test your declarations to make sure they work. This is simply a matter of creating an XML document that uses your local document type shell as its DTD and verifying that the `phase-of-moon=` attribute is now available on all elements that allow the selection attributes.

Attribute Specialization: XSD Version

As for DTD-based attribute domains, an XSD attribute domain module consists of a single XSD document whose name is "*attributeName*AttDomain.xsd", e.g., `phase-of-moonAttDomain.xsd`.

The XSD document contains a single `<xs:attributeGroup>` declaration:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:attributeGroup name="phase-of-moon-d-attribute">
    <xs:attribute name="phase-of-moon" type="xs:string"/>
  </xs:attributeGroup>
</xs:schema>buteGroup>
</xs:schema>
```

You integrate the attribute into shell XSDs by including the attribute module document and adding an entry for the attribute domain to the `domains=` attribute declaration. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  xmlns:ditaarch="http://dita.oasis-open.org/architecture/2005/">

  <xs:include schemaLocation="xsd/phase-of-moonAttDomain.xsd" />

  <!--  ================ GROUP DEFINITIONS ====================  -->
  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:metaDeclGrp.xsd:
1.2"/>
  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:tblDeclGrp.xsd:
1.2"/>
  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:topicGrp.xsd:
1.2"/>

  <!-- ================  MODULE INLCUDE DEFINITION  =================  --
>
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:commonElementMod.xsd:1.2"/>
  <!-- ======== Table elements ======== -->
  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:tblDeclMod.xsd:
1.2"/>
  <!-- ======= MetaData elements, plus keyword and indexterm ======= -->
  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:metaDeclMod.xsd:
1.2"/>


  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:topicMod.xsd:
1.2"/>

  <xs:redefine
schemaLocation="urn:oasis:names:tc:dita:xsd:commonElementGrp.xsd:1.2">
    <xs:attributeGroup name="props-attribute-extensions">
      <xs:attributeGroup ref="props-attribute-extensions"/>
      <xs:attributeGroup ref="phase-of-moon-d-attribute"/>
    </xs:attributeGroup>
 </xs:redefine>
```

```
  <xs:group name="info-types">
    <xs:sequence/>
  </xs:group>

  <xs:attributeGroup name="domains-att">
    <xs:attribute name="domains" type="xs:string" default="a(props phase-
of-moon)"/>
  </xs:attributeGroup>

</xs:schema>
```

This example is a topic shell integrates only the attribute domain. In practice you would normally include a number of element domains in addition to any attribute domains.

Element Domain Specialization Tutorial

Goal: Declare an element domain vocabulary module that provides elements for identifying mentions of XML constructs: element types, attributes, text entities, parameter entities, and numeric character references.

In any XML vocabulary for documents there is a general class of elements whose purpose is to hold the names or labels of real things in order to indicate what type of thing the name or label is for. A typical example would be an element named "person" whose purpose is to hold the names of people. Likewise, an element named "color" would identify the names of colors. Given the string "Brown" an author would tag it either `<person>Brown</person>` to indicate a reference to a Mr or Ms Brown or `<color>brown</color>` to indicate a mention of the color brown. This markup then enables finding all mentions of people named "Brown" but not the color brown. It also enables applying different typographic effects to person names and color names, if necessary. They can also enable things like automatic indexing where the element type is the primary entry (e.g., "colors") and the element content is used for secondary entries (e.g., "colors, brown", "colors, blue").

In my practice as a markup designer I refer to these types of elements as "mention" elements because their primary purpose is to identify mentions of things. In DITA the base element type for mentions is `<keyword>`. In technical documentation in particular mentions are very important because technical documentation is about real things that usually have specific component types or properties that need to be clearly identified, listed, searched on, and visually distinguished.

For Publishing applications, mentions are less important only because Publishing content tends to be less domain-specific and because the cost of adding mention markup is often not justified by the value, especially where content is not authored in XML originally. However, some Publishing content does benefit from specialized mention elements, especially reference content like travel and nature guides, where identifying things like place names or species names becomes important for both typography and search.

In DITA, mention elements can use the `keyref=` attribute to both get their effective text and become navigation links to the things they mention. This allows you to have mentions of things directly connected to their formal definition (e.g., in reference topics). This makes mention elements very powerful and a place where a relatively small investment in specialization can provide huge value for authors.

After attribute specialization, creating specialized mention elements is probably the easiest specialization to do and provides immediate value to authors. Because you don't normally need to worry about the content model details, this is entirely an exercise in naming and copying.

As much of what I write about involves XML markup, I find it necessary to have elements that identify mentions of different XML components: tag names (element types), attribute names, parameter entities, text (general) entities, and numeric character references. These components need distinguishing typographic presentation but that presentation can vary in different contexts. For example, I have historically used the convention "*attname*=" for attributes, but the DITA TC resolved on the convention "*@attname*" for attributes in the course of the development of DITA 1.2. By having markup for attribute name mentions it is easy to modify the style sheets to change from "*attname*=" to "*@attname*".

For this tutorial you will define an element domain module that provides the element types <xmlelem> (XML element), <xmlatt> (XML attribute), <textent> (general entity), <parment> (parameter entity), and <numcharref> (numeric character reference, e.g., "—").[5]

Element domains are typically either topic domains or map domains, meaning that they specialize either elements exclusively allowed within topics or exclusively within maps. However, if a domain only specializes elements that are common to maps and topics, then the domain may be used in both maps and topics. This includes domains that specialize from <text>, <keyword>, <term>, or <ph> and domains that specialize from <data>. This allows you to use the same mention elements within maps (e.g., within titles defined in maps and within metadata) and within topics.

Element Domain Specialization Process Overview (DTDs)

A DTD-syntax element domain vocabulary module consists of two files: one that declares the element types and one that defines parameter and text entities used to integrate the module into shell document types.

The element type declaration file has the extension ".mod", the entity declaration file has the extension ".ent". These two files together comprise the complete domain module.

The process of creating a new domain module and integrating it into a document type shell DTD is as follows:

1. Work out the design of the new element types, including what element types they are specializations of.
2. Declare the elements in the domain's .mod file.
3. Declare the integration entities in the domain's .ent file.
4. Integrate the domain into one or more documen type shell DTDs using the normal integration and configuration mechanisms defined by the DITA architecture
5. If necessary, provide extensions for DITA processors to apply specialization-specific processing to your specialized types.

   For example, if your specializations require a specific formatting effect that is not the default DITA behavior for the base types, you must extend the output processors you use to provide the formatting effect. Typically this means creating extension XSLT scripts for use with the DITA Open Toolkit, but it can also mean extending built-in editor configurations and style sheets, or extending or customizing content management systems, Web sites, and so on, depending on the nature of the specialization.

Element Domain Specialization Step 1: Design The Domain Element Types

The first step in specialization is designing your markup.

[Reference to more general discussion of markup analysis and design.] For this exercise, the element types to be declared are all specializations of <keyword>, which is a pretty typical use case: you just need a few additional keyword-type elements to mark up mentions of things that are specific to your documentation.

The element types to be declared are:

<xmlelem> Identifies mentions of XML element types. Intended typographic effect is to put the name in a monospaced font, surrounded by angle brackets: <an-element>.

<xmlatt> Identifies mentions of XML attributes. Intended typographic effect is to put the name in monospaced font with a leading "@" sign: an-attribute=.

<textent> Identifies mentions of XML text entities. Intended typographic effect is to put the name in monospaced font with a leading ampersand and trailing semicolon: &a-text-entity-name;. (To be pedantic about it, the formatting specification actually produces an entity *reference* rather than a mention of an entity *name*, but Charles Goldfarb and I are probably the only people on the planet who would both notice the difference and care about the distinction. So we put the closing semicolon ("entity reference close") because that's how people expect something starting with an ampersand to look. Alternatively I could have used a different

[5] This domain is also defined as part of the DITA for Publishers vocabulary.

typographic convention [say small caps or something] and omitted the ampersand, but that would probably have just made it more confusing.)

<parment> Identifies mentions of XML parameter entities. Intended typographic effect is to put the name in monospaced font with a leading percent sign and a trailing semicolon: `%a-parameter-entity-name;`. (To be pedantic again, technically, the "%" is part of the parameter entity name and also serves as the entity reference open, so even if for text entities we didn't show the ampersand we would show the "%" for parameter entity references. Again, probably only Charles Goldfarb and I know or care about the difference. But I am compelled by years of deep standards work and an uncontrollable pedantic streak to mention that there is in fact a difference. Not that anyone cares. And yes, I was that kid who always said "technically, a peanut is not a nut, it's a legume.")

<numcharref> Identifies mentions of XML numeric character references. Intended typographic effect is to put the number in monospaced font with a leading "numeric character reference open" (&#) and closing semicolon: `&#x2014;`. The content should be the numeric value, preceded by an "x" for hex values. (I don't even have to say it do, I?)

None of these elements allow any subelements, so their content models will all be just #PCDATA.

Domain modules must be given a short name that is used in the filenames, entity names, and in the `class=` attributes. For this tutorial, the domain module name is "xml". In practice, you should give your domains more distinctive names so that there is minimal chance of name collisions with other domains. You see this with the DITA 1.2 Learning and Training specializations, where both the domains and the element types all start with "lc" (for "Learning Content"). This is essentially the same as having a namespace prefix on your names, but of course DITA 1.x cannot use namespaces, so you have to make do with the simple convention of distinguishing bits in your names.

Element Domain Specialization Step 2: Declare The Domain Element Types

The element type declarations go in a file called "*modulename*Domain.xml".

The easiest way to do this is to copy an existing domain module file and use it as a template for creating your new one. However, for this tutorial I will describe the process as though you were creating the file from scratch.

Step 2-1. Create New .mod File

Create a new file named "xmlDomain.mod" and put a descriptive header comment at the top:

```
<!-- ============================================================
     XML Construct Domain Module

     Author: your name here

     Copyright (c) 2010 copyright holder

     license to use or not use or whatever

     ============================================================ -->
```

Step 2-2. Declare Element Type Name Parameter Entities

For each element type, declare a parameter entity that reflects the element type name:

```
<!-- ============================================================
     XML Construct Domain Module

     Author: your name here

     Copyright (c) 2010 copyright holder

     license to use or not use or whatever
```

```
     ================================================== -->

<!ENTITY % xmlelem    "xmlelem"  >
```

These entities allow the element type to be used in content models via the parameter entity. Document type shells can then integrate further specializations of this element type by redeclaring the parameter entity to include specializations of the element, which then are automatically allowed wherever the base element type is allowed. You can also use the parameter entity in constraint modules to specifically allow only the element type in place of its base type.

Step 2-3. Declare Elements and Attributes

For each element type, create an element type and attribute declaration:

```
<!-- ==================================================
     XML Construct Domain Module

     Author: your name here

     Copyright (c) 2010 copyright holder

     license to use or not use or whatever
     ================================================== -->

<!ENTITY % xmlelem    "xmlelem"  >

<!--                    LONG NAME: XML Element
-->
<!ENTITY % xmlelem.content
  "
  (#PCDATA)*
  "
>
<!ENTITY % xmlelem.attributes
  '
  %univ-atts;
  keyref
    CDATA
    #IMPLIED
  outputclass
    CDATA
    #IMPLIED
  '
>
<!ELEMENT xmlelem %xmlelem.content; >
<!ATTLIST xmlelem %xmlelem.attributes; >
```

Note that the content model for each element type must be at least as restrictive as the content model of the specialization base (in this case, `<keyword>`, although we haven't declared that yet). Looking at the DITA language reference (or the declaration in commonElements.mod), we see that the content model for `<keyword>` includes both #PCDATA as well as all other phrase-level elements. Since the XML component mentions don't need or want any subelements, we've reduced the content model down to just #PCDATA (just text), which is consistent with the content model of the `<keyword>` element.

Note also that the content model and attribute lists are defined as parameter entities that are then used in the actual ELEMENT and ATTLIST declarations. This is a change from DITA 1.1 to DITA 1.2 made to enable overriding of individual element's content models and attribute lists through constraint modules.)In this case the only content model more restrictive than "`(#PCDATA)*`" is "`EMPTY`".)

Tip:

Mistake to watch out for: Attributes with quoted default values quoted with the same quote character as the parameter entity text.

In parameter entity declarations you can use either single quotes or double quotes (" or ') to quote the entity value. Because attributes can have literal default values, you have to make sure the quotes used for the attribute value are different from the quotes used for the parameter entity text itself. This is most common with specializations of `<data>` where you want to set the `name=` attribute to a specific value (typically the tagname of the specialization).

For attribute declarations it's usually easiest to use single quotes so that attributes that specify literal default values can use double quotes, which is how most American's will declare them (and thus what you are most likely to copy from the base DITA declarations). Of course, if you're British you can swap that around.

An attributes parameter entity with a quote default value would look like this:

```
<!ENTITY % myDataElem.attributes
  '
    name
       CDATA
       "myDataElem"
    value
       CDATA
       #REQUIRED
    %univ-atts;
  '
>
```

Step 2-4. Declare `class=` Attributes

For each element type, create a separate attribute declaration that defines the `class=` attribute value:

```
<!-- ============================================================
     XML Construct Domain Module

     Author: your name here

     Copyright (c) 2010 copyright holder

     license to use or not use or whatever
     ============================================================ -->

<!ENTITY % xmlelem    "xmlelem"  >

<!--                    LONG NAME: XML Element
-->
<!ENTITY % xmlelem.content
  "
  (#PCDATA)*
  "
>
<!ENTITY % xmlelem.attributes
  '
  %univ-atts;
  keyref
    CDATA
    #IMPLIED
  outputclass
    CDATA
    #IMPLIED
  '
>
<!ELEMENT xmlelem %xmlelem.content; >
<!ATTLIST xmlelem %xmlelem.attributes; >
```

```
<!ATTLIST xmlelem     %global-atts;  class CDATA "+ topic/keyword xml-d/
xmlelem "  >
```

(The `%global-atts;` parameter entity is defined in one of the base DITA module files.)

There are several reasons for keeping the `class=` attribute declaration separate from the main element and attribute declaration. By putting all the class attribute declarations together in one place, it makes it easier to find the declarations and see what the specialization hierarchy is. It also makes it easier to re-use the element declarations by cut and paste. (This pattern becomes much more critical in XSD-based specialization, where the re-use of base element type declarations is by reference rather than via cut and paste.)

The complete `xmlDomain.mod` file should look this this:

```
<!-- ============================================================

     XML construct domain

     Provides phrase-level elements for identifying mentions of
     XML constructs: element types, attributes, etc.

     Copyright (c) 2010 copyright holder

     license to use or not use or whatever

     ============================================================ -->

<!-- ============================================================ -->
<!--                    ELEMENT NAME ENTITIES                     -->
<!-- ============================================================ -->

<!ENTITY % xmlelem "xmlelem"  >
<!ENTITY % xmlatt  "xmlatt"   >
<!ENTITY % textent "textent"  >
<!ENTITY % parment "parment"  >
<!ENTITY % numcharref "numcharref" >

<!-- ============================================================ -->
<!--                     ELEMENT DECLARATIONS                     -->
<!-- ============================================================ -->


<!--                     LONG NAME: XML Element
-->
<!ENTITY % xmlelem.content
  "
  (#PCDATA)*
  "
>
<!ENTITY % xmlelem.attributes
  '
  %univ-atts;
  keyref
    CDATA
    #IMPLIED
  outputclass
    CDATA
    #IMPLIED
  '
>
<!ELEMENT xmlelem %xmlelem.content; >
<!ATTLIST xmlelem %xmlelem.attributes; >
```

```
<!--                    LONG NAME: XML
Attribute                              -->
<!ENTITY % xmlatt.content
  "
  (#PCDATA)*
  "
>
<!ENTITY % xmlatt.attributes
  '
  %univ-atts;
  keyref
    CDATA
    #IMPLIED
  outputclass
    CDATA
    #IMPLIED
  '
>
<!ELEMENT xmlatt %xmlatt.content; >
<!ATTLIST xmlatt %xmlatt.attributes; >

<!--                    LONG NAME: Text entity -->
<!ENTITY % textent.content
  "
  (#PCDATA)*
  "
>
<!ENTITY % textent.attributes
  '
  %univ-atts;
  keyref
    CDATA
    #IMPLIED
  outputclass
    CDATA
    #IMPLIED
  '
>
<!ELEMENT textent %textent.content; >
<!ATTLIST textent %textent.attributes; >

<!--                    LONG NAME: Parameter entity -->
<!ENTITY % parment.content
  "
  (#PCDATA)*
  "
>
<!ENTITY % parment.attributes
  '
  %univ-atts;
  keyref
    CDATA
    #IMPLIED
  outputclass
    CDATA
    #IMPLIED
  '
>
<!ELEMENT parment %parment.content; >
<!ATTLIST parment %parment.attributes; >

<!--                    LONG NAME: Numeric character reference -->
<!ENTITY % numcharref.content
  "
```

```
  (#PCDATA)*
  "
>
<!ENTITY % numcharref.attributes
  '
  %univ-atts;
  keyref
    CDATA
    #IMPLIED
  outputclass
    CDATA
    #IMPLIED
  '
>
<!ELEMENT numcharref %numcharref.content; >
<!ATTLIST numcharref %numcharref.attributes; >


<!-- ============================================================= -->
<!--                    SPECIALIZATION ATTRIBUTE DECLARATIONS      -->
<!-- ============================================================= -->


<!ATTLIST xmlelem     %global-atts;  class CDATA "+ topic/keyword xml-d/
xmlelem "  >
<!ATTLIST xmlatt      %global-atts;  class CDATA "+ topic/keyword xml-d/
xmlatt "  >
<!ATTLIST textent     %global-atts;  class CDATA "+ topic/keyword xml-d/
textent "  >
<!ATTLIST parment     %global-atts;  class CDATA "+ topic/keyword xml-d/
parment "  >
<!ATTLIST numcharref  %global-atts;  class CDATA "+ topic/keyword xml-d/
numcharref "  >


<!-- ================== End XML Domain ==================== -->
```

You should be seeing that this is a largely mechanical process that is mostly cutting and pasting of repeated declaration patterns. All of the actual thinking goes into the element type design. The declaration activity is purely mechanical.

In particular, having created the first declaration for `<xmlelem>`, you copy it, paste it, and simply change "xmlelem" to the new element type name everywhere it occurs in that element's declarations.

Element Domain Specialization Step 3: Declare The Module Entities File

The module entities file declares entities that are used within document type shell DTDs to integrate the module into the shell.

The module entities file is named "*modulename*Module.ent".

There are two types of entities: type-specific parameter entities that integrate the module's element types into the appropriate content models (phrase, keyword, dl, etc.) and a domain usage declaration text entity that goes in the topic's or map's domain use declaration attribute to indicate what domains are being used in a given topic or map.

For the XML domain we only have element types specialized from `<keyword>`, so we only need to declare one type-specific parameter entity. The parameter entity is named "*modulename*-d-*basetype*", so in this case, the entity will be named "xml-d-keyword".

Step 3-1. Create xmlDomain.ent

Create a new file named "`xmlDomain.ent`" and put a descriptive header comment at the top:

```
<!-- ================================================
```

```
     XML Construct Domain Module

     Author: your name here

     Copyright (c) 2010 copyright holder

     license to use or not use or whatever

     ================================================== -->
```

Step 3-2. Declare Type-Specific Integration Entities

Declare a type-specific entity for the keyword-based elements, listing all the element types in the module:

```
<!-- ==================================================
     XML Construct Domain Module

     Author: your name here

     Copyright (c) 2010 copyright holder

     license to use or not use or whatever

     ================================================== -->

<!ENTITY % xml-d-keyword
  "xmlelem |
   xmlatt |
   textent |
   parment |
   numcharref
  "
  >
```

Step 3-3. Declare Domain Usage Text Entity

Declare the domain usage declaration text entity, which is named "*modulename*-d-att":

```
<!-- ==================================================
     XML Construct Domain Module

     Author: your name here

     Copyright (c) 2010 copyright holder

     license to use or not use or whatever

     ================================================== -->

<!ENTITY % xml-d-keyword
  "xmlelem |
   xmlatt |
   textent |
   parment |
   numcharref
  "
  >

<!ENTITY   xml-d-att     "(topic xml-d)"
>
```

Common Error: Do not accidently declare the domains attribute contribution entity as a parameter entity. That is, you want this:

```
<!ENTITY xml-d-att ...
```

not this:

```
<!ENTITY % xml-d-att
```

The " % " indicates a *parameter* entity and for the domains attribute contribution you want a *text* entity.

The keyword topic indicates that this is a topic domain rather than a map domain. The value "xml-d" is the name of this module, the "-d" indicating that it is a domain module and not a topic module.

Note however that this domain specializes exclusively from an element type allowed in both maps and topics (`<keyword>`), so even though this is a topic domain it may be used with maps as well as topics.

With the `xmlDomain.mod` and `xmlDomain.ent` files you now have a complete module declaration set ready to be integrated with any shell DTDs that need to use it.

Element Domain Specialization Step 4: Integrate The Module Into a Document Type Shell

Integration is the process of modifying a document type shell DTD to include different modules. It is also a purely mechanical process.

Normal practice when using DITA should be that the first thing you do is make local copies of all the DITA-provided shell DTDs, even if you don't change them in any way. This prepares you for the inevitable time that you want to add or remove domains or otherwise configure or extend from the base DITA types.

Note that the order in which the .ent and .mod files are included into the shell DTD is very important. This is because in DTDs, the order of occurrence of parameter entity declarations is significant. In XML, the first declaration of a given entity name wins, so in order to override entities defined in other included files, you must declare that entity first. Thus you import the .ent files first, which provides the module-specific entity declarations that override the default definitions provided in the .mod files. Then you define the values of any configuration parameter entities, then you include the .mod files, which use the configuration parameter entities to define the effective values for content models and attribute lists.

Step 4-1. Setup Local Copy of Document Type Shell DTD

Make a local copy of the appropriate document type shell DTD. The DITA-defined declarations include both all the DITA-defined modules as well as base shells for all of the DITA-defined top-level topic and map types: map, topic, ditabase, bookmap, concept, reference, task, glossentry, and so on. These shells are intended to be used as the starting point for creating local customized shells that reflect your local topic and domain requirements.

Copy the `concept.dtd` file from the DITA distribution to some location outside the scope of the DITA Open Toolkit (e.g., c:\mystuff\dita\dtd\concept.dtd).

Step 4-2. Create a Test Topic

Create a test document that will test the new document type shell by using it as its DTD. Put this file in the same directory as the shell DTD or in a nearby directory. The file should look like this:

```
<?xml version="1.0"?>
<!DOCTYPE concept SYSTEM "concept.dtd">
<concept id="topicid">
<title>Test concept</title>
</concept>
```

Modify the system identifier (the part in bold) to reflect the actual location of the shell DTD relative to the test document.

Remember that in XML all system identifiers are URIs, meaning you're specifying a URL, not a system-specific file path. So no backslashes if you're on Windows.

Validate this document. It should be valid at this point as you haven't modified the document type shell file in any way.

Step 4-3. Add .ent File to Shell

Add the domain entity declaration (.ent) file to concept.dtd. In your copy of concept.dtd, find the comment that says "DOMAIN ENTITY DECLARATIONS" and following this comment add the following parameter entity declaration and reference:

```
<!ENTITY % xml-d-dec
   SYSTEM "xmlDomain.ent"
>
%xml-d-dec;
```

The system identifier needs to reflect the actual location of the xmlDomain.ent file. The example assumes that concept.dtd and xmlDomain.ent are in the same directory.

Step 4-4. Update Type-Specific Parameter Entities in Shell DTD

Add the module type-specific parameter entities to the domain extension parameter entities. In concept.dtd, find the comment that says "DOMAIN EXTENSIONS". Update the `%keyword;` parameter entity declaration to include the `%xml-d-keyword;` parameter entity:

```
<!ENTITY % keyword
  "keyword |
   %xml-d-keyword;
  "
>
```

This declaration has the effect of including all the XML module's keyword-type elements wherever keyword is allowed in the base DITA-defined content models.

Step 4-5. Update `domains=` attribute with new domain

Add the XML domain to the `domains=` attribute declaration. In concept.dtd, find the comment that says "DOMAINS ATTRIBUTE OVERRIDE". Update the `&included-domains;` entity declaration to include the `&xml-d-att;` text entity:

```
<!ENTITY included-domains
   "&hi-d-att;
    &ut-d-att;
    &xml-d-att;
   "
>
```

Note that your `&included-domains;` value may be different. The example shows the value used for the concept shell used by this tutorial's topics, which includes only the highlight and utility domains.

Step 4-6. Include .mod Declaration Set

Include the domain element declarations. In concept.dtd, find the comment that says "DOMAIN ELEMENT INTEGRATION". Following that comment, add a declaration and reference for the xmlDomain.mod declaration set:

```
<!ENTITY % xml-d-def
  SYSTEM  "xmlDomain.mod"
>
%xml-d-def;
```

As for the .ent file, the actual system identifier needs to reflect the actual location of your version of xmlDomain.mod.

Step 4-7. Test the integration

Using an XML editor and the test concept you created in step 2 above, first verify that the document is still valid, which verifies that you didn't break anything in doing the integration of concept.dtd. If that succeeds, then add a `<conbody>`, a `<p>`, and verify that you can now add any of the XML domain elements within a paragraph.

If this succeeds, you're done. You have successfully defined a new domain and integrated it into a shell document type. Now all you have to do is repeat (that is copy) the shell integrations into the other shells you'll be using the domain with.

The next step, if needed, is to add domain-specific functionality to your DITA processors. For the XML domain, that means updating output processors to create the formatting effects for the different XML component mention elements.

Element Domain Specialization Step 5: Extend DITA Open Toolkit XHTML Processor

Per the documentation of the types in the XML domain, we need to create XSLT templates for each of the different mentions to produce the appropriate output effects.

Because the XML domain only defines specializations of `<keyword>`, the XSLT required is quite simple and requires very little knowledge of XSLT itself.

The Open Toolkit is designed to allow extension of base functionality through "plugins". This allows you to create processing specific to your specialization and then easily deploy it without the need to directly modify the base Toolkit processes. All Toolkit transformation types *should* provide appropriate extension points for plugins, but not all do. However, both the HTML and PDF transformation types do provide the extension points you need to easily add processing for new specializations.

The basic process is as follows:

1. Create an empty starting XSLT stylesheet in a directory you control (that is, outside the scope of the DITA Open Toolkit) and add an import of the base Toolkit-supplied HTML transform.
2. Create type-specific match templates for the of the element types in the XML domain.
3. Package the transform as an Open Toolkit plugin.
4. Deploy the plugin and test it.

Element Domain Specialization Step 5-1: Create Blank Specialization-Specific XSLT Transform Plugin

Create a directory to hold the plugin. As for document type plugins (see [reference to doctype plugin packaging section]), you should give your plugin a globally unique name, such as a Java-style package name like "org.example.xmldomain.html". The plugin name should reflect both the domain it supports and the transformation type it extends so it's clear what it's purpose is. As a matter of practice, I like to use the same engineering practice for Toolkit plugins that one normally uses for Java classes, namely keep them small and focused. Thus I typically have a separate plugin for each domain/transformation type pair. By naming them all with the same "package" prefix it is easy to see all the related plugins in a Toolkit's `plugins` directory. It aslo makes it easy to do things like use Ant to manipulate a set of plugins (for example, to package them for distribution or automate deployment from your development environment to your test Toolkit environment).

Because the plugin will contain a normal XSLT module, you can implement and test the XSLT in isolation before you build and test your plugin. This is easiest in the context of an XSLT integrated development environment like OxygenXML or XML Spy, which handles the details of applying the transform to your test data.

In your chosen directory, create the file "xmlDomain2html.xsl" and give it this content:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- ============================================================
     HTML generation templates for the XML Construct DITA domain
     specialization

     Copyright (c) 2010 Your Name Here
```

```
    ============================================================= -->
<xsl:stylesheet version="2.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">


</xsl:stylesheet>
```

As of version 1.5, the Open Toolkit requires the use of the Saxon XSLT engine, which means you can use XSLT 1 or XSLT 2. For this exercise the processing is so simple that it doesn't matter whether you use version 1 or 2. However, since the Toolkit allows XSLT 2 there's no reason not to declare your stylesheet as being an XSLT 2 style sheet.

For more complicated processing you will definitely want to use XSLT 2.

To be a Toolkit plugin, the directory must contain a file named `plugin.xml`. This file declares the name of the plugin and otherwise configures the plugin [reference to more general section on creating Toolkit plugins]. For this simple plugin, the plugin descriptor is:

```
<plugin id="org.example.xmldomain.html">
  <require plugin="org.example.xmldomain.doctypes"/>
  <feature extension="dita.xsl.xhtml" value="xmlDomain2html.xsl"
type="file"/>
</plugin>
```

The part in bold is the plugin name. The name must be unique across all plugins installed in a given Toolkit, so it's important to give the plugin a globally unique name. Again, using Java-style package names ensures that.

The `<require>` element indicates that this plugin requires the corresponding doctype plugin that contains the specialization module itself. The Toolkit's integration process checks this dependency and makes sure that all the necessary parts are available.

The `<feature>` element says that the XSLT file `xmlDomain2html.xsl` hooks into the extension point named "dita.xsl.xhtml" and is of type "file". This tells the Toolkit's integration process to add a reference to the file `xmlDomain2html.xsl` at the point in the base HTML files where the extension point "dita.xsl.xhtml" is defined. Note that you don't care where that extension point is defined, you only care about the name. Any number of plugins can extend that same extension point.

Tip: Common errors to watch for:

- Copying a plugin.xml file and forgetting to change the `id=` value. This will cause confusing errors in the Toolkit processing that can be hard to diagnose.

Element Domain Specialization Step 5-2: Implement Type-Specific XSLT Templates

The different element types in the XML domain require different "decorations" on output to visually distinguish them. This could be done via CSS if we put appropriate `class=` values on the HTML output or it can be done here using literal text. The advantage of using literal text is that it will work even when the CSS isn't present or supported by the browser. The advantage of using CSS is that you can change look without regenerating the HTML. For this tutorial the point is to learn to do some stuff with XSLT so we are going to use literal text.

The pattern for each element's template is the same:

```
<xsl:template match="*[contains(@class, ' xml-d/typename ')]">
  <code>
    <xsl:text>text before</xsl:text>
    <xsl:apply-templates/>
    <xsl:text>text after</xsl:text>
  </code>
</xsl:template>
```

The `priority=` attribute ensures that this template will be used instead of any imported template. The `<xsl:text>` elements hold the generated text. This could be replaced with other HTML elements if you wanted some more complicated effect, like blue brackets or a superscript or something.

Note the leading and trailing spaces around the class attribute value (" xml-d/xmlelem "). It is very important to include those spaces so that you only match on the specific element types you want and not on types whose name happens to begin with the same letters.

The template uses the <code> element as the main HTML element for the XML components as it seems to be the closest match for the semantic of these keywords and we want the default HTML behavior of putting the text into a monospaced font.

For the <xmlelem> element, we need to generate the bounding angle brackets. Using the above model template as a template, create this template following the include statement in the xmlDomain2HTML.xsl file:

```
  <xsl:template match="*[contains(@class, ' xml-d/xmlelem ')]"
priority="10">
    <code>
      <xsl:text>&lt;</xsl:text>
      <xsl:apply-templates/>
      <xsl:text>&gt;</xsl:text>
    </code>
  </xsl:template>
```

The other templates are just like this one, differing only in the text before and text after they produce. Simply cut and paste the template for <xmlelem> and modify it as needed for each of the other types.

The complete XSLT stylesheet should look like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- ============================================================
     HTML generation templates for the xmlDomain DITA domain.

     Copyright (c) 2010 Your Name Here

     ============================================================ -->

<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/
Transform">

  <xsl:template match="*[contains(@class, ' xml-d/xmlelem ')]"
priority="10">
    <code>
      <xsl:text>&lt;</xsl:text>
      <xsl:apply-templates/>
      <xsl:text>></xsl:text>
    </code>
  </xsl:template>

  <xsl:template match="*[contains(@class, ' xml-d/xmlatt ')]"
priority="10">
    <code>
      <xsl:text>@</xsl:text>
      <xsl:apply-templates/>
    </code>
  </xsl:template>

  <xsl:template match="*[contains(@class, ' xml-d/textent ')]"
priority="10">
    <code>
      <xsl:text>&amp;</xsl:text>
      <xsl:apply-templates/>
      <xsl:text>;</xsl:text>
    </code>
  </xsl:template>

  <xsl:template match="*[contains(@class, ' xml-d/parment ')]"
```

```
priority="10">
    <code>
      <xsl:text>%</xsl:text>
      <xsl:apply-templates/>
      <xsl:text>;</xsl:text>
    </code>
  </xsl:template>

  <xsl:template match="*[contains(@class, ' xml-d/numcharref ')]"
priority="10">
    <code>
      <xsl:text>&amp;#</xsl:text>
      <xsl:apply-templates/>
      <xsl:text>;</xsl:text>
    </code>
  </xsl:template>
</xsl:stylesheet>
```

That's all there is to it, at least for these relatively simple keyword-based specializations. A specialization of something more sophisticated, like `<simple-table>` or `<dl>` could, of course, require more work. But just adding a few `<ph>` or `<keyword>` specializations and styling them is simple enough that anyone should be able to do it if they need to.

For XSL-FO output, the stylesheet would be almost the same, except that you would be generating `<fo:inline>` elements instead of HTML `<code>` or `<span>` elements.

Element Domain Specialization Step 5-3: Test The Stylesheet

To test the stylesheet, deploy the plugin to your Toolkit and run the Toolkit's HTML transformation type against the test topic you created in Step 2

If you have an XSLT development environment such as OxygenXML or XMLSpy, you can also run the stylesheet directly against a topic—you won't get any processing that requires resolution of conrefs or generation of links but you'll be able to validate that the XSLT is producing the correct XSLT markup.

To deploy the plugin to your Toolkit, do the following:

1. Copy the plugin's directory to your Toolkit's `plugins` directory.
2. Run the integrator.xml Ant task:

   ```
   c:\DITA-OT > ant -f integrator.xml
   ```

   You should see output like this:

   ```
   Buildfile: integrator.xml

   integrate:
   [integrate] Using XERCES.

   BUILD SUCCESSFUL
   Total time: 2 seconds
   ```

Now run the Toolkit's HTML transformation type against your test topic as you normally would. You should see the new formatting for the various element types.

Element Domain Specialization: XSD Version

An XSD element domain module consists of a single XSD document that contains all the declarations for the domain.

A each element type in an element domain module requires the following declaration components:

- A group that defines the content model for the element type
- A group that defines the attribute list for the element type
- A complex type that combines the content model and the attributes for the element type and, where appropriate, defines the content model as mixed.

- A element type that uses the complex type and declares the `class=` attribute for the element type.
- A group that includes just the element by which the element type can be referenced in content models.

For the element types in the XML domain these components look like this:

```
<xs:element name="xmlelem">
  <xs:annotation>
    <xs:documentation>
      The XML element (&lt;<keyword>xmlelem</keyword>&gt;) element
represents a
      mention of an XML element (tag name).
    </xs:documentation>
  </xs:annotation>
  <xs:complexType mixed="true">
    <xs:complexContent>
      <xs:extension base="xmlelem.class">
        <xs:attribute ref="class" default="+ topic/keyword xml-d/xmlelem "/
>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:group name="xmlelem.content">
  <xs:sequence/>
</xs:group>
<xs:attributeGroup name="xmlelem.attributes">
  <xs:attribute name="outputclass" type="xs:string"/>
  <xs:attribute name="keyref" type="xs:string"/>
  <xs:attributeGroup ref="global-atts"/>
  <xs:attributeGroup ref="univ-atts"/>
</xs:attributeGroup>
<xs:group name="xmlelem">
  <xs:sequence>
    <xs:choice>
      <xs:element ref="xmlelem"/>
    </xs:choice>
  </xs:sequence>
</xs:group>
<xs:complexType name="xmlelem.class" mixed="true">
  <xs:sequence>
    <xs:group ref="xmlelem.content"/>
  </xs:sequence>
  <xs:attributeGroup ref="xmlelem.attributes"/>
</xs:complexType>
```

The individual components can go in any order but it makes sense to put the `<xs:element>` declaration first as it contains the documentation for the element type.

The only other required components are the groups used to enable integration of the domain on a per-element-type basis. For this domain all the element types specialize from topic/keyword so there is only one group required, which should be named "xml-d-keyword":

```
  <xs:group name="xml-d-keyword">
    <xs:choice>
      <xs:element ref="xmlelem"/>
      <xs:element ref="xmlatt"/>
      <xs:element ref="numcharref"/>
      <xs:element ref="parment"/>
      <xs:element ref="textent"/>
    </xs:choice>
  </xs:group>
```

This group can go at the top or bottom of the XSD document. I prefer to put it at the top where it's easy to find.

To implement the XML domain as an XSD vocabulary module, follow these steps:

1. Create a file named `xmlDomain.xsd` with this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

</xs:schema>
```

2. Copy the declarations shown above for the `<xmlelem>` element type and paste them in, one copy for each of the five element types.
3. Change the base tagname in each copy to the appropriate tag name.
4. Copy the xml-d-keyword group shown above and pasted it at the top of the XSD file.

To test the module in isolation, create a topic document type shell XSD file in a convenient directory relative to where you have saved the domain's XSD file, e.g., in the directory above it or in one nearby. This lets you create relative URLs from the shell XSD to the module so you don't have to set up an entity resolution catalog just to test the module. For example, you can just copy the topic.xsd file from the standard DITA schema files.

To integrate the new domain module, edit the shell XSD as follows:

1. Add a reference to the `xmlDomain.xsd` module at the top of the schema document

```
  ...
  <!--  ================ TOPIC DOMAINS ====================  -->
  <xs:include schemaLocation="xsd/xmlDomain.xsd" />

  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:uiDomain.xsd:
1.2"/>
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:softwareDomain.xsd:1.2"/>
  ...
```

   Change the value of the `schemaLocation=` attribute to reflect the real relative location of the `xmlDomain.xsd` file. Remember that schema locations are URLs, not filenames, so don't use Windows paths (no backslashes) if you are on Windows.
2. Within the `<xs:redefine>` for the common elements module, update the entry for `<keyword>` to include a reference to the xml-d-keyword group:

```
  ...
  <xs:redefine
schemaLocation="urn:oasis:names:tc:dita:xsd:commonElementGrp.xsd:1.2">
    <xs:group name="keyword">
      <xs:choice>
        <xs:group ref="keyword"/>
        <xs:group ref="pr-d-keyword" />
        <xs:group ref="ui-d-keyword" />
        <xs:group ref="sw-d-keyword" />
        <xs:group ref="xml-d-keyword" />
      </xs:choice>
    </xs:group>
    ...
  </xs:redefine>
  ...
```

Test the domain by creating a topic document that uses the shell you just created. The easiest place to put the document is in the same directory as the shell XSD file. Given that, the document should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<topic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="topic.xsd"
  id="topicid">
  <title>Test Topic</title>
  <body>
    <p><xmlelem>xmlelem</xmlelem></p>
    <p><xmlatt>xmlatt</xmlatt></p>
    <p><numcharref>numcharref</numcharref></p>
    <p><parment>parment</parment></p>
```

```
    <p><textent>textent</textent></p>
  </body>
</topic>
```

You should now be able to validate the document or just apply the Toolkit to it, which will have the effect of validating it.

The last step is to create an entity resolution catalog that assigns a URN to the `xmlDomain.xsd` file. This file should go in the same directory as the module file. The content should be like this:

```
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
prefer="public">

  <public publicId="urn:pubid:example.org:doctypes:dita:modules:xmlDomain"
            uri="xmlDomain.xsd"
    />
  <uri name="urn:pubid:example.org:doctypes:dita:modules:xmlDomain"
    uri="xmlDomain.xsd"
  />

</catalog>
```

You would need to adjust your version to replace the parts shown in bold with your own value, which needs to be globally unique (e.g., an appropriate Internet domain name).

To integrate this module with the Open Toolkit, you should package it as a plugin. See *Packaging Document Type Shells and Vocabulary Modules as Toolkit Plugins* on page 20.

Topic Specialization Tutorial

Goal: Define a new structural specialization of the base `<topic>` element type that supports the creation of FAQ (frequently asked question) topics.

Topic specialization generally requires defining a number of new element types at different levels in the element hierarchy, as opposed to domain specialization, which can be as a simple as defining a few new phrase-level element types.

The main reason to create structural specializations is to provide more-specific markup that reflects your local business requirements or the nature of your information. For most technical documentation applications, the value in specializing reference and task topic types is usually pretty obvious, because these are information types that work best when they directly reflect the details of the things being documented, specific editorial rules for how tasks should be structured, or the needs of other information systems that consume reference and task information (such as interactive task support systems).

Tip: DITA 1.2 introduces a new configuration feature, constraints, which allows you to modify content models without doing specialization. This allows you to reserve specialization for those requirements that truly require new markup. Before DITA 1.2 you also had to use specialization just to define more-constrained versions of base content models even when you weren't changing the meaning of the markup itself. This is no longer necessary in DITA 1.2. See *Topic Constraint Module Tutorial* on page 44.

For reference information, it is usually useful to define specialized reference topic types that directly reflect the objects being documented. For example, if you're documenting sprockets it probably makes sense to have a specialization of "reference" called "sprocket" or "sprocket-definition" that has specialized `<section>` elements that reflect the specific sets of properties or characteristics that sprockets have (tooth properties, shaft properties, material information, manufacturing notes, etc.).

For tasks, you may have specific editorial rules for how tasks should be constructed, rules that are more constraining than the base DITA rules for tasks (which are already pretty constraining). [6]

[6] DITA 1.2 defines a more generic base task type, which provides more opportunity to create specialized tasks that are different from the more-constraint DITA 1.1 `<task>`.

Because conceptual information is, by its nature, more generic, there is usually less need, or less obvious need, to specialize from `<concept>`. For example, the topics for this tutorial are all generic concepts (although they use a specialized domain for identifying mentions of XML constructs). However, there are still many good reasons to specialize concept topic types.

One strong reason to specialize from `<concept>` is to create element types that reflect specific levels in a governing organizational taxonomy where the taxonomy is an integral and invariant aspect of the information. Another reason would be to provide different more-specific concept element types that are familiar to your authors or that reflect a particular editorial style for presenting conceptual information. For example, in the case of this tutorial, the FAQ topic type is based on concept but has been specialized to provide a clear "short answer/ long answer" distinction, as well as disallow base types that we don't want to allow in FAQ topics (abstract in this case). In addition, having a specialized type for FAQ lets us apply FAQ-specific styling to the topics for presentation.

Note that specializing `<concept>` to reflect specific hierarchical levels within a traditional document, e.g., chapter, section, subsection, is normally not a good idea, because it binds a given topic to a specific level, making it harder to re-use or re-organize the topic in other contexts. Rather, if you want markup that directly reflects specific hierarchical levels, you should use map specializations, such as the standard `<bookmap>` or the DITA for Publishers publication map domain.

The only exception to this rule that I can think of is when you have an editorial policy that requires title-only topics (that is, topics with no body) to satisfy specific levels in a governing hierarchy (that is, a governing, invariant, taxonomy). In that case, it can make sense to define specializations of `<concept>` or `<topic>` whose names reflect the taxonomy or hierarchy level and that don't allow either `<body>` or nested topics (meaning that they only serve to be used from maps within a hierarchy of topic references). However, the value of this type of topic is dubious given that `<topichead>` elements within a map are sufficient to establish the hierarchy and provide the necessary titles.[7]

Note that you are not required to specialize from `<concept>`, `<task>`, or `<reference>`. You can specialize directly from `<topic>` or from a more-specialized topic type. The concept/task/reference model makes sense in the context of technical documentation where it reflects a well-established writing practice. But it is not always sensible for other uses of DITA. For example, in the context of Publishing, most content either does not naturally map to one of those three types or those distinctions simply aren't relevant (e.g., within a novel). In that case it can make sense to specialize directly from `<topic>`. For example, the DITA for Publishers vocabulary includes the topic types `<article>`, `<chapter>`, `<part>`, `<subsection>`, and `<sidebar>`, all specialized directly from `<topic>`, because they are intended to represent content at its most generic. The topic types simply provide a more obvious mapping to the basic document components Publishers would expect to see in any Publishing XML application.

Topic Specialization Process Overview

A DTD-syntax topic specialization module consists of two files: a .ent file that defines entities used to integrate the module into document type shell DTDs, and a .mod file that declares the element types. Starting with DITA 1.2 structural modules should provide `domains=` attribute declarations.

For XSD-based structural modules there are two XSD documents: a *Grp.xsd that defines groups used to integrate the module into document type shell XSDs, and a *Mod.xsd that declares the element types and attributes for the module.

The process of creating a new topic module and integrating it into a shell DTD is as follows:

1. Work out the design of the new element types, including what element types they are specializations of
2. Declare the elements in the topic's .mod file
3. Create the .ent file with the appropriate entity declarations
4. Integrate the topic into one or more document type shell DTDs using the normal integration and configuration mechanisms defined by the DITA architecture
5. If necessary, provide extensions for DITA processors to apply specialization-specific processing to your specialized types.

---

[7] DITA 1.2 also clarifies the fact that topicheads can processed as though they referred to title-only topics.

For example, if your specializations require a specific formatting effect that is not the default DITA behavior for the base types, you must extend the output processors you use to provide the formatting effect. Typically this means creating extension XSLT scripts for use with the DITA Open Toolkit, but it can also mean extending built-in editor configurations and style sheets, or extending or customizing content management systems, Web sites, and so on, depending on the nature of the specialization.

Topic Specialization Step 1: Design The Topic Element Types

For topic specialization you have to think about several things in designing your specialization:

- What should the new topic element type name be?
- What should be allowed within the topic body?
- Should the topic allow any nested topics?
- Does the topic require specialized topic-level metadata?

Of course, to answer these questions, you have to first understand the requirements, both for the information content and the information presentation and processing.

For this tutorial, our task is to create a specialization of `<concept>` that supports the requirements of FAQ information, that is, questions and answers.

Note: If you just want to know what the mechanics are and don't really care about how we arrived at the design used in the rest of the tutorial, you can skip on to step 2 at this point. But once you've worked through the tutorial I'd urge you to come back here and read this step all the way through.

I chose FAQs as the subject of the tutorial because they are both familiar to most Web users (and now even non-Web users), they are relatively simple (at least on their face) but not so trivial as to be boring, and they have some potential sophistications that could make for interesting exploration beyond the immediate task of "what are the mechanics of defininining and implementing a new topic type?" In addition, there are any number of useful and reasonable ways that FAQs could be constructed using DITA and what is presented in this tutorial is only one, and not necessarily the best one.

For this tutorial I have decided that the each question should be a separate topic, rather than having one topic that contains multiple question/answer pairs. This design follows the general understanding that I've arrived at that making `<topic>` the primary unit of organization and granularity works well, even if it leads to topics that some people might initially or intuitively think were too small. But I wouldn't go to the mat to defend this design decision and won't claim it's necessarily the best. It has a logic I can defend but that's as far as I'll go.

As you work through the tutorial, take the time to ask yourself how you would have done it and why a different way would or wouldn't be better for some reason. This type of analytical thought is all part of understanding your requirements and mapping those requirements to implementations in order to ensure you have the most appropriate solution.

For the purposes of this tutorial, let us define an FAQ as a set of one or more question and answer pairs, where the question is a relatively short statement and the answer may be as short or long as needed. We would like the markup to reflect this essential nature, that is, there should be something named something like "question" and something named something like "answer". There are no particular requirements for the contents of answers themselves. We would like to be able to get a presentation where the question and answer are clearly identified, e.g., "Q. Question statement", "A. Answer response". Default topic presentation would *not* be sufficient in this case.

Note that these requirements are pretty simple. In any sort of engineering activity, it is best to start off as simply as you can and use iterative refinement to satisfy new requirements as you discover them. In the world of agile methods this is known as "the simplest thing that could possibly work".

This approach does several things: it lets you get something working quickly, it gives you immediate practical experience that will feed back into the design and implementation quickly, and it avoids designing and implementing things that you don't actually need. When designing XML markup it is quite easy to over-design and build complex markup structures that nobody actually wants or needs or, perhaps, can understand how to use. I've certainly done my share of this in the past. I now find it much more effective to start small and build up as needed. Often this refinement process all happens over the course of a few hours as I implement a new document type or specialization and start testing it with real data, sometimes it happens over weeks or months as the new

markup design is tested by its target users. In any case, for this tutorial, we will start small and, once we have something working that minimally meets our requirements, we can start thinking about other things we might need.

Another characteristic of agile development methods is "test-driven development", that is, the use of test cases to drive the implementation, rather than implementing first and testing later. The basic idea is that you write the test case first, which will of course initially fail (because there's no code yet) and then you do the implementation until the test case passes, at which point you know you're done. The test cases reflect the requirements as you understand them at the time you write the test case (and if the requirements change, you update the test case to reflect your new understanding).

For markup design, this translates into creating document instances and then implementing the DTD or schema that will validate those instances. When the instances are valid, you know you're done (as long as your instances reflect all the important cases the schema needs to support). This is as opposed to simply going from requirements straight to markup declarations and then only creating instances after the fact, which is the way we had to do it back in SGML days. One of the nice things about XML is that you can have documents with no document type, so you can start with instances and add DTDs or schemas later.

Names are always important and in this case there is a slight problem with the name to use for the topic element itself, namely "faq", which would be the obvious choice. The problem is that the term "FAQ" can be read as either singular or plural (a set of questions) but here we want a single topic to reflect a single question/answer pair. The name "question-and-answer" might work except that that could also imply more of a test-type question environment than an FAQ environment. Thus I have arrived at the name "faq-question"—it's technically redundant but fairly clear and not too long:

```
<faq-question id="q1">
</faq-question>
```

The topic title will be the question statement and it's probably useful to specialize `<title>` to `<faq-question-statement>` to make that clear:

```
<faq-question id="q1">
  <faq-question-statement>Can I add attributes to specific element types?</
faq-question-statement>
</faq-question>
```

In this case we don't have any particular requirements for the topic body content so we could leave `<body>` unspecialized, but since the body will be the FAQ answer, it makes sense to rename `<body>` to `<faq-answer>`.

Note that just "answer" is probably too generic—one challenge with DITA 1.x is that because you cannot use namespaces, all element types, including all specialized element types, must be unique. While you can't guarantee that your specialized types won't conflict with somebody else's, you should try to use names that are reasonably specific to your stuff. This can sometimes lead to names that are longer or more cumbersome than they would need to be if we could use namespaces in DITA 1.x. A good example is the element types defined by the DITA 1.2 Learning and Training vocabulary modules, which all start with "lc" (for "learning content"), which functions as a sort of "namespace prefix" and helps ensure that no other vocabularies will have names that collide with the Learning and Training types.

So our topic body should look like this:

```
<faq-question id="question-id">
  <faq-question-statement>Can I add attributes to specific element types?</
faq-question-statement>
  <faq-answer >
    <p>No, you can only define global attributes, specialized either from
&amp;lt;base&amp;gt;
    or @props.</p>
  </faq-answer>
</faq-question>
```

This design should be sufficient to get us going. Note that we are deferring all issues of how to organize the FAQ questions into FAQs using maps, where we know we already have everything we need to create sets of questions and to do things like group questions into titled groups.

Topic Specialization Step 2: Declare the Topic Module Element Types

The topic declarations go in a file called "*modulename*.mod", where "modulename" is usually the same as the tagname of the specialized topic element, this case f`faq-question.mod`.

To get started, the easiest thing to do is copy an existing topic module, either the one for the base you're specializing from, or one that is similar to the specialization you want to create.

Step 2-1. Create Test Case Document Instance

In your working area create a directory named "faq-question" and under that create the directory `dtd`. This will hold all the DTD-related materials for the topic specialization. You should create this directory outside the scope of the DITA Open Toolkit. (In the tutorial materials there are two directories, `faq-question-v1` and `faq-question`. The `faq-question-v1` directory contains the first iteration of the module, the `faq-question` directory contains the final, refined version.)

In the faq-question directory, create the file "faq-question-test-01.xml" with this content:

```
<?xml version="1.0"?>
<!DOCTYPE faq-question SYSTEM "dtd/faq-question.dtd">
<faq-question id="question-id">
  <faq-question-statement>Can I add attributes to specific element types?</
faq-question-statement>
  <faq-answer >
    <p>No, you can only define global attributes, specialized either from
&lt;base&gt;
    or @props.</p>
  </faq-answer>
</faq-question>
```

Note that the SYSTEM ID of the DTD is a local, relative URL. This is to keep things simple for testing purposes and avoid the added complexity of mapping PUBLIC or SYSTEM IDs through catalogs. Later we will set up the necessary catalog entries so documents can use an absolute URI for the faq-question shell DTD.

Open this document in an XML-aware editor. It should either fail to open with a "can't find the DTD" message or open but say that it can't be validated. Now our goal is to make this file validate.

Step 2-2. Create New Document Type Shell DTD

In the base DITA distribution, find the file "`concept.dtd`" and copy it into the `faq-question/dtd` directory and rename it "`faq-question.dtd`" (in the sample materials this file is named `faq-question-v1.dtd`).

Edit this file as follows:

- Delete the header comments and replace them with your own that indicates this document type shell DTD is owned by you and specific to the FAQ question module.
- Remove any references to any domain modules and entity files you don't need. Which ones you need is entirely your decision—if the FAQ is about a software product you may in fact need the software and UI domains. For simplicity you can leave all those declarations alone for now and worry about it later (or let the users of your module worry about it). In the tutorial materials I've used only the highlight and utility domains within this shell.
- Find the parameter entity named `%concept-info-types;` and rename it to `%faq-question-info-types;`. Set its replacement text to "no-topic-nesting", rather than "%info-types;".

  The `<no-topic-nesting>` element is one of the DITA "specialization" element types that exist to work around limitations in DTD syntax. In particular, if you have a content model declared like so:

  ```
  (%some-parameter-entity;)
  ```

You cannot simply set `%some-parameter-entity;` to an empty string, because the resolved result would be:

```
()
```

which is not valid.

The solution is to define an empty element type that acts as a placeholder, thus `<no-topic-nesting>`.

Try validating or opening the test document again. This time you should not get a "DTD not found" error but you should get "element type 'faq-question' not declared" sort of errors, indicating that it found the DTD but not the declarations for the FAQ-specific element types, which of course we haven't created yet. But we're making progress.

Step 2-3. Create New .mod File and Integrate Into Shell

Find the file "`concept.mod`" in the regular DITA DTD distribution and copy it to "`faq-question.mod`" in your `faq-questions/dtd` directory.

Edit this file as follows:

- Delete the header comments and replace them with your own that indicates this module DTD is owned by you and is the DTD declarations for the FAQ question module.

Now edit the `faq-question.dtd` file (the document type shell DTD) and integrate the module as follows:

1. Find the declaration for the `%concept-typemod;` entity declaration.
2. *After* this declaration, add this declaration and reference to the faq-question topic type module:

```
<!ENTITY % faq-question-typemod
  SYSTEM "faq-
question.mod"
 >
%faq-question-typemod;
```

The topic element integration section of the faq-question.dtd shell should now look like this:

```
<!-- ============================================================= -->
<!--                    TOPIC ELEMENT INTEGRATION                  -->
<!-- ============================================================= -->

<!--                    Embed topic to get generic elements        -->
<!ENTITY % topic-type   PUBLIC
"-//OASIS//ELEMENTS DITA Topic//EN"
"topic.mod"                                                         >
%topic-type;


<!--                    Embed concept to get specific elements     -->
<!ENTITY % concept-typemod
                        PUBLIC
"-//OASIS//ELEMENTS DITA Concept//EN"
"concept.mod"
>
%concept-typemod;

<!ENTITY % faq-question-typemod
  SYSTEM "faq-
question.mod"
 >
%faq-question-typemod;
```

Note also that I didn't bother to define either a public identifer or a URN for the `%faq-question-typemod;` parameter entity. This is to keep things simple, as with our test document. Later, once we've got everything working, we can replace the relative URL with a URN and set up the necessary catalog mappings.

Trying validating your test document again. Now you should get errors to the effect that the "concept" element type is declared multiple times, as well as the same messages about the FAQ-specific element types not being declared. This indicates that we've got the module integrated with the shell correctly.

Note: You may be now be thinking that I'm going about this in the wrong order—shouldn't I declare the element types for the module first and then integrate them? The method in this madness is that by starting from the back and working forward, we know both what isn't done yet (by the failures reported when we validate our test) and we'll know for sure when we're done (by the fact that the failures will go away) and, most important, we'll know that when the failures do go away, it's because everything really is hooked up correctly and not for some other hard-to-debug reason, like we forgot to hook in a particular module or something. Essentially we start with where we want to end up and then work backwards until we get there. If you see what I mean.

Step 2-4. Declare FAQ Question Topic Type Elements and Attributes

Per our markup design from step 1, we will need to declare the following specialized element types:

- `<faq-question>` as a specialization of `<concept>`
- `<faq-question-statement>` as a specialization of `<title>`
- `<faq-answer>` as a specialization of `<conbody>`

To create these new element types, edit the faq-question.mod file and modify it as follows:

1. Find the declaration for the `<concept>` element type. Change "concept" to "faq-question" in the element and attribute list declarations.
2. Within the content model for what is now `<faq-question>`, change "%title;" to "%faq-question-statement;" and "%conbody;" to "%faq-answer;".
3. Find the declaration for the `<conbody>` element type. Change "conbody" to "faq-answer" in the element and attribute list declarations.
4. From the base DITA distribution, open commenElements.mod and find the declaration for the `<title>` element. Copy this declaration and paste it into faq-question.mod. Change "title" to "faq-question-statement" in the element and attribute list declarations.

The element type declarations should look like this:

```
<!--                    LONG NAME: FAQ question  -->
<!ENTITY % faq-question.content
 "
  ((%faq-question-statement;),
   (%titlealts;)?,
   (%prolog;)?,
   (%faq-answer;)?,
   (%related-links;)?,
   (%faq-question-info-types;)* )
">
<!ENTITY % faq-question.attributes
'
             id         ID                                  #REQUIRED
             conref     CDATA                               #IMPLIED
             %select-atts;
             %localization-atts;
             %arch-atts;
             outputclass
                        CDATA                               #IMPLIED
             domains    CDATA                 "&included-domains;"
'>
<!ELEMENT faq-question %faq-question.content; >
<!ATTLIST faq-question %faq-question.attributes; >


<!--                    LONG NAME: FAQ answer details -->
<!ENTITY % faq-answer.content
```

```
"
  ((%body.cnt;)*,
   (%section;|
    %example;)*
  )
">
<!ENTITY % faq-answer.attributes
'
             %id-atts;
             %localization-atts;
             outputclass
                        CDATA                                #IMPLIED
'>
<!ELEMENT faq-answer %faq-answer.content; >
<!ATTLIST faq-answer %faq-answer.attributes; >

<!--                    LONG NAME: FAQ question statement -->

<!ENTITY % faq-question-statement.content
"
  (%title.cnt;)*
">
<!ENTITY % faq-question-statement.attributes
'
             %id-atts;
             %localization-atts;
             outputclass
                        CDATA                                #IMPLIED
'>
<!ELEMENT faq-question-statement %faq-question-statement.content; >
<!ATTLIST faq-question-statement %faq-question-statement.attributes; >
```

If you want, validate the test document again. You should now only get "parameter entity 'question-statement' was referenced but not declared" sorts of errors (and maybe some errors about the declaration of `<faq-question>` itself). This indicates we've declared the necessary element types, at least minimally, but we need to declare the parameter entities as well.

Step 2-5. Declare Element Type Parameter Entities

In `faq-question.mod`, find the comment labeled "ELEMENT NAME ENTITIES".

These parameter entities allow document type shell DTDs to extend the base content models that reference these element types by redeclaring these parameter entities to either include additional element types (e.g., specializations of the parameter entity's associated element type) or replace the original type with new types.

In this section of the file, make the following changes to the parameter entity declarations:

- Change "concept" to "faq-question" for both the parameter entity name and value.
- Change "conbody" to "faq-answer" for both the parameter entity name and value.
- Declare a new parameter entity `%faq-question-statement;` like so:

```
<!ENTITY % faq-question-statement "faq-question-statement" >
```

The element name entities section should now look like this:

```
<!-- ============================================================= -->
<!--                    ELEMENT NAME ENTITIES                      -->
<!-- ============================================================= -->


<!ENTITY % faq-question           "faq-question" >
<!ENTITY % faq-answer             "faq-answer"   >
```

```
<!ENTITY % faq-question-statement "faq-question-statement" >
```

Validate the test document again. The test document should validate.

If it does validate, then we know that the declarations are correct in the module, at least as far allowing our new element types go, and that the document type shell DTD is hooked up correctly.

At this point another test would be to see if our requirement that no nested topics, and in particular, no nested FAQ questions are allowed is satisfied. Only the `<no-topic-nesting>` element should be allowed.

Note that if you wanted to be really forceful about the requirement for no topic nesting, you could simply modify the content model of `<faq-question>` to remove the reference to `%faq-question-info-types;` entirely. That would be a statement that nested topics should *never* be allowed, while the `<no-topic-nesting>` element simply says "for this configuration, I don't want to allow topics". That is, the content model of `<faq-question>` as it is currently defined would allow users to allow nested topics via configuration of a document type shell DTD and would allow specializers to allow nested topics in a specialization of `<faq-question>`. If it was essential to the semantic of FAQ questions that nested topics *never* be allowed, it would be better to remove the reference to `%info-types;`. But that would probably be too constraining in this case. In general, when designing an element type that is general enough that it is likely to itself be a base for specialization, you should err on the side of laxity and let configurators and specializers add in the constraints they want.

While the declarations are now sufficient to allow us to create FAQ question instances, it's not yet complete because we haven't declared the DITA `class=` attributes that define the specialization hierarchies for the new specialized element types. This means that if we try to process our test document with the DITA Open Toolkit we won't get any output because the toolkit will not know that `<faq-question>` is actually a specialization of `<concept>`.

Step 2-6. Declare FAQ Question `class=` Attributes

In the `faq-question.mod` file, find the comment with the text "SPECIALIZATION ATTRIBUTE DECLARATIONS".

Edit the declarations within this section as follows:

- For the `<concept>` element's declaration, change the element type name to "faq-question". To the `class=` attribute value, add " faq-question/faq-question ".
- For the `<conbody>` element's declaration, change the element type name to "faq-answer" and add " faq-question/faq-answer " to the end of the `class=` attribute value.
- Copy one of the declarations and change the element type name to "faq-question-statement" and change the `class=` attribute value to "- topic/title faq-question/faq-question-statement " (remember the trailing space character at the end of the `class=` value).

The specialization attribute declarations section should now look like this:

```
<!-- ============================================================= -->
<!--                    SPECIALIZATION ATTRIBUTE DECLARATIONS      -->
<!-- ============================================================= -->

<!ATTLIST faq-question           %global-atts;  class CDATA "- topic/topic
concept/concept faq-question/faq-question ">
<!ATTLIST faq-answer             %global-atts;  class CDATA "- topic/body
concept/conbody faq-question/faq-answer ">
<!ATTLIST faq-question-statement %global-atts;  class CDATA "- topic/title
concept/title   faq-question/faq-question-statement ">
```

Validate the test document to check that you didn't introduce any syntax errors. You can also use your editor to test that the elements now allow `class=` attributes with the expected default values.

If you are using an editor like OxygenXML, as soon as you add the `class=` attributes to the module it should recognize the document as a DITA topic and automatically format it appropriately in the tags-off mode (Author mode in OxygenXML).

The `class=` attribute is sufficient for DITA processors to recoganize the elements as DITA elements, but the declaration is not quite complete. We still need to set up the `domains=` attribute for this specialization.

Step 2.7: Define `domains=` Attribute Contribution Entity (.ent File)

Both structural and domain modules require a .ent file in addition to the .mod file. For a structural domain (map or topic type) the .ent file simply defines the `domains=` attribute contribution for the topic type.

Create a file named `faq-question.ent` in the `faq-question/dtd` directory. Add a descriptive comment to the top of it indicating the name of the module and your ownership.

Add the following text entity declaration:

```
<!ENTITY faq-question-att
"(topic concept faq-question)"
>
```

The value of the entity is the list of the ancestor topic types for the new topic type, "topic concept" in this case, followed by the name of the topic type defined by the module itself. If the faq-question topic type required any domains it would list them following its own name.

Edit the `faq-question.dtd` file and find the comment "TOPIC ENTITY DECLARATIONS" and insert this parameter entity declaration and reference after the comment (if you don't find this comment, find the comment "DOMAIN ENTITY DECLARATIONS" and insert the following declaration *before* it):

```
<!ENTITY % faq-question-dec
  PUBLIC "faq-question.ent"
         "faq-question.ent"
>%faq-question-dec;
```

Note that the publid ID is just the filename. This is just for testing purposes. For production use you would use an appropriate URN or public ID.

Find the declaration for the `&included-domains;` text entity and add a reference to the entity `&faq-question-att;` to it. It should look like this:

```
<!ENTITY included-domains
                         "
                          &faq-question-att;
                          &hi-d-att;
                          &ut-d-att;
                         "
>
```

Validate the test document again. It should still be valid. Check the root `<faq-question>` element to verify that it has a `domains=` attribute and that its value includes the "(topic concept faq-question)" component in addition to those for the highlight and utility domains.

At this point the new topic module is complete. All that remains is to assign appropriate public identifiers to the .mod, .ent, and .dtd files and package it as an Open Toolkit plugin so that the module can be used for production.

Topic Specialization Step 3. Package the Modules as a Toolkit Plugin

Now that the declarations are complete and validated, we need to package the declarations for use in documents. The best way to do this is to package it as an Open Toolkit plugin. Making it into a plugin does two things:

- Provides entity resolution catalogs for mapping public identifiers to the various files.
- Makes it easy to deploy the vocabulary modules to Toolkit instances, which in turn makes the modules automatically available to tools that use the Toolkit to access vocabulary modules.

To set up the plugin, follow these steps:

1. In the `faq-question` directory create a file named `plugin.xml` with this content:

   ```
   <plugin id="org.example.faq-question.doctype">
     <feature extension="dita.specialization.catalog.relative"
   ```

```
value="catalog.xml" type="file"/>
</plugin>
```

Note: This defines a single plugin with just this module. In practice you would normally package all the related document type shells, vocabulary modules, and constraint modules together as a single Toolkit plugin. See *Packaging Document Type Shells and Vocabulary Modules as Toolkit Plugins* on page 20.

2. In the `faq-question` directory create a file named `catalog.xml` with this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
prefer="public">

  <nextCatalog catalog="dtd/catalog.xml"/>

</catalog>
```

3. In the `faq-question/dtd` directory create a file named `catalog.xml` with this content:

```
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
prefer="public">

  <public
publicId="urn:pubid:example.org:doctypes:dita:modules:entities:faq-
question"
    uri="faq-question.ent"
  />

  <public publicId="urn:pubid:example.org:doctypes:dita:modules:faq-
question"
    uri="faq-question.mod"
    />

  <public publicId="urn:pubid:example.org:doctypes:dita:faq-question"
    uri="faq-question.dtd"
    />

</catalog>
```

Note: In practice you would replace "example.org" with your own domain name or its universally-unique equivalent in the URNs.

4. Edit the file `faq-question/dtd/faq-question.dtd` and update the entity declarations for the .ent and .mod files to use the URNs defined in the catalog file:

```
...

<!ENTITY % faq-question-dec
  PUBLIC "urn:pubid:example.org:doctypes:dita:modules:entities:faq-
question-v1"
         "faq-question-v1.ent"
>%faq-question-dec;

...

<!ENTITY % faq-question-typemod
  PUBLIC "urn:pubid:example.org:doctypes:dita:modules:faq-question-v1"
"faq-question-
v1.mod"
 >
%faq-question-typemod;

...
```

Tip: Common mistake: Using the .mod file's URN for the .ent file or visa versa.

This is an easy mistake to make and a hard one to catch because the URNs are so similar. One symptom of this is that the shell that's in the same directory as the modules works, because it's system IDs are

resolvable, but a different shell that uses the same modules fails because the URNs are wrong and the system IDs are not resolvable. That is one reason to make sure that system IDs are not resolvable--it helps reveal this sort of error.

Making this mistake can lead to baffling validation errors from XML parsers, such as duplicate element type declarations or other DTD syntax errors. Whenever you get those the first thing to check is the public IDs of the module entity declarations in the shell document types.

5. Copy the test document to create a new test document, `faq-question-test-02.xml`, and modify its DOCTYPE declaration so it uses the URN of the .dtd file and uses an bogus system identifier (so the parser has to use the public ID to resolve the reference to the DTD file[8]):

```
<?xml version="1.0"?>
<!DOCTYPE faq-question
  PUBLIC "urn:pubid:example.org:doctypes:dita:modules:faq-question"
         "bogus"
>
<faq-question id="question-id">
  <faq-question-statement>Can I add attributes to specific element types?
</faq-question-statement>
  <faq-answer >
    <p>No, you can only define global attributes, specialized either
from @ase
    or @props.</p>
  </faq-answer>
</faq-question>
```

   Edit this document: it should not validate as you haven't deployed the plugin to your Open Toolkit yet.

6. Deploy the plugin to your Open Toolkit by copying the `faq-question` directory to the `plugins` directory of your Open Toolkit.

   Run the `integrator.xml` Ant task to integrate the plugin:

```
c:\DITA-OT > ant -f integrator.xml
```

   This has the effect of adding a reference to the `faq-question/catalog.xml` file into the Toolkit's master entity resolution catalog, `catalog-dita.xml`.

7. Configure your editor or a validation tool to use the catalog catalog-dita_template.xml and verify that the `faq-question-test-02.xml` document validates.

   If you are using OxygenXML and you deployed the plugin to Oxygen's Toolkit then you should just need perform the "Reset cache and validate" action. With other editors or validators you may need to explicitly configure the use of the `catalog-dita.xml` catalog.

Your new topic type is now ready to use.

Topic Specialization Step 4: Extending the Toolkit To Support the Specialization

For many topic specializations you won't require any change to the base Open Toolkit processing because you're using the specialization primarily to support authoring requirements and imposing editorial rules or to define specific reference structures or task structures that don't require any special processing. However, sometimes you want or need different or additional functionality, in which case you have to extend the Toolkit (or whatever tools you're using to process your DITA-based information).

There are two ways to extend a given transformation type in the Open Toolkit, "extension" and "override":

- Extension uses any extension points provided by the transformation type to hook in additional processing needed by a specific vocabulary module.
- Override involves creating a new transformation type that includes and overrides the base transformation type.

Which of these you use depends on what you need to accomplish for your vocabulary module.

[8] Pedantically, the reference to the external parameter entity declaration set.

If you are adding support for new element types or attributes, then you should use extension, since the processing for your new types cannot (or rather, should not) interfere with any base processing. For most vocabulary modules you will use extension, as is the case for the FAQ question topic type module.

If you need to implement processing that is specific to a particular publication type, user group within a larger organization, or otherwise limited to specific cases, then you should normally use overrides so you do not modify the the base processing for users of the base transformation type. For example, you might have different customizations of the PDF processor for different product groups within an enterprise. These customizations should be overrides not extensions.

If you need to implement processing that modifies the default processing but that should be used universally within a given organization, then you should use extensions. For example, you might have a corporate standard for specific HTML elements that should be reflected in any HTML output. In that case it would be appropriate to define an extension plugin that overrides the base processing.

Also, for HTML output, you can customize some presentation aspects through the use of custom CSS style sheets without the need to modify the HTML generation itself.

See *Configuring and Extending the DITA Open Toolkit* for a more complete discussion of extending and overriding Toolkit processing.

For the purposes of this tutorial we will extend the HTML generation to reflect the formatting specifications for FAQ Question.

For the FAQ Question specialization, the HTML formatting specifications are:

- The question statement (the topic title) should be introduced with "Q. " and the title itself should have a light pastel background.
- The answer should be introduced with a "A. " in the same size and font as the title.

There are several ways to achieve this sort of effect, including defining default values for the `output-class=` attribute in the DTDs (which would then let you style the HTML using a CSS style sheet without the need for a separate transform). However, for the purposes of this tutorial we will create a transform if for no other reason than to see how it's done.

As for creation of the DTDs themselves, the process of creating a specialization-specific XSLT script can be fairly mechanical and not require any great depth of XSLT knowledge, especially if all you're doing is tweaking the HTML output.

The basic process is as follows:

1. Create a directory to contain your Toolkit plugin, which will consist of the plugin descriptor file (`plugin.xml`) and the XSLT modules that implement your custom processing.
2. Create an empty XSLT stylesheet that will hold the templates specific to your specialization.
3. Create the plugin descriptor document that integrates your XSLT modules into the base transformation type.
4. Create match templates for each element type and context you'll need to handle
5. In the base Toolkit-supplied style sheets, find the template that would otherwise handle one of your specialized elements and copy its content and paste it into your local template.
6. Modify the local template to do whatever you need it to do
7. Repeat steps 4 through 6 for each specialized element type that needs different processing.

Topic Specialization Step 4-1: Create Initial Toolkit Plugin Components

Outside the toolkit, create the directory `org.example.faq-question.html` to contain your plugin. I use the convention of "*moduleName.transtype*" for the plugin name, thus "faq-question.html" for the HTML plugin, "faq-question.fo" for the PDF plugin, etc.

In the `org.example.faq-question.html` directory create the file `plugin.xml` with this content:

```
<!--
  Plugin descriptor for the FAQ question HTML extensions.
  -->
<plugin id="org.example.faq-question.html">
  <require plugin="org.example.faq-question.doctype"/>
```

```
  <feature extension="dita.xsl.xhtml" value="xsl/faq-question2html.xsl"
type="file"/>
</plugin>
```

This descriptor names the plugin ("org.example.faq-question.html"), indicates that it is dependent on the FAQ question document type plugin, and binds the plugin's XSLT module to the extension point "dita.xsl.xhtml", which is defined in the base DITA-to-HTML transformation type.

The `<require>` element is not strictly required but it makes it clearer that this module supports the FAQ question vocabulary module and is not, for example, a more general extension or override. Note also that the document type plugin does *not* state a dependency on the HTML plugin. This is because there may be users who want to use the vocabulary but not your particular implementation of the processing for it. Thus you should always have separate plugins for the vocabulary modules and their supporting processing so that users of your vocabulary modules can easily substitute their own processing if they want to.

Create the directory `xsl` within the `org.example.faq-question.html` directory. In the `xsl` directory create the file `faq-question2html.xsl` with this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs"
  version="2.0">

  <!-- ============================================================
    HTML generation templates for the FAQ Question DITA topic
    specialization.

    Copyright (c) 2010 Your Name Here

    ============================================================ -->


</xsl:stylesheet>
```

At the moment this stylesheet does nothing.

You can test the plugin by deploying it to your Toolkit and running the HTML transformation type against one of the FAQ question test documents. The processing should produce completely generic output but should not fail with any XSLT-related errors.

Note that this XSLT module is an XSLT 2 module. As of version 1.4.3 of the Open Toolkit, the Toolkit uses the Saxon XSLT engine exclusively. Saxon implements XSLT 2 so it is safe to use XSLT 2 with the Toolkit. It doesn't matter that the base transformation modules are XSLT 1 modules.

Topic Specialization Step 4-2: Create Templates For Specialized Elements

For the FAQ Question specialization, we need to modify the presentation of `<faq-question-statement>` and `<faq-answer>`. We don't need to do anything special for `<faq-question>` because we just want the normal topic processing. Thus we need match templates for `<faq-question-statement>` and `<faq-question>`. In addition, we will create a no-op template for `<faq-question>` just to demonstrate how to have a custom template and still use the base Toolkit-supplied processing, even though we don't need it, at least based on our current formatting specifications.

For DITA processing, match statements always have the same basic form, which is:

```
<xsl:template match="*[contains(@class, ' module/typename ')]">
```

Note the space characters before and after the module/typename pair.

This pattern for match expressions ensures that elements are always processed in terms of their specialization hierarchy and not their local tagnames. You can, of course, combine the "*[...]" matches together to match on elements in context, just as you would if you were using tagnames in the match statement.

For FAQ question create these three template instructions:

```
  <!-- ============================================================
    HTML generation templates for the FAQ Question DITA topic
    specialization.

    Copyright (c) 2010 Your Name Here

    ============================================================ -->

<xsl:template match="*[contains(@class, ' faq-question/faq-question ')]">
</xsl:template>

<xsl:template match="*[contains(@class, ' faq-question/faq-question-
statement ')]" >
</xsl:template>

<xsl:template match="*[contains(@class, ' faq-question/faq-answer ')]">
</xsl:template>

</xsl:stylesheet>
```

Topic Specialization Step 4-3: Copy Template Contents from Base XSLTs

In the Open Toolkit distribution, find the file `dita2htmlImpl.xsl`. It should be in the `xsl/xslhtml` directory.

Within this file, find the template that matches on " topic/title " within " topic/topic ". This can sometimes be a challenge because the indirect nature of the matches can make it hard to find the specific template you need. One technique is to search on " topic/title " and keep repeating the search until you find the template you're looking for. In this case, you should find this template:

```
<!-- NESTED TOPIC TITLES (sensitive to nesting depth, but are still
processed for contained markup) -->
<!-- 1st level - topic/title -->
<!-- Condensed topic title into single template without priorities; use
$headinglevel to set heading.
     If desired, somebody could pass in the value to manually set the
heading level -->
<xsl:template match="*[contains(@class,' topic/topic ')]/
*[contains(@class,' topic/title ')]">
  <xsl:param name="headinglevel">
      <xsl:choose>
          <xsl:when test="count(ancestor::*[contains(@class,' topic/topic
')]) > 6">6</xsl:when>
          <xsl:otherwise><xsl:value-of
select="count(ancestor::*[contains(@class,' topic/topic ')])"/></
xsl:otherwise>
      </xsl:choose>
  </xsl:param>
  <xsl:element name="h{$headinglevel}">
      <xsl:attribute name="class">topictitle<xsl:value-of
select="$headinglevel"/></xsl:attribute>
      <xsl:call-template name="commonattributes"/>
      <xsl:apply-templates/>
  </xsl:element>
  <xsl:value-of select="$newline"/>
</xsl:template>
```

Copy everything inside the `<xsl:template>` element and paste it into the template for `<faq-question-statement>` in faq-question2html.xsl:

```
<xsl:template match="*[contains(@class, ' faq-question/faq-question-
statement ')]">
```

```
  <xsl:param name="headinglevel">
      <xsl:choose>
          <xsl:when test="count(ancestor::*[contains(@class,' topic/topic
')]) > 6">6</xsl:when>
          <xsl:otherwise><xsl:value-of
select="count(ancestor::*[contains(@class,' topic/topic ')])"/></
xsl:otherwise>
      </xsl:choose>
  </xsl:param>
  <xsl:element name="h{$headinglevel}">
      <xsl:attribute name="class">topictitle<xsl:value-of
select="$headinglevel"/></xsl:attribute>
      <xsl:call-template name="commonattributes"/>
      <xsl:apply-templates/>
  </xsl:element>
  <xsl:value-of select="$newline"/>
</xsl:template>
```

For <faq-answer> we want the processing applied to concept/conbody. However, if you search for " concept/conbody " you won't find anything, which means that there is no special processing for <conbody>. That means you must search for the next level up the specialization hierarchy, namely " topic/body ". Copy the contents of that template into the template for <faq-answer>:

```
<xsl:template match="*[contains(@class, ' faq-question/faq-answer ')]">
  <xsl:variable name="flagrules">
    <xsl:call-template name="getrules"/>
  </xsl:variable>
<div>
  <xsl:call-template name="commonattributes"/>
  <xsl:call-template name="gen-style">
    <xsl:with-param name="flagrules" select="$flagrules"></xsl:with-param>
  </xsl:call-template>
  <xsl:call-template name="setidaname"/>
  <xsl:call-template name="start-flagit">
    <xsl:with-param name="flagrules" select="$flagrules"></xsl:with-
param>
  </xsl:call-template>
  <xsl:call-template name="start-revflag">
    <xsl:with-param name="flagrules" select="$flagrules"/>
  </xsl:call-template>
  <!-- here, you can generate a toc based on what's a child of body -->
  <!--xsl:call-template name="gen-sect-ptoc"/--><!-- Works; not always
wanted, though; could add a param to enable it.-->

  <!-- Insert prev/next links. since they need to be scoped by who they're
'pooled' with, apply-templates in 'hierarchylink' mode to linkpools (or
related-links itself) when they have children that have any of the
following characteristics:
       - role=ancestor (used for breadcrumb)
       - role=next or role=previous (used for left-arrow and right-arrow
before the breadcrumb)
       - importance=required AND no role, or role=sibling or role=friend
or role=previous or role=cousin (to generate prerequisite links)
       - we can't just assume that links with importance=required are
prerequisites, since a topic with eg role='next' might be required, while
at the same time by definition not a prerequisite -->

  <!-- Added for DITA 1.1 "Shortdesc proposal" -->
  <!-- get the abstract para -->
  <xsl:apply-templates select="preceding-sibling::*[contains(@class,'
topic/abstract ')]" mode="outofline"/>
```

```
  <!-- get the shortdesc para -->
  <xsl:apply-templates select="preceding-sibling::*[contains(@class,'
topic/shortdesc ')]" mode="outofline"/>

  <!-- Insert pre-req links - after shortdesc - unless there is a prereq
section about -->
  <xsl:apply-templates select="following-sibling::*[contains(@class,'
topic/related-links ')]" mode="prereqs"/>

  <xsl:apply-templates/>
  <xsl:call-template name="end-revflag">
    <xsl:with-param name="flagrules" select="$flagrules"/>
  </xsl:call-template>
  <xsl:call-template name="end-flagit">
    <xsl:with-param name="flagrules" select="$flagrules"></xsl:with-param>
  </xsl:call-template>
</div><xsl:value-of select="$newline"/>

</xsl:template>
```

For <faq-question>, add a next-match statement to the template body:

```
<xsl:template match="*[contains(@class, ' faq-question/faq-question ')]">
  <xsl:next-match/>
</xsl:template>
```

The next-match simply says "take the context node (that is, the <faq-question> element) and apply whatever template would have matched if this template hadn't matched. This has the effect of just applying the default formatting for concept topics but gives us a ready-made place to add new or different processing should we need it. It also demonstrates the use of next-match.

If you redeploy the plugin and apply the Toolkit to your FAQ topics, you should again get the normal output. You can verify that you are really using your stylesheet by either introducing a syntax error and verifying that the processing fails, or by adding something to a template that will have a visual effect in the output.

Once you have verified that everything is working as expected, you are ready to start modifying the processing.

Topic Specialization Step 4-4: Implement Specialization-Specific XSLT Processing

For the FAQ question, the new processing is relatively easy: just generate "Q. " before the title content:

```
<xsl:template match="*[contains(@class, ' faq-question/faq-question-
statement ')]">
  <xsl:param name="headinglevel">
      <xsl:choose>
          <xsl:when test="count(ancestor::*[contains(@class,' topic/topic
')]) > 6">6</xsl:when>
          <xsl:otherwise><xsl:value-of
select="count(ancestor::*[contains(@class,' topic/topic ')])"/></
xsl:otherwise>
      </xsl:choose>
  </xsl:param>
  <xsl:element name="h{$headinglevel}">
    <xsl:attribute name="class">topictitle<xsl:value-of
select="$headinglevel"/></xsl:attribute>
    <xsl:call-template name="commonattributes"/>
    <div class="faq-question-statement" style="background-color: #FFFFA0">
    <span class="faq-question-statement-q">Q. </span>
    <xsl:apply-templates/>
    </div>
  </xsl:element>
  <xsl:value-of select="$newline"/>  <xsl:param name="headinglevel">
```

```
</xsl:template>
```

The with the class= value isn't strictly necessary but it provides a handy hook for using CSS styles to further control the presentation of the question.

For the question answer, the solution is not quite so obvious. In order to produce the answer with the initial "A. " text such that it is immediately followed by the text of the first paragraph requires processing the first child paragraph of <faq-answer> specially and then processing the rest of the content. However, the base template provides processing for elements that would be presented *before* the first paragraph, such as the abstract and the short description.

This presents a dilemma and indicates that we haven't fully thought through the markup design or the presentation design. Clearly we were thinking of questions as just being a title with the question and answers as paragraphs. But DITA topics can be more sophisticated. The upshot is that we have to account for these elements in some way.

One way would be to simply eliminate them from the allowed content of <faq-question>. That would certainly simplify the problem but might make our FAQ question topics too simple. For example, there might be systems that depend on short descriptions or abstracts for some cool functionality.

The better thing would be to provide FAQ-specific processing for these elements that ensures the correct presentation. In addition, there might be ways to integrate these elements with the FAQ to make them more useful.

In particular, it probably makes sense to refine the model for <faq-question> to *require* <shortdesc> to be the first paragraph of the answer, with the body as the rest of the answer. This would enable, for example, an FAQ presentation that shows just the question and short description with the topic body hidden until requested.

This point that we've come to is one of the reasons that it's so important to test new markup designs in the context of realistic processing as well as in the context of authoring. As you design new markup you should expect to go through several iterations of design/implement/rework. One advantage of using DITA as a base is that it makes it fairly inexpensive to do this iterative development because you can quickly extend the base functionality rather than having to first implement a large base of processing just to get to a point where you can see some output.

At this point, we put our XSLT work on hold for a moment and go back to the DTD declarations to refine the markup rules.

Topic Specialization Step 4-5: Refine Markup Design for FAQ Question

As a result of our initial output processing implementation effort, we've realized that we should be requiring the <shortdesc> element to act as the required first paragraph of a potentially longer answer. We will also disallow the <abstract> element as it's not really relevant to what we're trying to do.

This change also requires us to rethink our element type names. If the short description is going to act as the first paragraph of the answer, then "shortdesc" is probably not the best name. By the same token, the name "faq-answer" is probably not the best name for the topic body. Better names are probably "faq-short-answer" for the short description and "faq-answer-details" for the topic body.

To implement this new design, first modify the test document to use this new markup:

```
<faq-question id="question-id">
  <faq-question-statement>Can I add attributes to specific element types?</
faq-question-statement>
  <faq-short-answer>No, you can only define global attributes, specialized
either from @base
    or @props.</faq-short-answer>
  <faq-answer-details>
    <p>As of DITA 1.1, there is no mechanism for defining new attributes
for specific element types.
    The problem is that there is no DITA-defined syntax for declaring a
given attribute as being
    an attribute that should be recognized as such by a DITA-aware
```

```
processor and not just some
    random attribute that was allowed in error (as the DITA architecture
otherwise does not
    allow non-DITA-defined attributes.</p>
    <p>Providing this ability is a booked feature request for DITA.</p>
  </faq-answer-details>
</faq-question>
```

Verify that the document is no longer valid.

Edit faq-question.mod and do the following:

1. Modify the content model for <faq-question> as follows:
   a. Remove "| %abstract;"
   b. Change "%shortdesc;" to "%faq-short-answer;"
   c. Remove the "?" following the group containing "%faq-short-answer;" to make it required.
   d. Change "%faq-answer;" to "%faq-answer-details;"

   The resulting declaration should look like this:

```
<!ENTITY % faq-question.content
 "
  ((%faq-question-statement;),
   (%titlealts;)?,
   (%faq-short-answer;),
   (%prolog;)?,
   (%faq-answer-details;)?,
   (%related-links;)?,
   (%faq-question-info-types;)* )
">
```

2. Find the declaration of the %faq-answer; parameter entity and change it's name and value to "faq-answer-details"
3. Create a new element name parameter entity declaration for <faq-answer-details>.
4. Find the element and attribute declarations for <faq-answer> and change the element type name to "faq-answer-details".

   The new declarations should look like this:

```
<!--                    LONG NAME: FAQ answer details -->
<!ENTITY % faq-answer-details.content
"
  ((%body.cnt;)*,
   (%section;|%example;)*
  )
">
<!ENTITY % faq-answer-details.attributes
'
             %id-atts;
             %localization-atts;
             outputclass
                        CDATA                              #IMPLIED
'>
<!ELEMENT faq-answer-details %faq-answer-details.content; >
<!ATTLIST faq-answer-details %faq-answer-details.attributes; >
```

5. In the DITA-supplied commonElements.mod file, find the element type declaration for <shortdesc>. Copy it and paste it after the element type and attribute declaration for <faq-question>. (I like to list element type declarations more or less in the order they occur in the document's structural hierarchy.)
6. Change "shortdesc" to "faq-short-answer" in the declarations you just copied.

   The resulting declaration should look like this:

```
<!--                    LONG NAME: FAQ short answer                -->
<!ENTITY % faq-short-answer.content
"
```

```
  (%title.cnt;)*
">
<!ENTITY % faq-short-answer.attributes
'  %univ-atts;
   outputclass
      CDATA #IMPLIED
'>
<!ELEMENT faq-short-answer %faq-short-answer.content; >
<!ATTLIST faq-short-answer %faq-short-answer.attributes; >
```

7. Find the `class=` attribute declarations at the end of the .mod file and edit them as follows:
   a. In the attribute declaration for `<faq-answer>`, change both occurrences of "faq-answer" to "faq-answer-details".
   b. Copy the declaration for `<faq-answer-details>` and change "faq-answer-details" to "faq-short-answer".
   c. In the `class=` attribute declaration for what is now `<faq-short-answer>`, change "topic/body" to "topic/shortdesc". Change " concept/conbody " to "concept/shortdesc" (all the levels of the specialization hierarchy must be accounted for in the `class=` attributes even if the element type name is not changed).

The class declarations should now look like this:

```
<!ATTLIST faq-question        %global-atts;  class CDATA "- topic/
topic     concept/concept   faq-question/faq-question ">
<!ATTLIST faq-short-answer    %global-atts;  class CDATA "- topic/
shortdesc concept/shortdesc faq-question/faq-short-answer ">
<!ATTLIST faq-answer-details  %global-atts;  class CDATA "- topic/
body      concept/conbody   faq-question/faq-answer-details ">
<!ATTLIST question-statement  %global-atts;  class CDATA "- topic/
title     concept/title     faq-question/faq-question-statement ">
```

Redeploy the faq-question Toolkit plugin and try validating the updated test document. It should validate.

Topic Specialization Step 4-4 (continued): Implement Specialization-Specific XSLT Processing

Having reworked the markup design to provide a short answer and answer details, pop back up to step 4-4 and continue implementing the XSLT processing.

The first thing to do is update the existing templates to reflect our new element type names, which means changing the template for `<faq-answer>` to match on "faq-answer-details" instead of "faq-answer":

```
  <xsl:template match="*[contains(@class, ' faq-question/faq-answer-
details ')]">
```

Now the challenge is to modify the code within this template to give us the result we want. Modify the template as follows:

1. Delete all the comments between the call-template to "start-revtag" and the apply-templates for "topic/abstract".
2. Because we have removed `<abstract>` from the content model for `<faq-question>`, you can delete the apply-templates instruction that matches on "topic/abstract". Having the instruction there wouldn't hurt anything (there could never be an abstract to match on), but it might confuse somebody who knows the content model for `<faq-question>`. However, since somebody who knows the content model for topic might expect there to be a match on `<abstract>`, it's probably a good idea to provide a comment to the effect that `<faq-question>` doesn't allow `<abstract>`.
3. Put a `<div>` with a `class=` attribute value of "faq-answer-details" around the last "apply-templates" call:

   ```
   <div class="faq-answer-details">
     <xsl:apply-templates/>
   </div>
   ```

   This `<div>` clearly separates the answer details from the short answer and the `class=` value provides a hook that can be used in CSS styles or JavaScript to add additional formatting or behavior to the answer details.

The reworked template should look like this (changed bits are shown in bold):

```
  <xsl:template match="*[contains(@class, ' faq-question/faq-answer-
details ')]">
    <xsl:variable name="flagrules">
      <xsl:call-template name="getrules"/>
    </xsl:variable>
    <div>
      <xsl:call-template name="commonattributes"/>
      <xsl:call-template name="gen-style">
        <xsl:with-param name="flagrules" select="$flagrules"></xsl:with-
param>
      </xsl:call-template>
      <xsl:call-template name="setidaname"/>
      <xsl:call-template name="start-flagit">
        <xsl:with-param name="flagrules" select="$flagrules"></xsl:with-
param>
      </xsl:call-template>
      <xsl:call-template name="start-revflag">
        <xsl:with-param name="flagrules" select="$flagrules"/>
      </xsl:call-template>
      <!-- Abstract not allowed in faq-question topic type -->
      <!-- get the shortdesc para -->
      <xsl:apply-templates select="preceding-sibling::*[contains(@class,'
topic/shortdesc ')]" mode="outofline"/>

      <!-- Insert pre-req links - after shortdesc - unless there is a
prereq section about -->
      <xsl:apply-templates select="following-sibling::*[contains(@class,'
topic/related-links ')]" mode="prereqs"/>

      <div class="faq-answer-details">
        <xsl:apply-templates/>
      </div>
      <xsl:call-template name="end-revflag">
        <xsl:with-param name="flagrules" select="$flagrules"/>
      </xsl:call-template>
      <xsl:call-template name="end-flagit">
        <xsl:with-param name="flagrules" select="$flagrules"></xsl:with-
param>
      </xsl:call-template>
    </div><xsl:value-of select="$newline"/>
  </xsl:template>
```

Without having tested it yet, the template now looks like it should be right: it's not getting the abstract, it is processing the short description (our new `<faq-short-answer>`) and the rest of the processing should be appropriate.

Redeploy the Toolkit plugin and run the transform against your sample FAQ question. You should get the same output you got before since all we really did was change the template match value and cut away stuff we didn't need.

The next step in the implementation is to handle the `<faq-short-answer>` element.

The formatting specification says to output "A. " followed by the the answer text. To do this we need a template that matches on " faq-question/faq-short-answer ". Note that even though `<faq-short-answer>` is specialized from `<shortdesc>`, it would not make sense to simply override the processing for `<shortdesc>`, as that would change the formatting for all other topic types.

Create the following template:

```
  <xsl:template match="*[contains(@class, ' faq-question/faq-short-answer
')]" mode="outofline">
    <div class="faq-short-answer">
```

```
      <span class="faq-short-answer-a">A. </span>
      <xsl:apply-templates/>
    </div>
  </xsl:template>
```

Note the `mode=` attribute with the value "outofline". This is necessary because the short description is being processed out of its normal source order, that is from within the processing for the topic body (to which shortdesc is a sibling) rather than from with the processing of topic (of which it is a child). The base DITA style sheets have been set up to handle this in a generic way using the mode "outofline" for all templates that handle elements processed out of their normal document order. You can see that the mode is used on the apply templates in the template for `<faq-question>`:

```
  <xsl:template match="*[contains(@class, ' faq-question/faq-answer-
details ')]" priority="10">
    <div>

      ...

      <!-- get the shortdesc para -->
      <xsl:apply-templates select="preceding-sibling::*[contains(@class,'
topic/shortdesc ')]"
        mode="outofline"/>

      ...

    </div>
    <xsl:value-of select="$newline"/>
  </xsl:template>
```

If you didn't specify the corresponding mode on the match template for `<faq-short-answer>`, you would get the answer text twice, once from base outofline mode template that matches on " topic/shortdesc " and once for the default mode template that matches on " faq-question/faq-short-answer ".

This template will put out "A. " before the short description. The `class=` values are there to support styling the result using CSS.

Redeploy the Toolkit plugin again and run the sample FAQ answer through the Toolkit and inspect the results. You should now see the "A. " before the short answer text (the first paragraph of the result). This means that the XSLT is done. The next step is to create or update the applicable CSS style sheet to add the appropriate styling to the HTML pages.

Topic Specialization Step 4-6: Implement CSS Styles

The DITA Open Toolkit provides a general framework for using CSS style sheets with the HTML generated from topics. It provides the "commonltr.css" and "commonrtl.css" styles, which are used by default for all generated HTML. You can also supply your own custom CSS and specify it as part of the HTML generation process.

In the tutorial materials there is "faq-question_html.css" in the "html/css" directory. For your own environment you can create a new CSS style sheet. You should store and manage it outside the scope of the DITA Open Toolkit. The Toolkit processing can automatically copy it to the output location for you. Or, if you are publishing your topics to a larger Web site, the CSS styles may be managed separately, in which case you just specify the name and path of the CSS but turn off the copying. This is all explained reasonably well in the Open Toolkit documentation.

For this tutorial, create a new CSS file called "faq-question.css" and give it this initial content:

```
/*

  CSS for FAQ questions

  Copyright (c) 2010 Your Name Here
```

```
 */
```

To use your custom CSS stylesheet you supply the full path of the CSS file, wherever you've put it, with the "/css" parameter of the HTML transformation type and specify "/csscopy" as "yes".

The main thing this CSS needs to do is make the "A. " before the short answer big and bold to match the "Q. " for the question statement. Do this like so:

```
*[class = 'faq-short-answer-a'] {
  font-family: sans-serif;
  font-size: 16pt;
  font-weight: bold;
}
```

You can test this by either re-running the toolkit or, once you've run it once so that the CSS stylesheet is referenced from the HTML, you can just manually copy the changed CSS to the output directory and see how it looks. You might also find it easiest to develop the CSS style in place and then once you're happy with the result, copy it back to the original location. Just be careful not to run the toolkit in the meantime or you'll overwrite your changes (doh!).

This style specification gives the result that was specified but it doesn't look quite right: the "A. " is now too big relative to the text of the answer. I don't pretend to have any skill at graphic design but my response to this is to make the short answer text bigger:

```
*[class = 'faq-short-answer'] {
  font-family: sans-serif;
  font-size: 14pt;
}
```

Note that the style is using the `class=` value on the `<div>` that you put around the short answer.

Test your styles until you're satisfied with the visual result. If you want to get really sophisticated, you can add JavaScript to show or hide the answer details, but that's beyond the scope of this tutorial.

That's it, you're done.

Topic Specialization: XSD Version

The process for creating an XSD version of the FAQ question topic type is as follows (assuming you have already created the final DTD version of the faq-question topic type):

1. Create the directory `faq-question/xsd` in your working area.
2. Copy the files `concept.xsd`, `conceptGrp.xsd`, and `conceptMod.xsd` from the base DITA schema set to the `faq-question/xsd` directory.
3. Rename the files from "concept*" to "faq-question*", resulting in `faq-question.xsd`, `faq-questionGrp.xsd`, and `faq-questionMod.xsd`.
4. Copy the `catalog.xml` file from the `dtd` directory into the `xsd` directory.
5. Edit `xsd/catalog.xml` and change it as follows:
   a. Change ".dtd" to ".xsd"
   b. Change ".mod" to "Mod.xsd"
   c. Change ".ent" to "Grp.xsd"
   d. In the public identifiers, append the corresponding last part of the filename to the end of the URN, e.g. "urn:pubid:example.org:doctypes:dita:modules:faq-question" becomes "urn:pubid:example.org:doctypes:dita:modules:faq-questionMod.xsd"
   e. Copy all the `<public>` elements and change "public" to "uri" and the "publicId" attributes to "name".

      This is necessary because the Apache Xerces parser incorrectly uses public ID resolution to resolve XSD schema locations. It should use URI entries (because schema locations are URIs not entity references and

therefore should be resolved through URI entries in catalogs). Other systems that use the catalog will likely do the correct thing and use URI entries to resolve schema locations. So you need both forms of entry.

If you want to cover all bases you can make yet another copy of the entries and change `<public>` to `<system>` and "publicId" to "systemId". That covers the case where a processor treats a schema location as a system ID rather than a URI. That would also be wrong but some systems may do it.

6. Edit the `faq-question/catalog.xml` file, copy the `<nextCatalog>` element and change "dtd" to "xsd" in the new copy, resulting in this catalog file:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
prefer="public">

  <nextCatalog catalog="dtd/catalog.xml"/>
  <nextCatalog catalog="xsd/catalog.xml"/>

</catalog>
```

7. Copy one of the faq-question test documents to use for testing the XSD document type shell. Modify it as follows:
   a. Delete the DOCTYPE declaration from the top of the file.
   b. Add the following attributes to the `<faq-question>` element:

```
<?xml version="1.0"?>
<faq-question
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="xsd/faq-question.xsd"
  id="question-id">

  ...

</faq-question>
```

   Where the value of the `xsi:noNamespaceSchemaLocation=` reflects the appropriate relative URL to the `faq-question.xsd` file.

   The document should not be valid at this point because the `faq-question.xsd` file is just an unmodified copy of the concept XSD shell.

8. Edit `faq-question.xsd` and modify it as follows:
   a. Replace the header comment with something that reflects your ownership.
   b. Remove the domain module inclusions for all but the hightlight and utility domains, leaving just these declarations:

```
  <!--  ================ TOPIC DOMAINS INLCUSION
====================  -->
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:highlightDomain.xsd:1.2"/>
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:utilitiesDomain.xsd:1.2"/>

  <!--  ================ CONCEPT GROUP
DEFINITION====================  -->
```

   c. After the "CONCEPT GROUP DEFINITION" comment, add a reference to the `faq-questionGrp.xsd` file:

```
  <!--  ================ CONCEPT GROUP
DEFINITION====================  -->
  <xs:include schemaLocation="faq-questionGrp.xsd"/>
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:conceptGrp.xsd:1.2"/>
```

   d. From the `<xs:redefine>` element, delete all the groups except the groups for "ph" and "fig":

```
  <xs:redefine
schemaLocation="urn:oasis:names:tc:dita:xsd:commonElementGrp.xsd:1.2">
```

```
    <xs:group name="ph">
      <xs:choice>
        <xs:group ref="ph"/>
        <xs:group ref="pr-d-ph" />
        <xs:group ref="ui-d-ph" />
        <xs:group ref="hi-d-ph" />
        <xs:group ref="sw-d-ph" />
      </xs:choice>
    </xs:group>

    <xs:group name="fig">
      <xs:choice>
        <xs:group ref="fig"/>
        <xs:group ref="pr-d-fig"/>
        <xs:group ref="ut-d-fig" />
      </xs:choice>
    </xs:group >

  </xs:redefine>
```

e. From the group for "ph", delete the group references for pr-d-ph, ui-d-ph, and sw-d-ph, leaving this group definition:

```
    <xs:group name="ph">
      <xs:choice>
        <xs:group ref="ph"/>
        <xs:group ref="hi-d-ph" />
      </xs:choice>
    </xs:group>
```

f. From the group for "fig" delete the group reference for pr-d-fig, leaving this group definition:

```
    <xs:group name="fig">
      <xs:choice>
        <xs:group ref="fig"/>
        <xs:group ref="ut-d-fig" />
      </xs:choice>
    </xs:group >
```

g. After the include for the conceptMod.xsd file, add an include of the faq-questionMod.xsd file:

```
  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:topicMod.xsd:
1.2" />
  <xs:include
schemaLocation="urn:oasis:names:tc:dita:xsd:conceptMod.xsd:1.2" />
  <xs:include schemaLocation="faq-questionMod.xsd" />
```

h. Modify the default value of the `domains=` attribute to reflect the domains and topic types actually used:

```
  <xs:attributeGroup name="domains-att">
    <xs:attribute name="domains" type="xs:string"
      default="(topic hi-d) (topic ut-d) (topic concept faq-question)"/
>
  </xs:attributeGroup>
```

At this point, the test document is still not valid. When you validate it you should see messages about the schema containing two occurrences of global components (concept, conbody, etc.), since the faq-questionMod.xsd file is still just a copy of the conceptMod.xsd file.

9. Edit `faq-questionGrp.xsd` and modify it as follows:

a. Replace the header comment with one that reflects your ownership.

b. Change "concept" to "faq-question".

c. Change "conbody" to "faq-answer-details".

d. Make two copies of one of the groups to create new groups for faq-short-answer and faq-question-statement

The resulting XSD file should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ============================================================
```

```
    FAQ Question topic type element-type group definitions

    Copyright (c) 2010 Your Name Here
    ============================================================= -->

<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>
  <xs:group name="faq-question">
      <xs:sequence>
         <xs:choice>
            <xs:element ref="faq-question"/>
         </xs:choice>
      </xs:sequence>
   </xs:group>

  <xs:group name="faq-answer-details">
      <xs:sequence>
         <xs:choice>
           <xs:element ref="faq-answer-details"/>
         </xs:choice>
      </xs:sequence>
   </xs:group>

  <xs:group name="faq-short-answer">
    <xs:sequence>
      <xs:choice>
        <xs:element ref="faq-short-answer"/>
      </xs:choice>
    </xs:sequence>
  </xs:group>

  <xs:group name="faq-question-statement">
    <xs:sequence>
      <xs:choice>
        <xs:element ref="faq-question-statement"/>
      </xs:choice>
    </xs:sequence>
  </xs:group>

</xs:schema>
```

Validate the test document again. This time you should get messages about being unable to resolve the references to the FAQ-specific element types referenced from the groups you just created.

10. Edit `faq-questionMod.xsd` and modify it as follows:

a. Replace the header comment with one that reflects your ownership.

b. Modify the `<xs:annotation>` element to add " faq-question" to the domains value string:

```
   <xs:annotation>
    <xs:appinfo>
      <dita:domainsModule xmlns:dita="http://dita.oasis-open.org/
architecture/2005/"
        >(topic concept faq-question)</dita:domainsModule>
    </xs:appinfo>
    <xs:documentation>

    </xs:documentation>
  </xs:annotation>
```

c. Make the global change "concept." to "faq-question." (note the trailing ".", very important) and "conbody." to "faq-answer-details.".

This change changes all the building blocks for `<concept>` and `<conbody>` to their FAQ question equivalents. The trailing "." is essential because you do not want to change "concept" or "conbody" where it occurs in `class=` attribute values.

d. In the element type declaration for "concept", change `name="concept"` to `name="faq-question"`.

e. In the value of the `class=` attribute, append " faq-question/faq-question " to the end of the default value.

f. Update the documentation to reflect the semantics for `<faq-question>`.

The resulting declaration should look like this:

```
  <xs:element name="faq-question">
    <xs:annotation>
      <xs:documentation>
        The &lt;<keyword>faq-question</keyword>&gt; element is the top-
level
        element for a topic that represents a single question/answer
pair representing
        a single question in a set of frequently asked questions.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="faq-question.class">
          <xs:attribute ref="class" default="- topic/topic concept/
concept faq-question/faq-question "/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
```

g. Modify the `<xs:element>` for "conbody" to reflect the `<faq-answer-details>`, resulting in this element declaration:

```
  <xs:element name="faq-answer-details">
    <xs:annotation>
      <xs:documentation>
        The &lt;<keyword>faq-answer-details</keyword>&gt; element is
the main body-level
        element for an faq-question. It holds any additional details
for the FAQ question.
        Note that the first paragraph of the question answer is always
held in the
        the <keyword>faq-short-answer</keyword> element.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="faq-answer-details.class">
          <xs:attribute ref="class"
            default="- topic/body concept/conbody faq-question/faq-
answer-details "/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
```

h. Change the group "concept-info-types" to "faq-question-info-types", removing the reference to the "concept" group:

```
  <xs:group name="faq-question-info-types">
    <xs:choice>
      <xs:group ref="info-types" />
      <!-- Removed reference to concept topic type -->
    </xs:choice>
  </xs:group>
```

i. Find the group "faq-question.content" (created by the global change you made earlier) and modify it to reflect the content model for `<faq-question>`:

```
   <xs:group name="faq-question.content">
     <xs:sequence>
```

```
       <xs:sequence>
          <xs:group ref="faq-question-statement"/>
          <xs:group ref="titlealts" minOccurs="0"/>
          <xs:choice minOccurs="1">
            <xs:group ref="faq-short-answer" />
            <!-- Removed reference to abstract -->
          </xs:choice>
          <xs:group ref="prolog" minOccurs="0"/>
          <xs:group ref="faq-answer-details" minOccurs="0"/>
          <xs:group ref="related-links" minOccurs="0"/>
          <xs:group ref="faq-question-info-types" minOccurs="0"
maxOccurs="unbounded"/>
        </xs:sequence>
     </xs:sequence>
   </xs:group>
```

j. Copy the `<xs:element>` for "faq-answer-details" and rename it "faq-question-statement". Change "faq-answer-details" to "faq-question-statement" and change the `domains=` attribute value to "- topic/title concept/title faq-question/faq-question-statement ":

```
   <xs:element name="faq-question-statement">
     <xs:annotation>
       <xs:documentation>
         The &lt;<keyword>faq-question-statement</keyword>&gt; element
holds
         the question part of the question/answer pair (it is the FAQ
topic's
         title).
       </xs:documentation>
     </xs:annotation>
     <xs:complexType>
       <xs:complexContent>
         <xs:extension base="faq-question-statement.class">
           <xs:attribute ref="class"
             default="- topic/title concept/title faq-question/faq-
question-statement "/>
         </xs:extension>
       </xs:complexContent>
     </xs:complexType>
   </xs:element>
```

k. Copy the `<xs:element>` for "faq-answer-details" and rename it "faq-short-answer". Change "faq-answer-details" to "faq-short-answer" and change the `domains=` attribute value to "- topic/shortdesc concept/shortdesc faq-question/faq-short-answer ":

```
   <xs:element name="faq-short-answer">
     <xs:annotation>
       <xs:documentation>
         The &lt;<keyword>faq-short-answer</keyword>&gt; element holds
         the first or only paragraph of the answer. This is the short
description
         for the topic.
       </xs:documentation>
     </xs:annotation>
     <xs:complexType>
       <xs:complexContent>
         <xs:extension base="faq-short-answer.class">
           <xs:attribute ref="class"
             default="- topic/shortdesc concept/shortdesc faq-question/
faq-short-answer "/>
         </xs:extension>
       </xs:complexContent>
     </xs:complexType>
   </xs:element>
```

l. Open the file `commonElementMod.xsd` from the standard DITA schema distribution and find the `<xs:complexType>` declaration for "title.class". Copy the complexType, title.content group, and title.attributes group and paste it into the faq-questionMod.xsd.

m. Change "title." to "faq-question-statement." in the newly-pasted components except for the reference to "title.cnt". The resulting declarations should be:

```
<xs:complexType name="faq-question-statement.class" mixed="true">
  <xs:sequence>
    <xs:group ref="faq-question-statement.content"/>
  </xs:sequence>
  <xs:attributeGroup ref="faq-question-statement.attributes"/>
</xs:complexType>
<xs:group name="faq-question-statement.content">
  <xs:sequence>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="title.cnt" minOccurs="0"/>
    </xs:choice>
  </xs:sequence>
</xs:group>
<xs:attributeGroup name="faq-question-statement.attributes">
  <xs:attribute name="outputclass" type="xs:string"/>
  <xs:attribute name="base" type="xs:string"/>
  <xs:attributeGroup ref="base-attribute-extensions"/>
  <xs:attributeGroup ref="id-atts"/>
  <xs:attributeGroup ref="localization-atts"/>
  <xs:attributeGroup ref="global-atts"/>
</xs:attributeGroup>
```

n. From the `commonElementMod.xsd` find the `<xs:complexType>` declaration for "shortdesc.class". Copy the complexType, title.content group, and title.attributes group and paste it into the faq-questionMod.xsd.

o. Change "shortdesc." to "faq-short-answer." everywhere, resulting in this set of declarations:

```
<xs:complexType name="faq-short-answer.class" mixed="true">
  <xs:sequence>
    <xs:group ref="faq-short-answer.content"/>
  </xs:sequence>
  <xs:attributeGroup ref="faq-short-answer.attributes"/>
</xs:complexType>
<xs:group name="faq-short-answer.content">
  <xs:sequence>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="title.cnt" minOccurs="0"/>
      <xs:group ref="draft-comment" minOccurs="0"/>
    </xs:choice>
  </xs:sequence>
</xs:group>
<xs:attributeGroup name="faq-short-answer.attributes">
  <xs:attribute name="outputclass" type="xs:string"/>
  <xs:attributeGroup ref="univ-atts"/>
  <xs:attributeGroup ref="global-atts"/>
</xs:attributeGroup>
```

11. Validate the test document. It should be valid.

At this point the new topic type declarations are correct but in order to make them usable you need to replace all the local URL references with the URNs defined in the entity resolution catalog.

If you have already packaged the DTD version of the module as a Toolkit plugin you can simply redeploy it in order to have the updated catalog for the XSD components hooked in. If you haven't packaged it as a Toolkit plugin, you should do so now.

12. Edit the test document and change the `xsi:noNamespaceSchemaLocation=` value to the URN you associated with the `faq-question.xsd` file, e.g.:

```
<faq-question
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
  xsi:noNamespaceSchemaLocation="urn:pubid:example.org:doctypes:dita:faq-
question.xsd"
 id="question-id">

 ...

</faq-question>
```

Validate the document. Assuming that you've deployed the new catalog correctly or otherwise hooked up the catalog into your validation system, the document should be valid.

13. Edit `faq-question.xsd` and modify the references to the `faq-questionGrp.xsd` and `faq-questionMod.xsd` files to use the corresponding URNs from the entity resolution catalog:

```
  ...

  <!--  ================ CONCEPT GROUP DEFINITION====================
-->
  <xs:include
schemaLocation="urn:pubid:example.org:doctypes:dita:modules:entities:faq-
questionGrp.xsd"/>
  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:conceptGrp.xsd:
1.2"/>

  ...

  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:topicMod.xsd:
1.2" />
  <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:conceptMod.xsd:
1.2" />
  <xs:include
schemaLocation="urn:pubid:example.org:doctypes:dita:modules:faq-
questionMod.xsd" />

  ...
```

14. Redeploy the Toolkit plugin and validate the test document. It should be valid.

Map Specialization Tutorial

Goal: Define a new map type that represents an FAQ publication.

Having defined a new topic type for FAQ questions (*Topic Specialization Tutorial* on page 71), we can now define a corresponding map type for creating complete FAQ sets.

New map types are used to either represent specific types of publications (what most people would call a "document type" in the generic sense, not the XML sense) or to represent specific combinations of topic types. For example, a common pattern is to have a task with associated concept and reference topics that together represent a single unit of supporting information for a specific task. It could be useful to define a separate map type that codifies this structure[9]

For this tutorial, we want to be able to define frequently asked question publications that consist entirely or mostly of FAQ question topics.

While you do this by simply defining a new map type that defined topicref types specifically for organizing FAQ entries, in practice it is usually better to define new topicref types as map domains so that they can be used in any map type. If you still want a specific map type you can define a map type that depends on the new map domain in order to define appropriately constrained content models.

The Learning and Training learningMapDomain and learningMap modules are a good example of this approach. The learningMap domain lets you use learning-specific topicrefs in any map type while the learningMap gives

[9] *Design patterns for information architecture with DITA map domains*, Erik Hennum, Don Day, John Hunt, and Dave Schell, September 2004, *http://www.ibm.com/developerworks/xml/library/x-dita7/*.

you a ready-to-use map specifically for learning content. The DITA for Publishers vocabulary takes the same approach with the publication map domain and pubmap map types.

Thus, this tutorial defines two vocabulary modules:

- An FAQ map domain that defines new topic types for organizing sets of FAQ entries
- An FAQ map type that uses the map domain to define a map that is exclusively an FAQ set.

To implement the domain and the map type that uses it we could start by defining the map domain and then defining the map type that uses it. However, the principle of test-driven development suggests that we start with the map type and work toward implementing the map domain.

Map Specialization Step 1: Design the Map Element Types

A typical FAQ consists of one or more groups of questions, where each group has a descriptive title. There may be some introductory material that is not itself FAQ questions.

That suggests that an FAQ map needs the following components:

- A topicref for referring to individual FAQ questions, e.g. "faq-question"
- A topic head type for creating sets of related questions, e.g. "faq-question-set"
- Some metadata for the FAQ set as a whole
- The ability to refer to generic (non-FAQ question) topics at the start of an FAQ set.

The metadata requirement is handled by the generic metadata for topicrefs and maps (at least for the purpose of this exercise). References to generic topics can be done using unspecialized `<topicref>`.

That just leaves two element types to be defined for the FAQ map domain:

`<faq-question>` Represents the use of a single FAQ question topic. Would normally be used with faq-question topics. Because we've defined the `<faq-question>` topic type to require short descriptions, it is probably appropriate to set the default value for the `type=` attribute to "faq-question" so that users are warned if they point to other topic types. Because FAQ questions are explicitly atomic, `<faq-question>` does not allow subordinate topicrefs.

`<faq-question-set>` Creates a group of related FAQ questions. May have an initial generic topic or may itself refer to a non-FAQ-question topic to act as an introduction or to satisfy the need for title-only topics in some processing environments.

The content model is:

```
(topicmeta?,
 topicref?,
 faq-question*)
```

.

For the FAQ map type we need the following element types:

`<faq-map>` Specializes from `<map>` and serves as the root of an FAQ publication. It's content model is:

```
(title,
 topicmeta?,
 keydef*,
 faq-question-set,
 reltable*)
```

By requiring exactly one `<faq-question-set>` the map clearly establishes the root of the overall FAQ navigation structure.

Map Specialization Step 2: Create New Document Type Shell DTD

In your work area create the directories `faq-map`. Within `faq-map` create the directory `dtd`.

Also create the directories `faq-mapDomain` and `faq-mapDomain/dtd`. In the `faq-mapDomain/dtd` directory create the empty files `faq-mapDomain.ent` and `faq-mapDomain.mod`. If you want to have some content in these files you can create appropriate DTD header comments, e.g.:

```
<!-- ================================
     FAQ Map Domain

     Copyright (c) 2010 Your Name Here

     ================================ -->
```

These stub files will allow us to create references to these modules from the faq-map document type shell before we've actually defined the faq-mapDomain module.

In the `faq-map` directory create the file `catalog.xml` with this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
prefer="public">

  <nextCatalog catalog="dtd/catalog.xml"/>
  <!--<nextCatalog catalog="xsd/catalog.xml"/>-->

</catalog>
```

In the standard DITA DTD distribution, find the file `base/basemap.dtd` and copy it as file "`faq-map/dtd/faq-map.dtd`".

Edit `faq-map.dtd` and modify it as follows:

1. Replace the header comment with one that reflects the new map type and your ownership.
2. Find the comment "DOMAIN ENTITY DECLARATIONS" and after it add this parameter entity declaration and reference:

   ```
   <!-- ============================================================= -->
   <!--                    DOMAIN ENTITY DECLARATIONS                 -->
   <!-- ============================================================= -->

   <!ENTITY % faq-map-d-dec
     PUBLIC "faq-mapDomain.ent"
            "../../faq-mapDomain/dtd/faq-mapDomain.ent"
   >%faq-map-d-dec;
   ```

   Remember that we haven't created the FAQ map domain yet, so this reference currently points to a file that doesn't exist.
3. Find the comment "DOMAIN EXTENSIONS" and delete the declaration for `%topicref;`.

   You are deleting the declaration for `%topicref;` because while we need the mapGroup domain so we can use the `<keydef>` element type, we will be allowing it in a specific place in the content model for `<faq-map>`, so we don't want it to be allowed wherever `<topicref>` is allowed.

   

   Note: Normally you would expect to add a reference `%faq-question-set;` here, but because the faq-map map type will depend on the faq-map domain it will be directly defining where `<faq-question-set>` is allowed and therefore we don't want to generally allow `<faq-question-set>` anywhere `<topicref>` is allowed.
4. Find the declaration for the `%map-type;` parameter entity and add this declaration after it:

   ```
   <!ENTITY % map-type
     PUBLIC "-//OASIS//ELEMENTS DITA 1.2 Map//EN"

   "map.mod"
   >%map-type;

   <!ENTITY % faq-map-type
     PUBLIC "fap-map.mod"
            "fap-
   ```

```
map.mod"
>%faq-map-type;
```

5. Find the comment "DOMAIN ELEMENT INTEGRATION" and add this parameter entity declaration and reference:

```
<!-- ============================================================= -->
<!--                      DOMAIN ELEMENT INTEGRATION               -->
<!-- ============================================================= -->

<!ENTITY % faq-map-d-def
  PUBLIC "faq-mapDomain.mod"
         "../../faq-mapDomain/dtd/faq-mapDomain.mod"
>%faq-map-d-def;
```

6. In the `faq-map` directory create a new XML document named `fap-map-test-01.ditamap` and give it this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE faq-map
  PUBLIC "faq-map" "dtd/faq-map.dtd">
<faq-map>
  <title>FAQ Map Test 01</title>
</faq-map>
```

   Validate the document. It should fail with a message to the effect that it cannot find the file "faq-map.mod".

At this point you have what should be a good document type shell and a document that uses it.

The next step is to implement the faq-map type module.

Map Specialization Step 3: Create faq-map Map Type Module

Find the file `map.mod` in the standard DITA DTD distribution. You will be cutting and pasting from `map.mod` into the new module file for the faq-map map type.

Construct the new module file as follows:

1. Create the file `faq-map/dtd/faq-map.mod` and give it this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--=======================================
    FAQ Map map type module.

    Represents a single FAQ publication

    Copyright (c) 2010 Your Name Here

    ======================================== -->



 <!-- ======== End of FAQ Map Module ======  -->
```

2. In `map.mod` find the declaration for `%map.content;` and copy that declaration and the next three declarations (`%map.attributes;` and the ENTITY and ATTLIST declarations for `<map>`) and paste them into the `faq-map.mod` file after the header comment:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--=======================================
    FAQ Map map type module.

    Represents a single FAQ publication

    Copyright (c) 2010 Your Name Here

    ======================================== -->

<!ENTITY % map.content
```

```
                        "((%title;)?,
                          (%topicmeta;)?,
                          (%anchor; |
                           %data.elements.incl; |
                           %navref; |
                           %reltable; |
                           %topicref;)* )"
>
<!ENTITY % map.attributes
             "title
                          CDATA
                                    #IMPLIED
              id
                          ID
                                    #IMPLIED
              %conref-atts;
              anchorref
                          CDATA
                                    #IMPLIED
              outputclass
                          CDATA
                                    #IMPLIED
              %localization-atts;
              %topicref-atts;
              %select-atts;"
>
<!ELEMENT map    %map.content;>
<!ATTLIST map
              %map.attributes;
              %arch-atts;
              domains
                          CDATA
                                    "&included-domains;"
>


 <!-- ======== End of FAQ Map Module ======  -->
```

3. Modify these declarations by changing "map" to "faq-map" everywhere, e.g.:

```
<!ENTITY % faq-map.content
                        "((%title;)?,
                          (%topicmeta;)?,
                          (%anchor; |
                           %data.elements.incl; |
                           %navref; |
                           %reltable; |
                           %topicref;)* )"
>
<!ENTITY % faq-map.attributes
 ...
<!ELEMENT faq-map    %faq-map.content;>
<!ATTLIST faq-map
              %faq-map.attributes;
  ...
>
```

4. Change the definiition of the `%faq-map.content;` parameter entity to:

```
<!ENTITY % faq-map.content
                        "((%title;)?,
                          (%topicmeta;)?,
                          (%keydef)*,
                          (%faq-question-group;),
                          (%reltable;)*)"
>
```

5. Change the value of the `domains=` attribute declaration in the pub-map ATTLIST declaration to reflect the module hierarchy and the required map domains:

```
<!ATTLIST faq-map
  %faq-map.attributes;
  %arch-atts;
     domains
     CDATA
       "(map faq-map mapGroup faq-mapDomain)
        &included-domains;"
>
```

6. Validate the test document. You should get messages the effect that the parameter entities `%keydef;` and `%faq-question-group;` were referenced but not declared.

   Because this map type module depends on specific domains, we have to locally declare the parameter entities for those element types we use from the domains because the domain-provided parameter entities are not included by the shell DTD until after the map type module is included.

7. Immediately after the header comment, add these parameter entity declarations:

```
<!ENTITY % keydef           "keydef"           >
<!ENTITY % faq-question-set "faq-question-set" >
```

   Validate the test document again. It should now give a message that the document is missing required content.

8. Edit the test document to add the minimum allowed content, resulting in this document:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE faq-map
  PUBLIC "faq-map" "dtd/faq-map.dtd">
<faq-map>
  <title>FAQ Map Test 01</title>
  <topicmeta></topicmeta>
  <keydef keys="foo"/>
  <faq-question-set>

  </faq-question-set>
</faq-map>
```

   Validate the test document again. It should now complain that the element type `<faq-question-set>` is not declared, which of course it is not.

9. Immediately before the ending comment, add the `class=` attribute declaration for the `<faq-map>` element:c

```
<!ATTLIST faq-map    %global-atts;  class CDATA "- map/map faq-map/faq-
map "        >
```

At this point the faq-map map type should be ready to use as soon as the faq-mapDomain module is defined.

Map Specialization Step 4: Create faq-mapDomain Module

The faq-mapDomain map domain defines the `<faq-question-set>` and `<faq-question>` topicref types.

Create the module as follows:

1. Edit the file `faq-mapDomain/dtd/faq-mapDomain.ent` and add these declarations:

```
<!-- ============================================================= -->
<!--                    ELEMENT EXTENSION ENTITY DECLARATIONS      -->
<!-- ============================================================= -->

<!ENTITY % faq-map-d-topicref
  "faq-question-set |
   faq-question
  "
>



<!-- ============================================================= -->
<!--                    DOMAIN ENTITY DECLARATION                  -->
```

```
<!-- ============================================================= -->

<!ENTITY faq-map-d-att
  "(map faq-map-d)"
>
```

2. Edit the file `faq-mapDomain/dtd/faq-mapDomain.mod` and add these parameter entity declarations:

```
<!-- ================================
     FAQ Map Domain

     Copyright (c) 2010 Your Name Here

     ================================ -->

<!ENTITY % faq-question-set "faq-question-set" >
<!ENTITY % faq-question     "faq-question" >
```

3. In the file `map.mod` find the declarations for the `<topicref>` element and copy them into the `faq-mapDomain.mod` file:

```
<!ENTITY % topicref.content
                       "((%topicmeta;)?,
                         (%anchor; |
                          %data.elements.incl; |
                          %navref; |
                          %topicref;)* )"
>
<!ENTITY % topicref.attributes
             "navtitle
                        CDATA
                                  #IMPLIED
              href
                        CDATA
                                  #IMPLIED
              keyref
                        CDATA
                                  #IMPLIED
              keys
                        CDATA
                                  #IMPLIED
              query
                        CDATA
                                  #IMPLIED
              copy-to
                        CDATA
                                  #IMPLIED
              outputclass
                        CDATA
                                  #IMPLIED
              %topicref-atts;
              %univ-atts;"
>
<!ELEMENT topicref    %topicref.content;>
<!ATTLIST topicref    %topicref.attributes;>
```

4. Globally change "topicref" to "faq-question-set" except for the reference to the `%topicref.atts;` parameter entity:

```
<!ENTITY % faq-question-set.content
                       "((%topicmeta;)?,
                         (%anchor; |
                          %data.elements.incl; |
                          %navref; |
                          %faq-question-set;)* )"
>
<!ENTITY % faq-question-set.attributes
             "navtitle
```

```
                        CDATA
                                    #IMPLIED
            href
                        CDATA
                                    #IMPLIED
            keyref
                        CDATA
                                    #IMPLIED
            keys
                        CDATA
                                    #IMPLIED
            query
                        CDATA
                                    #IMPLIED
            copy-to
                        CDATA
                                    #IMPLIED
            outputclass
                        CDATA
                                    #IMPLIED
            %topicref-atts;
            %univ-atts;"
>
<!ELEMENT faq-question-set    %faq-question-set.content;>
<!ATTLIST faq-question-set    %faq-question-set.attributes;>
```

5. Change the content model for <faq-question-set> to:

```
<!ENTITY % faq-question-set.content
                      "((%topicmeta;)?,
                        (%faq-question;)*)"
>
```

6. Copy the declarations for <faq-question-set> and paste them after the ATTLIST declaration for <faq-question-set>. Change "faq-question-set" to "faq-question":

```
<!ENTITY % faq-question.content
                      "((%topicmeta;)?,
                        (%faq-question;)*)"
>
<!ENTITY % faq-question.attributes
             "navtitle
                        CDATA
                                    #IMPLIED
              href
                        CDATA
                                    #IMPLIED
              keyref
                        CDATA
                                    #IMPLIED
              keys
                        CDATA
                                    #IMPLIED
              query
                        CDATA
                                    #IMPLIED
              copy-to
                        CDATA
                                    #IMPLIED
              outputclass
                        CDATA
                                    #IMPLIED
              %topicref-atts;
              %univ-atts;"
>
```

```
<!ELEMENT faq-question    %faq-question.content;>
<!ATTLIST faq-question    %faq-question.attributes;>
```

7. Change the content model for <faq-question> to:c

```
<!ENTITY % faq-question.content
                      "((%topicmeta;)?)"
>
```

8. From map.mod find the class= attribute declaration for topicref= and paste it into faq-mapDomain.mod.

```
<!ATTLIST topicref    %global-atts;  class CDATA "- map/topicref "   >
```

9. Change the element type from "topicref" to "faq-question", change the "-" to "+", and add " faq-map-d/faq-question " to the end of the attribute value:

```
<!ATTLIST faq-question    %global-atts;  class CDATA "+ map/topicref faq-
map-d/faq-question "   >
```

10. Validate the test document. It should validate. Verify that you can add a <faq-question> topicref within <faq-question-set>.

You're done. The FAQ map type and map domains have been declared and validated. All that remains is adding the appropriate catalog entries and updating the faq-map document type shell to use the public IDs for the module files and packaging the lot as one or more Toolkit plugins.

Map Specialization: XSD Version

As for the DTD version of the map specialization, we will create both a map domain and a map type.

Map Specialization XSD Step 1: Define faq-map Map Type

To define the <faq-map> map type, do the following:

1. Create the directory faq-map/xsd.
2. Create the directory faq-mapDomain/xsd.
3. Create the file faq-mapDomain/xsd/faq-mapDomain.xsd with this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">

  <xs:import namespace="http://www.w3.org/XML/1998/namespace"
    schemaLocation="urn:oasis:names:tc:dita:xsd:xml.xsd:1.2"/>

</xs:schema>
```

   This file acts as the stub for the FAQ map map domain, which we will define in Step 2.
4. Find the file basemap.xsd from the standard DITA schema distribution and copy it into the faq-map/xsd directory and rename it to faq-map.xsd.
5. Create test document faq-map/faq-map-test-xsd-01.ditamap with this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<faq-map xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="xsd/faq-map.xsd">
  <title>FAQ Map Test 01</title>
  <topicmeta></topicmeta>
  <keydef keys="foo"/>
  <faq-question-set>
    <faq-question/>
  </faq-question-set>
</faq-map>
```

   Validate the document. It should complain that the element type <faq-map> is not declared.
6. Edit faq-map.xsd.

7. Replace the header comment with one reflecting your ownership:

```
<!-- ============================================================
     FAQ Map document type shell

     Copyright (c) 2010 Your Name Here
     ============================================================ -->

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  xmlns:ditaarch="http://dita.oasis-open.org/architecture/2005/">
  <!-- ======= INCLUDE ======= -->

  ...

</xs:schema>
```

8. Find the reference to mapMod.xsd and add an inclusion to faq-mapMod.xsd after it:

```
 ...

 <xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:mapMod.xsd:
1.2" />
 <xs:include schemaLocation="faq-mapMod.xsd" />

 ...
```

9. Find the declaration for the "domains-att" attribute group and add "(map faq-map faq-map-d) " to the `domains=` attribute value:

```
  <xs:attributeGroup name="domains-att">
    <xs:attribute name="domains" type="xs:string"
       default="(map faq-map faq-map-d) (map mapgroup-d) (topic indexing-
d) (topic delay-d)"/>
  </xs:attributeGroup>
```

10. Create the file `faq-map/xsd/faq-mapMod.xsd` with this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  xmlns:ditaarch="http://dita.oasis-open.org/architecture/2005/">


</xs:schema>
```

11. Add a header comment as for the `faq-map.xsd` file.

12. Edit `mapMod.xsd` from the standard DITA schema distribution. Find the comment "Import - XML Attributes and Namespaces" and copy it and the two imports following it into `faq-mapMod.xsd`:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  xmlns:ditaarch="http://dita.oasis-open.org/architecture/2005/">

  <!-- ======= IMPORT - XML attributes and namespace ======= -->
  <xs:import namespace="http://www.w3.org/XML/1998/namespace"
schemaLocation="urn:oasis:names:tc:dita:xsd:xml.xsd:1.2"/>
  <xs:import namespace="http://dita.oasis-open.org/architecture/2005/"
schemaLocation="urn:oasis:names:tc:dita:xsd:ditaarch.xsd:1.2"/>

  ...

</xs:schema>
```

13. Find the `<xs:redefine>` element for mapGrp.xsd and add an include of `faq-mapDomain.xsd` immediately before it:

```
...

<xs:include schemaLocation="../../faq-mapDomain/xsd/faq-mapDomain.xsd"/>

<xs:redefine schemaLocation="urn:oasis:names:tc:dita:xsd:mapGrp.xsd:
1.2">

...
```

14. In `mapMod.xsd` find the declarations for the `<map>` element type (they should all be together in the file) and copy them into `faq-mapMod.xsd`.
15. Change "map" to "faq-map" everywhere except in the default value of the `class=` attribute:

```
<xs:element name="faq-map">
  <xs:annotation>
    <xs:documentation>
      The &lt;<keyword>faq-map</keyword>&gt; element represents
      an FAQ publication.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="faq-map.class">
        <xs:attribute ref="class" default="- map/map " />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:complexType name="faq-map.class" >
  <xs:sequence>
    <xs:group ref="faq-map.content"/>
  </xs:sequence>
  <xs:attributeGroup ref="faq-map.attributes"/>
</xs:complexType>

<xs:group name="faq-map.content">
  <xs:sequence>
    <xs:sequence>
      <xs:group ref="title" minOccurs="0" />
      <xs:group ref="topicmeta" minOccurs="0" />
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:group ref="navref" />
        <xs:group ref="anchor" />
        <xs:group ref="topicref" />
        <xs:group ref="reltable" />
        <xs:group ref="data.elements.incl" />
      </xs:choice>
    </xs:sequence>
  </xs:sequence>
</xs:group>

<xs:attributeGroup name="faq-map.attributes">
  <xs:attribute name="title" type="xs:string" />
  <xs:attribute name="id" type="xs:ID" />
  <xs:attributeGroup ref="conref-atts" />
  <xs:attribute name="anchorref" type="xs:string" />
  <xs:attribute name="outputclass" type="xs:string" />
  <xs:attributeGroup ref="domains-att"/>
  <xs:attributeGroup ref="topicref-atts" />
  <xs:attributeGroup ref="select-atts" />
  <xs:attributeGroup ref="localization-atts"/>
  <xs:attribute ref="ditaarch:DITAArchVersion"/>
```

```
    <xs:attributeGroup ref="global-atts" />
  </xs:attributeGroup>
```

16. Set the default value of the `class=` attribute to "- map/map faq-map/faq-map ":

```
  <xs:element name="faq-map">
    <xs:annotation>
      <xs:documentation>
        The &lt;<keyword>faq-map</keyword>&gt; element represents
        an FAQ publication.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="faq-map.class">
          <xs:attribute ref="class" default="- map/map faq-map/faq-map
" />
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
```

17. Modify the content model defined in the faq-map.content group to:

```
  <xs:group name="faq-map.content">
    <xs:sequence>
      <xs:sequence>
        <xs:group ref="title" minOccurs="0" />
        <xs:group ref="topicmeta" minOccurs="0" />
        <xs:group ref="keydef" minOccurs="0" maxOccurs="unbounded"/>
        <xs:group ref="faq-question-set" minOccurs="1" maxOccurs="1"/>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:group ref="reltable" />
        </xs:choice>
      </xs:sequence>
    </xs:sequence>
  </xs:group>
```

18. Validate the test document. You should get a message to the effect that the group named "faq-question-set" cannot be resolved.

At this point the FAQ-map map type document type shell and module is complete. Now we just need to define the FAQ-map map domain.

Map Specialization XSD Step 2: Define faq-mapDomain Map Domain

Create the FAQ-map map domain module as follows:

1. Edit the file `faq-mapDomain/xsd/faq-mapDomain.xsd`.
2. In `mapGrp.xsd` from the standard DITA schema distribution, find the group for "topicref" and copy it into `faq-mapDomain.xsd` after the `<xs:import>` element:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">

  <xs:import namespace="http://www.w3.org/XML/1998/namespace"
    schemaLocation="urn:oasis:names:tc:dita:xsd:xml.xsd:1.2"/>

  <xs:group name="topicref">
    <xs:sequence>
      <xs:choice>
        <xs:element ref="topicref"/>
      </xs:choice>
    </xs:sequence>
  </xs:group>

  ...
```

```
</xs:schema>
```

3. In `mapMod.xsd` find the declarations for `<topicref>` and copy them to `faq-mapDomain.xsd`:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">

  <xs:import namespace="http://www.w3.org/XML/1998/namespace"
    schemaLocation="urn:oasis:names:tc:dita:xsd:xml.xsd:1.2"/>

  <xs:group name="topicref">
    <xs:sequence>
      <xs:choice>
        <xs:element ref="topicref"/>
      </xs:choice>
    </xs:sequence>
  </xs:group>

  <xs:element name="topicref">
    <xs:annotation>
      <xs:documentation>
        The &lt;<keyword>topicref</keyword>&gt; element designates a
topic
        (such as a concept, task, or reference) as a link in a DITA map.
A &lt;<keyword>topicref</keyword>&gt;
        can contain other&lt;<keyword>topicref</keyword>&gt; elements,
allowing you to
        express navigation or table-of-contents hierarchies, as well as
implying relationships
        between the containing &lt;<keyword>topicref</keyword>&gt; and
its children.
        You can set the collection-type of a container
&lt;<keyword>topicref</keyword>&gt;
        to determine how its children are related to each other.
Relationships end
        up expressed as links in the output (with each participant in a
relationship
        having links to the other participants).
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="topicref.class">
          <xs:attribute ref="class" default="- map/topicref " />
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="topicref.class">
    <xs:sequence>
      <xs:group ref="topicref.content"/>
    </xs:sequence>
    <xs:attributeGroup ref="topicref.attributes"/>
  </xs:complexType>

  <xs:group name="topicref.content">
    <xs:sequence>
      <xs:sequence>
        <xs:group ref="topicmeta" minOccurs="0"/>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:group ref="navref" />
          <xs:group ref="anchor" />
```

```
        <xs:group ref="topicref" />
        <xs:group ref="data.elements.incl" />
      </xs:choice>
    </xs:sequence>
  </xs:sequence>
</xs:group>

<xs:attributeGroup name="topicref.attributes">
  <xs:attribute name="navtitle" type="xs:string"/>
  <xs:attribute name="href" type="xs:string"/>
  <xs:attribute name="keys" type="xs:string"/>
  <xs:attribute name="keyref" type="xs:string"/>
  <xs:attribute name="query" type="xs:string"/>
  <xs:attribute name="copy-to" type="xs:string"/>
  <xs:attributeGroup ref="topicref-atts" />
  <xs:attributeGroup ref="univ-atts" />
  <xs:attribute name="outputclass" type="xs:string"/>
  <xs:attributeGroup ref="global-atts" />
</xs:attributeGroup>

</xs:schema>
```

4. Change "topicref" to "faq-question-set" everywhere but in the default value for the `class=` attribute and in the reference to the "topicref-atts" attribute group:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified">

<xs:import namespace="http://www.w3.org/XML/1998/namespace"
  schemaLocation="urn:oasis:names:tc:dita:xsd:xml.xsd:1.2"/>

<xs:group name="faq-question-set">
  <xs:sequence>
    <xs:choice>
      <xs:element ref="faq-question-set"/>
    </xs:choice>
  </xs:sequence>
</xs:group>

<xs:element name="faq-question-set">
  <xs:annotation>
    <xs:documentation>
      The &lt;<keyword>faq-question-set</keyword>&gt; element
      groups sets of <keyword>faq-question</keyword> references
      together. The question set should either point to a topic
      that provides a title for the question set or specify
      a navigtion title.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="faq-question-set.class">
        <xs:attribute ref="class" default="- map/topicref " />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:complexType name="faq-question-set.class">
  <xs:sequence>
    <xs:group ref="faq-question-set.content"/>
  </xs:sequence>
  <xs:attributeGroup ref="faq-question-set.attributes"/>
</xs:complexType>
```

```
<xs:group name="faq-question-set.content">
  <xs:sequence>
    <xs:sequence>
      <xs:group ref="topicmeta" minOccurs="0"/>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:group ref="navref" />
        <xs:group ref="anchor" />
        <xs:group ref="faq-question-set" />
        <xs:group ref="data.elements.incl" />
      </xs:choice>
    </xs:sequence>
  </xs:sequence>
</xs:group>

<xs:attributeGroup name="faq-question-set.attributes">
  <xs:attribute name="navtitle" type="xs:string"/>
  <xs:attribute name="href" type="xs:string"/>
  <xs:attribute name="keys" type="xs:string"/>
  <xs:attribute name="keyref" type="xs:string"/>
  <xs:attribute name="query" type="xs:string"/>
  <xs:attribute name="copy-to" type="xs:string"/>
  <xs:attributeGroup ref="topicref-atts" />
  <xs:attributeGroup ref="univ-atts" />
  <xs:attribute name="outputclass" type="xs:string"/>
  <xs:attributeGroup ref="global-atts" />
</xs:attributeGroup>
```

5. Validate your test document. You should now get a message to the effect that the content of `<faq-question-set>` is invalid starting with `<faq-question>`.
6. Modify the content model for `<faq-question-set>` to:

```
<xs:group name="faq-question-set.content">
  <xs:sequence>
    <xs:sequence>
      <xs:group ref="topicmeta" minOccurs="0"/>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:group ref="faq-question" />
      </xs:choice>
    </xs:sequence>
  </xs:sequence>
</xs:group>
```

7. Modify the value of the `class=` attribute to "+ map/topicref faq-map-d/faq-question-set ":

```
<xs:element name="faq-question-set">
  <xs:annotation>
    <xs:documentation>
      The &lt;<keyword>faq-question-set</keyword>&gt; element
      groups sets of <keyword>faq-question</keyword> references
      together. The question set should either point to a topic
      that provides a title for the question set or specify
      a navigtion title.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="faq-question-set.class">
        <xs:attribute ref="class" default="+ map/topicref faq-map-d/
faq-question-set " />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

8. Validate your test document. You should now get a message to the effect that the group "faq-question" cannot be resolved.
9. Copy all the declarations for `<faq-question-set>` and rename "faq-question-set" to "faq-question", resulting in these declarations:

```
  <xs:group name="faq-question">
    <xs:sequence>
      <xs:choice>
        <xs:element ref="faq-question"/>
      </xs:choice>
    </xs:sequence>
  </xs:group>

  <xs:element name="faq-question">
    <xs:annotation>
      <xs:documentation>
        The &lt;<keyword>faq-question</keyword>&gt; element
        represents a single FAQ question within a larger
        FAQ question set.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="faq-question.class">
          <xs:attribute ref="class" default="+ map/topicref faq-map-d/
faq-question " />
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="faq-question.class">
    <xs:sequence>
      <xs:group ref="faq-question.content"/>
    </xs:sequence>
    <xs:attributeGroup ref="faq-question.attributes"/>
  </xs:complexType>

  <xs:group name="faq-question.content">
    <xs:sequence>
      <xs:sequence>
        <xs:group ref="topicmeta" minOccurs="0"/>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:group ref="faq-question" />
        </xs:choice>
      </xs:sequence>
    </xs:sequence>
  </xs:group>

  <xs:attributeGroup name="faq-question.attributes">
    <xs:attribute name="navtitle" type="xs:string"/>
    <xs:attribute name="href" type="xs:string"/>
    <xs:attribute name="keys" type="xs:string"/>
    <xs:attribute name="keyref" type="xs:string"/>
    <xs:attribute name="query" type="xs:string"/>
    <xs:attribute name="copy-to" type="xs:string"/>
    <xs:attributeGroup ref="topicref-atts" />
    <xs:attributeGroup ref="univ-atts" />
    <xs:attribute name="outputclass" type="xs:string"/>
    <xs:attributeGroup ref="global-atts" />
  </xs:attributeGroup>
```

10. Validate your test document. It should be valid.

11. Modify the content model of `<faq-question>` to remove the reference to `<faq-question-set>`:

```
<xs:group name="faq-question.content">
  <xs:sequence>
    <xs:sequence>
      <xs:group ref="topicmeta" minOccurs="0"/>
    </xs:sequence>
  </xs:sequence>
</xs:group>
```

12. Validate your test document. It should still be valid. Verify that you cannot add any subelements to `<faq-question>` other than `<topicmeta>`.

At this point the FAQ map and FAQ-map map domain declarations are complete. All that remains is defining appropriate URNs for the schema locations, modifying the document type shell to use the URNs rather than relative URLs, and package it all up as a Toolkit plugin.

Index

A

C

D

E

F

I

M

O

P

R

S

T

U

V

X