The classpack LaTeX 2$_\varepsilon$ package*†

XML mastering for LaTeX classes and packages

Peter Flynn
Silmaril Consultants
Textual Therapy Division
(peter@silmaril.ie)

May 28, 2013

Summary

LaTeX document classes and packages are normally created, maintained, and distributed in `.dtx` format using the ltxdoc class, which provides facilities for modular or fragmentary coding combined with interleaved documentation. However, the accurate construction of these files is technically challenging.

ClassPack allows a developer to create an XML document containing user documentation and annotated code, based on the *DocBook* vocabulary (with some minor abuses). An XSLT2 script then generates the `.dtx` and `.ins` files, ensuring that all the relevant pieces are emitted in the correct order and in the correct syntax.

This is experimental software, and is incomplete. It has been successfully used in-house by the author since 2008 for several institutional and commercial packages and classes. There are some known deficiencies which remain to be corrected, and some legacy code (originally included for one specific package) which needs to be removed to an external file.

A paper describing the system has been accepted for the Balisage markup conference 2013 in Montréal.

---

*This document corresponds to classpack *v.* 0.77, dated 2013/05/28.

†Development has been supported by UCC Electronic Publishing Unit.

Contents

1 Introduction

LaTeX document classes (templates) and packages (styles) are traditionally distributed as pairs of `.dtx` and `.ins` files, written to the specifications and recommendations of the doc, ltxdoc, and clsguide packages.

- The `.dtx` (DocTeX) file is a literate-programming document, containing modular code and annotations interleaved in such a way that each fragment of code and its explanation are adjacent;
- The `.ins` file is an installer: when run through LaTeX, it extracts the code from the `.dtx` file into the relevant class (`.cls`), package (`.sty`), and other files;
- Running LaTeX on the `.dtx` file itself extracts and typesets the documentation.

The construction of a `.dtx` document is quite complex, with a special set of tags and conventions to allow documentation to be separately identifiable to code. The file format relies on the documentation being shielded by a leading percent-space (`%␣`) armour on each line to prevent it being interpreted as part of the code; and the environment tags surrounding the code itself must be shielded by four such spaces (`%␣␣␣␣`). Apart from the documentation, the treatment of the code and control statements resembles more a data specification (which in some ways it is) than a conventional text document.

XML, particularly in its traditional ‘document’ mode, as distinct from its use as a data exchange format, offers many similar features to LaTeX (for example, the named identification of document components), but with a rigid and invariable syntax that can be checked programmatically by any validating XML processor. By contrast, a LaTeX document (and more specifically, a `.dtx` document) can only be proved by running it through LaTeX itself: there is no equivalent to the ‘pre-flight’ type of standalone parsing or validating available with XML.

The *DocBook* vocabulary of XML is designed for technical documentation in computing. It provides markup both for textual documentation *and* for data-like structures that occur in computer documentation, making it a viable candidate for describing a literate-programming type of document such as DocTeX.

The *ClassPack* system is an experiment in using *DocBook* XML as the storage format for LaTeX class and package source code, using the XLST2 language to transform the XML into pairs of `.dtx` and `.ins` files. There are a number of advantages to this approach:

- XML's syntax and document construction is extremely robust, and the design of the language means that an XML file can be machine-checked for errors of syntax and construction;
- XML markup is traditionally self-descriptive, with element types being named according to what they are intended to contain. For example, a variable name can be marked up as `<varname>foo</varname>`.

  While it is perfectly possible to create a `\varname` control sequence (macro) to do the same in LaTeX, it is rarely done. Instead, authors have typically preferred to use visual formatting like `\textit{foo}` for italics or `\verb+foo+` for monospace type. This method means the variable reference is not immediately identifiable as containing a variable name — it could be anything;
- Given suitably-descriptive markup, sharing document fragments between applications can be done programmatically, so a fragment implementing a LaTeX feature (with its associated documentation) can be re-used in other class and package applications at the XML level (eg with XInclude, or as an external entity) without the need for manual cutting and pasting;
- There is a very wide range of software (editors and processors, both free and non-free) available to handle XML documents, including a lot of useful tools for document management and information extraction.

Everything comes at a cost. The drawbacks of using XML for this include:

- It's another language to learn. Despite being so widespread, it's not yet a common skill. In particular, programmers dislike XML because it's a markup language, not a programming language, and the syntax is different from that of programming languages;
- Although there is plenty of software for editing XML, it is not well-developed for text documents in synchronous typographic form (often called 'WYSIWYG'); even the best or most expensive editors are designed for XML experts, not for the average user.

1.1 Contents

The *ClassPack* framework is enabled by two principal software components:

1. An XML vocabulary (the *DocBook* DTD or Schema) used for naming the component parts of documentation and code, and specifying where they belong and how they fit together;
2. An XSLT2 script to implement the logic of combination and separation needed to create the `.dtx` and `.ins` files.

The XML vocabulary used is *DocBook*, which is in widespread use for the documentation of computer systems, and is well-supported on all platforms. The current system uses version 5. Although it is highly modular and easily adapted for many purposes, only a few minor changes have been made for its use here, but a number of element types have been put to uses not envisaged by the developers.

This habit of using (some say, abusing) XML markup for different purposes is very common, and often deprecated because it is usually undocumented. This document explains what has been [ab]used and for what reasons. Once this system settles down, a more formal expression of the vocabulary can be made from the RNG source, removing the parts that are not required, and making it simpler to edit with.

The adaptation of *DocBook* in this version subsists mainly in the addition of some attributes and entity declarations to allow the conversion to LaTeX of special characters and other features that would otherwise involve extensive re-parsing of the character data content (text). The changes also implement a few modifications to the way the LaTeX code is output by the XSLT2 program for typographical purposes.

As an example, the entity defining the em rule character `—` is declared as a LaTeX tie (non-breaking space) followed by an em rule followed by a normal space (`~---␣`), so that it can be used between words — like this — without the need to worry about special spacing in the XML, or the inadvertent breaking of a line before the rule:

```
can be used between words&mdash;like this&mdash;without the
```

If a document style requires the use of unspaced dashes, all that needs changing is the entity declaration, not the whole document.

The current driver in Document Type Definition (dtd) format is listed in section A on page 45, and is distributed as file `doctexbook.dtd` so that it can be referenced as such in a document without having to copy and paste the declarations into every document.

1.2 Invocation

To create or edit a *ClassPack* XML document you must have the *ClassPack* DTD *and* the DocBook DTD installed and known to your XML editor. The DTD customisation file, `docbooktex.dtd` is distributed with this package; an RNG schema version will be available in the future.

With the DTD, the standard procedure is to specify it in the first line of your XML document. Your class or package documents must therefore start with a Document Type Declaration. This can specify the Formal Public Identifier (fpi) and the filename of the DTD:

```
<?xml version="1.0"?>
<!DOCTYPE book PUBLIC
  "+//Silmaril//DTD DocBook 5.0 for DocTeX//EN"
  "doctexbook.dtd">
```

or you can omit the FPI and specify the DTD as a `SYSTEM` keyword instead:

```
<?xml version="1.0"?>
<!DOCTYPE book SYSTEM "doctexbook.dtd">
```

The DTD file `doctexbook.dtd` can be anywhere on your system: in these examples it is assumed to be in the same directory as your document. If you store it elsewhere, just give the full filepath, for example:

```
<?xml version="1.0"?>
<!DOCTYPE book SYSTEM "/usr/local/lib/xml/dtds/doctexbook.dtd">
```

When you open a document starting like this, a conformant XML editor will look for `doctexbook.dtd` and read it, so that it then knows the names of all the element types that you can use, and how they fit together into a document.

1.3 Document structure

Every XML document must have one outermost enclosing element (the ‘root’ element) which holds everything else. The root element type used in a ClassPack document is `book`, as described in detail in section 2.1 on page 11.

Within the `book` element, the *metadata* (information about the document) is held in an `info` element, the *user documentation* in a `part` element with the ID of `doc`, and the *annotated code* in a second `part` element, with the ID of `code`.

```
<?xml version="1.0"?>
<!DOCTYPE book SYSTEM "doctexbook.dtd">
<book>
  <info>...</info>
  <part xml:id="doc">...</part>
  <part xml:id="code">...</part>
  <part xml:id="files">...</part>
</book>
```

An optional third `part` with the ID of `files` can be used to include ancillary files that are to be recreated as-is, without separate annotation or display, such as sample data or example documents (see section 1.3.3 on the next page).

Ancillary files that require annotating must go in the `code` part as described in section 3.2.1 on page 34.

1.3.1 User documentation

The documentation for the end user is descriptive text which explains how to use the package or class. The main division within a ClassPack `part` is the `chapter`, which can contain `sect1` sections, which can contain `sect2` subsections, and so on, as described in section 3.1 on page 32. User documentation must go in a `chapter` and its sub-elements: it must not be directly in the `part`.

(Because the latexdoc package is based on the article package, the ClassPack `chapter`s become latexdoc `\section`s; the ClassPack `sect1`s become latexdoc `\subsection`s, and so on.)

1.3.1.1 Structure

Your document can use most of the conventional structural features of any DocBook document: paragraphs, lists, figures, tables, and examples of code (both illustrative and for extraction) as described in section 3.2 on page 33. Most of the specialist constructs of DocBook are not implemented in ClassPack except for a few used in the `info` section for setup purposes. You should refer to the list of markup in section 3.2 on page 33 for exactly which element types do what.

1.3.1.2 Inline markup

Within the text you can use much of the conventional semantic markup provided in *DocBook* as described in section 3.3 on page 36.

1.3.2 Annotated code

In the `code` Part, you annotate the inner workings of your package or class. The use of the `chapter` and `sect1` divisional structure is also mandated here: everything within the chapters is regarded as the package or class.

Your code can be explained in fragments, either as alternating `para` and `programlisting` elements, or as `annotation` elements, each one describing a single macro or environment or other object, and containing `para` and `programlisting` (this makes it indexable). See the examples in section 3.2 on page 33.

1.3.3 Ancillary files

There are two types of extractable ancillary file:

- Ancillary files to be annotated and extracted along with the class or package file must each go in their own `appendix` element immediately after the last of the `chapter` element of the package or class in the `code` Part
- Ancillary files which just need to be extracted whole, and do not have any separate documentation must go in the `files` Part as described in section 1.3.3.

1.4 Tooling up

You need the following tools:

- an XML editor: I use *Emacs* with *psgml-mode* and *xxml-mode*, but any competent XML editor will do
- an XSLT2 processor: I use *Saxon* (this also means I installed *Java*)
- a full installation of LaTeX
- a PDF reader
- XML tools: I find the *LTxml2* toolkit from Edinburgh University invaluable for ad-hoc querying of documents.

2 Setting up a ClassPack document

Using *ClassPack* to create and maintain LaTeX classes and packages requires the following initial steps. These are only done once, at the start of a new class or package. A few items need periodic updating, such as the version number, when it changes; the revision history; and the list of packages, as and when needed.

1. setting up the configuration (see section 2.1) to specify the name, date, version, type, audience, status, and other key metadata;
2. setting up the documentary metadata (see section 2.2 on page 14) such as the title, author, contact details, abstract, and the initial entry in the revision history;
3. setting up the list of packages (see section 2.3 on page 17) required for *a*) the documentation (see section 2.3.1.1 on page 19) and *b*) the class or package itself (see section 2.3.1.2 on page 20), plus any additional initialization commands needed for the documentation.

Note particularly that the list of packages required by the class or package itself is not *stored inline to the code of the class or package. It is stored separately for reasons that are explained in detail in section 2.3 on page 17.*

2.1 Configuration and setup

The `book` element is the outermost container for the document. It is used to carry the configuration information in attributes, for example:

```
<book xml:id="classpack" arch="class" version="0" revision="71"
  status="beta" conformance="LaTeX2e" condition="2011-06-27"
  os="all" audience="lppl" security="0" vendor="Silmaril"
  xml:base="tex/latex" xlink:role="xxx" userlevel="cls"
  annotations="\raggedright" remap="a4paper,12pt">
```

`xml:id`: This must be the name of the class or package. It will be used as the LaTeX filename (with `.cls` or `.sty` and `.ins` added automatically), so it should be all lowercase and must consist of letters, digits, and hyphens only, starting with a letter.

The XML rules for IDs require this restriction, which currently makes it impossible to use ClassPack to maintain a class or

packages whose name begins with a digit.

arch: The ‘architecture’ of the document, which defines the type of file you are going to produce; this must be either `"class"` or `"package"`, corresponding exactly with the value of the `userlevel` attribute below.

userlevel: The file type of the class or package document; this must be either `cls` or `sty`, corresponding exactly with the value of the `arch` attribute above.

version: The major version of the class or package. Conventionally, development (α) or pre-release (β) versions of software start at version zero.

revision: The sub-version or release of your class or package. This is combined with the major version number, separated by a dot, to produce the complete version number.

The most recent revision history entry gets tested against this when the file is processed, and an error message is displayed if the version numbers do not match, as a warning that you have updated one without updating the other, and may therefore have forgotten to document a change (see the `revision` element in ‘**revhistory**’, the last item in the list in section 2.2 on page 16).

status: The development status of the class or package, eg `"alpha"`, `"beta"`, `"candidate"`, `"draft"`, `"final"`, etc.

conformance: The TEX format required to process the `.dtx` document. Only the value `"LaTeX2e"` is supported at the moment.

condition: The version of the format identified in `conformance`, expressed as an ISO date (`"yyyy-mm-dd"`).

Note that this is *not* in the LATEX format (`yyyy/mm/dd`).

os: The operating system[s] for which the class or package is relevant. Currently, only the value `"all"` is supported.

audience: The licence under which the class or package is made available. For normal publicly-available LATEX classes or packages which will be uploaded to CTAN, use the value `"lppl"` (LATEX Project Public Licence).

A copy of the LPPL is distributed with ClassPack in a file called `lppl.xml`, which must be copied or soft-linked (aliased) to each

directory in which you process ClassPack documents.

Classes or packages for private or commercial use will probably need to use another value, but it is used as a filename, so a `.xml` file with that name must exist in the directory in which your ClassPack document is processed. It must contain a *DocBook* `chapter` element containing the text of the licence, which will be included at the end of the documentation.

`security:` The checksum value emitted by ltxdoc when it processes your class or package to format your documentation. See the documentation for the ltxdoc package for details.

Setting this to zero avoids the LaTeX error message during early development, when every edit would change the checksum. As with the version values, you must update this value, if non-zero, to match the one that ltxdoc reports.

`vendor:` Your name, or the name of the organisation responsible for the work on this class or package.

`remap:` Any options to pass to the ltxdoc package, such as **`a4paper`**, **`12pt`**, etc.

This, and the following `annotations` attribute, make it easier to change the global formatting of the documentation.

`annotations:` Any LaTeX commands required for global application at the start of the documentation that cannot easily be included anywhere else (eg `\raggedright`, `\sffamily`, etc).

`xml:base:` The name of the *subdirectory* where the resulting `.cls` or `.sty` file should be installed in a TDS-compliant TeX installation, relative to the `texmf/tex/latex` directory of the tree. For most packages, this means the name of the directory you want created by the installation `.tds.zip` file, in which the `.cls` or `.sty` file will be put.

`xlink:role:` Optional. If this is set to a value, then the package being written will be included in the setup for the documentation (perhaps so that it can be used in examples), with the value of this attribute being used as the optional argument to the generated `\usepackage` command.

If the package is required with no options, use this attribute but set it to null (`""`). The date constraint is added automatically,

using the current package date as defined by the most recent entry in the `revhistory`.

Note that this only works for packages, not classes. Classes cannot not be documented using themselves.

It is essential to get these values correct, otherwise subsequent processing will produce unexpected results, or no results at all.

2.2 Metadata

The titling, specification of packages (separately for the documentation and for the class or package itself), revision history, abstract, copyright, and availability are all kept at the top of the document in an element called `info`, immediately after the `book` start-tag:

```
<info>
  <cover>...</cover>
  <!-- THE METADATA STARTS HERE -->
  <title>XML mastering for &amp;LaTeX; document classes...
  <author>...
  <copyright>...
  <releaseinfo>http://latex.silmaril.ie/software</releaseinfo>
  <annotation>...
  <abstract>...
  <revhistory>...
</info>
```

Of these, the `cover` element is the most complex, as it is [ab]used to hold all the details of packages and settings required for the class or package *and* for the production of the documentation. It therefore gets the whole of section 2.3 on page 17 to itself.

The other metadata element types are more obvious, and largely use the *DocBook* markup as intended.

`title:` This contains the title of the class or package in natural language. This is *not* the content of the `\title` command in LaTeX terms, in the `.dtx` file, which is automatically preset to the phrase ‘The *name* LaTeX 2ε document class’ (or ‘package’; where *name* is the name of your class or package as specified in the `xml:id` attribute of the `book` root element).

The title you give in *this* `title` element is the explanatory subtitle which appears *under* the automatically-generated one on the first page of the typeset documentation.

`author:` The `author` element provides identity markup for the author[s], using a `personname` element for each author, containing subelements for firstname, surname, and other forms of naming; optionally followed by an `affiliation` element, where you can identify employer or other status; address; email; and URI, as in the following example. The name, affiliation, and email address are used in the `\author` command of the `.dtx` file.

```
<author role="maintainer">
  <personname>
    <firstname>Peter</firstname>
    <surname>Flynn</surname>
  </personname>
  <affiliation>
    <orgname>Silmaril Consultants</orgname>
    <orgdiv>Textual Therapy Division</orgdiv>
  </affiliation>
  <address>Cork, Ireland</address>
  <email>peter@silmaril.ie</email>
  <uri>http://blogs.silmaril.ie/peter</uri>
  <contrib role="sponsor">UCC</contrib>
</author>
```

For multiple authors, you must enclose multiple `author` elements (one per author) in an outer `authorgroup` container element. One of the authors must be identified as the maintainer of the package or class, by adding the `role` attribute with the value `maintainer`.

A `contrib` element with a `role` attribute set to `"sponsor"` can be added in an `author` block, to give the name of an organisation sponsoring the development of the class or package.

`copyright:` This element lets you identify the `year` and the name of the `holder` (you, your employer, or some other entity).

```
<copyright>
  <year>2012</year>
  <holder>Silmaril Consultants</holder>
</copyright>
```

`releaseinfo:` In the absence of other ways of identifying where to find your class or package (assuming it will eventually find its

way onto CTAN), this element can be used to hold the URI of a location where it can be downloaded, such as your personal or business web site.

annotation: This element is used for a warning or notice you want placed in the Preamble of the `.ins` file, which is used for extracting your class or package from the `.dtx` file, where it will be seen by users installing the software.

abstract: The Abstract is formatted on the front page of your documentation. Like any abstract, it should summarise what the class or package does, and who might want to use it.

The `abstract` element may start with an optional `title` element, which (if present) will be used to change the value of the `\abstractname` in the `.dtx` file ('Summary' is a common choice).

The rest of the abstract is just paragraphs; note that lists, block quotations, figures, tables, etc are not allowed in an Abstract.

revhistory: This holds the top-level information about each major and minor revision, outlining the main changes. The version number of the most recent revision, as identified by the latest value of the `conformance` attribute of the `date` element, must match the version number composed from the major version and the revision in the `book` root element (see '**revision**', the fifth item in the list in section 2.1 on page 12).

```
<revision version="0.72">
  <date conformance="2012-02-11"/>
  <revdescription>
    <itemizedlist>
      <title>Wrote internal documentation</title>
      <listitem>
        <para>Created the classpack.xml template
                  as an example.</para>
      </listitem>
    </itemizedlist>
  </revdescription>
</revision>
```

Comments on individual changes to the code should be documented *at the code location*, using the `remark` element (see section 3.5 on page 40), eg

```
<remark version="0.70" revision="2010-05-29">Added
    timestamp</remark>
```

These remark elements get collated as \changes commands, and gathered together in the changelog by ltxdoc during processing.

2.3 Packages and related commands

As mentioned above, the cover element is used to provide a place where packages and other LaTeX preliminaries can be specified. Using this structure means each entry is separately editable, and the same structure is used both for packages for the documentation *and* packages for the class or package itself.

It would of course have been possible just to allow a slab of LaTeX code at these points, but that would have made commenting and documentation harder, and would also have made it more difficult to perform an XML element-copy–element-paste or an XInclude when using one package or class's settings as the basis for another.

The most important reason is that specifying package lists as separately-identifiable blocks makes it possible to automate the invocation of frequently-used packages, parameters, options, or settings which you may store separately for re-use (see section 2.3.2 on page 21) by adding your own modifications that you like to have included whenever you use a particular package.

In particular, the *autopackage* feature added to ClassPack in v0.74 means that most packages needed for documentation are now detected automatically on the basis of features you use in your documentation, making it unnecessary to specify them by hand. For example, if you use compact lists, ClassPack will detect this and add the enumitem package for you.

The XSLT2 program also uses this markup in order to modify the behaviour of the LaTeX code at several points (such as fixing the broken abstract formatting when the parskip package is used).

There are at least two, possibly four, sections in the cover element where the packages, commands, and other data can be defined:

```
<constraintdef xml:id="docpackages">
 ...packages for the user documentation are defined here...
</constraintdef>

<constraintdef xml:id="startdoc">
 ...special commands for the user documentation go here...
```

```
</constraintdef>

<constraintdef xml:id="clspackages" linkend="options">
 ...packages needed for the class or package are defined here...
</constraintdef>

<constraintdef xml:id="manifest">
 ...files to add to the MANIFEST/zip file are listed here...
</constraintdef>
```

These exact `xml:id` values are mandatory when the relevant `constraintdef` elements are used, as they are referenced from elsewhere in the document by the XSLT2 program. The only variation is that when writing a package (`.sty` file), the `"clspackages"` must read `"stypackages"` instead. The first three letters are used to match the three-letter filetype used as the value of the `arch` attribute that you specified on the `book` root element (see '**`arch`**', the second item in the list in section 2.1 on page 12).

Each `constraintdef` element can hold one or (in some circumstances, more) of the following element types:

- `segmentedlist` (in a `docpackages`, `clspackages`, or `stypackages` type of `constraintdef`*only*), a list structure used to specify the packages required: see section 2.3.1 on the next page;
- `cmdsynopsis`, used for defining *user documentation* setup commands to be placed *in* the Preamble. This is only meaningful in the `"docpackages"` type of `constraintdef`: see section 2.3.3 on page 25;
- `procedure`, used for holding blocks of *user documentation* setup commands to be placed *after* the Preamble (that is, at the start of the document body, after the `\begin{document}` command). This is only meaningful in the `"docpackages"` type of `constraintdef`.

  Note that this is distinct from commands to be placed *in* the Preamble, which are held in a more structured manner in the `cmdsynopsis` element deacribed above: see section 2.3.4 on page 29;
- `simplelist`, a list whose `member` elements are used to name additional files to be included in the distribution zip file (`MANIFEST`). This is only meaningful in a `manifest` type of `constraintdef`: see section 2.3.5 on page 30.

2.3.1 Specifying packages

There are three parts to using `constraintdef` for this:

1. specifying packages for your user documentation (see section 2.3.1.1);
2. specifying packages for the class or package you are writing (see section 2.3.1.2 on the following page);
3. automating the inclusion of extra settings to be used whenever you specify a particular package (see section 2.3.2 on page 21).

For the first two, the `segmentedlist` element is used. This contains a sequence of `seglistitems`, one per package, each containing a `seg` element holding the package name. An optional `segtitle` element may start the list, and if present, is used as a comment (for the documentation packages) or a subheading (class or package packages).

```
<segmentedlist>
  <segtitle>Packages required for documentation</segtitle>
  <seglistitem role="Use the Charter typeface for documentation.">
    <seg version="2005-04-12">charter</seg>
  </seglistitem>
  <seglistitem role="Use Helvetica as the sans-serif, but scale it
     to fit">
    <seg role="scaled=0.8333">helvet</seg>
  </seglistitem>
    ...
</segmentedlist>
```

Each `seglistitem` provides for a documentary comment about why this package is required, using the `role` attribute. In the case of the packages for your own class or package, this comment is reproduced in the documentation of the code.

The package itself is specified as the content of the `seg` element in each item.

Any options for the package being loaded must be supplied in the `role` attribute of the `seg` element. If the package must conform to a specific version, the date must be provided (in ISO format) in the `version` attribute.

2.3.1.1 Declaring packages needed for your documentation

Packages required for your documentation must be included in the

type of list described above, in the `constraintdef` element that has the `xml:id` value of `"docpackages"`. Note that some packages are automatically when certain types of formatting are implied: see section 2.3.2.2 on page 23 for details.

The relevant `\usepackage` commands get included in the `.dtx` file right after the `\begin{document}` command.

If you also want the package you are maintaining to be included in the documentation (perhaps so you can use it for examples), remember to set the `xlink:role` attribute on the `book` root element as described in '**`xlink:role`**', the last item in the list in section 2.1 on page 13.

See section 2.3.2 on the following page for details of how to specify package command settings that you want included by default every time you specify a particular package.

2.3.1.2 Declaring packages needed by the class or package itself

All the packages required for the class or package being written must be included in the type of list described above, in the `constraintdef` element that has the `xml:id` value of `"clspackages"` (for classes) or `"stypackages"` (for packages).

2.3.1.2.1 Specifying where in the `.dtx` file to output them:

Because your class or package design may include preliminary commands needed before packages are included, the relevant `\RequirePackage` commands must be added to the `.dtx` file *in a location that you must specify yourself*. This is done by giving the `linkend` attribute on the enclosing `constraintdef` element the value of an `xml:id` which you have assigned to a chapter or section in your annotated code.

There is no default: you must specify this link yourself, otherwise the list of required packages will not be output.

The reason is that you may need to write some of your class or package code (option declarations, for example, or a `\LoadPackage` command), *before* the specified packages are loaded.

- If the chapter or section you have specified has content (text) in it, the `\RequirePackage` commands are output as the content of a new chapter or section immediately preceding or following it, as specified by the value of the `role` attribute ('before' or 'after').

- If the chapter or section you have specified is (deliberately) empty, the `\RequirePackage` commands are output as the content of that chapter or section.

As an example, let us say you specify the `constraintdef` with

```
<constraintdef xml:id="clspackages" linkend="options" role="after">...
```

You must then have a chapter or section in your documented code with the `xml:id` value of `"options"`. If it is empty (no character data content) like this:

```
<sect1 xml:id="options">
  <title/>
  <para/>
</sect1>
```

then the list of `\RequirePackage` commands will be output in its place.

If, on the other hand, the specified section has text and code of its own:

```
<sect1 xml:id="options">
    <title>Options</title>
    <para>text...</para>
    <programlisting>
\some{code}
    </programlisting>
</sect1>
```

then the list of `\RequirePackage` commands will be output immediately after it, as a new section at the same level.

In both cases, the `segtitle` of the `segmentedlist` will be used as the title of the section.

2.3.2 Automated settings for declared packages

There are several reasons for automating package setup:

- Many LaTeX authors and designers have ‘favourite’ settings that they like to use every time they specify a particular package.
- Some options have now become the *de facto* convention for their package, (for example, the **T1** option on the fontenc package).

- There are commands that need to be used whenever a particular package is invoked (for example, the makeidx package means you need to add the `\makeindex` command to the Preamble).
- Some packages are only needed in the documentation if a particular formatting feature is used (for example compact list spacing requires the enumitem package). This avoids you having to remember to include a specific package when you use such a feature; and to remove it if you cease to use the feature.

To help automate these, an ancillary (lookup) file called `prepost.xml` is used, which is a *DocBook* document with a `refsection` root element type containing two `procedure` elements, shown below.

The `prepost.xml` file must be in the directory specified by your setting of the `repo` runtime parameter.

```
<refsection>
  <title>Commands to use before and after packages</title>
  <procedure xml:id="prepackage">
    ...steps...
  </procedure>
  <procedure xml:id="postpackage">
    ...steps...
  </procedure>
</refsection>
```

The `"prepackage"` procedure is for material which needs to go *before* a package is invoked. The `"postpackage"` procedure is for material which needs to go *after* a package is invoked.

2.3.2.1 Identifying each package

Within these procedures, each package is identified in a `step` element.

- the `remap` attribute holds the package name;
- the `condition` attribute holds the type[s] of output it is intended to be effective for, `"doc"`, `"cls"`, or `"sty"` (space-separated if more than one);
- the `role` attribute holds any default options (comma-separated);
- an optional `para` element holds a textual description of the package and its use. This is only meaningful for packages marked in the `condition` attribute for use in the class or package. If present, this gets output to the code documentation.

```
<step role="utf8x" remap="inputenc" condition="cls sty">
```

```
  <para>UTF-8 is the default character set, to allow for use of
        any character in any writing system. Some characters
        are not specified for all fonts, so may have to be
        specified manually.</para>
</step>
```

In this example, specifying inputenc in the document, in a seg element as described in section 2.3.1 on page 19, results in the package being added with \RequirePackage to the class or package code.

2.3.2.2 Automating inclusion Each step may contain one or more constructorsynopsis elements which specify the condition[s] under which the package will automatically be included *without it needing to be specified* in a seg element as described in section 2.3.1 on page 19

- the condition attribute holds the name of an element type which, if present in the documentation, will cause the package to be included automatically;
- one or more methodparam subelements can be used to specify attribute conditions on the element type named in the condition attribute:
  - the parameter element specifies the name of an attribute. If no modifier element is present, the specified attribute is simply tested for presence (Boolean test)
  - a modifier element is used to specify a value for which the attribute is tested

```
<step remap="dcolumn" condition="doc">
  <constructorsynopsis condition="colspec">
    <methodparam>
      <parameter>align</parameter>
      <modifier>char</modifier>
    </methodparam>
  </constructorsynopsis>
</step>
```

In the example above, a colspec element anywhere in the document with an align attribute equal to "char" will cause the dcolumn package to be included automatically (the package handles decimal-column alignment).

A special case involves the use of the funcparams subelement instead of the parameter attribute, to specify that an IDREF attribute must be

checked for the *type* of element it refers to.

```
<step condition="doc" remap="fmtcount">
  <constructorsynopsis condition="xref">
    <methodparam>
      <funcparams>linkend</funcparams>
      <modifier>varlistentry</modifier>
    </methodparam>
    <methodparam>
      <funcparams>linkend</funcparams>
      <modifier>listitem</modifier>
    </methodparam>
  </constructorsynopsis>
</step>
```

In this example, the fmtcount package will be included if there is an `xref` element anywhere in the documentation with a `linkend` attribute which points at a `varlistentry` or `listitem` element — that is, the `xml:id` attribute whose value matches the `linkend` value is on such an element type (the fmtcount package enables ordinal counting, needed when making a reference to an item in a list that is not numbered).

2.3.2.3 Adding extra code before or after a package

Each `step` may also contain one or more `constraintdef` elements containing `cmdsynopsis` elements containing `command` elements to hold LaTeX code to be inserted, in exactly the same format as shown in section 2.3 on page 17.

```
<step remap="apacite" condition="doc">
  <constraintdef>
    <cmdsynopsis>
      <command>\AtBeginDocument{\edef\ApaciteRestoreAtCode%
    {\catcode‘@=\the\catcode‘@\relax}}</command>
    </cmdsynopsis>
  </constraintdef>
</step>
```

If this step is given in the `"prepackage"` section, the code is inserted *before* the package is included; if the step is given in the `"postpackage"` section, the code is inserted *after* the package is included.

A package listed in this file can be given default options by specifying them in the `role` attribute of the `step` element. It is then unnecessary

to specify them additionally in the main document (although it won't matter, as they are checked for duplication).

2.3.3 Defining commands required for documentation setup

Documenting classes or packages will often require additional commands to be defined in order to set up special counters or lengths, establish conditions, or create new macros to be used in your documentation.

This is quite different from needing to issue standard LaTeX commands in order to set existing standard LaTeX values, such as `\setlength{\parskip}{5mm}`. *That kind of adjustment is dealt with in section 2.3.4 on page 29.*

Commands to be defined for the documentation to work must go in `cmysynopsis` elements in the `constraintdef` element that has the `xml:id` value of `"docpackages"`, *after* the end of the `segmentedlist` of packages.

The commands are specified with the command name (control sequence) or environment name *separately* from the definition: this allows a structure to be imposed which enables the identification and re-use of these specifications.

In giving the commands (control sequences) that you define, you specify just the names, with no backslash; and the definitions you give must not have the outermost set of curly braces. Both the backslash and the curly braces are added by the XSLT2 program when writing the `.dtx` file.

Commands containing an 'at' sign (`@`) in their name or definition are automatically enclosed in `\makeatletter` and `\makeatother` commands.

There are several attributes which specify what you are defining, so that the data is output to the `.dtx` file correctly:

Simple commands: A simple new LaTeX command with a textual expansion is defined with the `command` element holding the name of the command being defined, and an `arg` element holding the definition.

```
<cmdsynopsis>
  <command>LyX</command>
```

```
  <arg>L\kern-.1667em\lower.25em\hbox{Y}\kern-.125emX</arg>
</cmdsynopsis>
```

This creates the definition:

```
\newcommand{\LyX}{L\kern-.1667em\lower.25em\hbox{Y}\kern-.125emX}
```

Renewed commands: If the command is a renewal of an existing command, use a `role` attribute of `"renew"` on the `command` element.

```
<cmdsynopsis>
  <command role="renew">vstrut</command>
  <arg>\vrule height1.2em depth.6667ex width0pt</arg>
</cmdsynopsis>
```

This creates the definition:

```
\renewcommand{\vstrut}{\vrule height1.2em depth.6667ex width0pt}
```

Plain TeX commands: A `\def` command in Plain TeX syntax can be specified with the attribute `xml:lang` set to `"TeX"` on the `command` element.

```
<cmdsynopsis xml:lang="TeX">
  <command>hline</command>
  <arg>\noalign{\ifnum0=‘}\fi
       \@ifnextchar[{\@@hline}{\@@hline[\arrayrulewidth]}</arg>
</cmdsynopsis>
```

This creates the definition:

```
\def\hline{\noalign{\ifnum0=‘}\fi
    \@ifnextchar[{\@@hline}{\@@hline[\arrayrulewidth]}
```

Commands with arguments: If a defined command needs arguments, specify the number of arguments in the `wordsize` attribute of the `arg` element:

```
<cmdsynopsis>
  <command>componentbox</command>
  <arg wordsize="2">\begin{tabular}[m]{@{}|c|@{}}\hline
      \cellcolor{#1}\hbox to1em{\hss%\vrule height1em width0pt
        \raisebox{.2ex}{\ttfamily\tiny#2}\hss}\\\hline
      \end{tabular}</arg>
</cmdsynopsis>
```

This creates the definition:

```
\newcommand{\componentbox}[2]{\begin{tabular}[m]{@{}|c|@{}}\hline
    \cellcolor{#1}\hbox to1em{\hss%\vrule height1em width0pt
        \raisebox{.2ex}{\ttfamily\tiny#2}\hss}\\\hline
    \end{tabular}
```

If a default first argument is required, the value must be provided in the `condition` attribute of the `command` element.

There is no provision in this version of the software for the specification of the extended argument array provided by the `xargs` package.

Counters, lengths, and `\newwrites`: Counters, lengths, and `\newwrite` commands can be defined by using the `remap` attribute set to the value `"counter"`, `"length"` or `"newwrite"` as appropriate. in this case, no `arg` element is required unless a counter or length is to be assigned a default value.

```
<cmdsynopsis>
  <command remap="newwrite">fnotes</command></cmdsynopsis>
```

This creates the definition:

```
\newwrite\fnotes
```

However, if the length or counter needs an initial value, give it in an `arg` element.

```
<cmdsynopsis>
  <command remap="length">revmarg</command>
  <arg>3cm</arg>
</cmdsynopsis>
<cmdsynopsis>
  <command remap="counter">cards</command>
  <arg>42</arg>
</cmdsynopsis>
```

This creates the definition:

```
\newlength{\revmarg}\setlength{\revmarg}{3cm}
```

References to attributes: One specialist use is predefined: assigning the type of document (class or package) to a command:

```
<cmdsynopsis>
  <command>classorpackage</command>
  <arg remap="arch"/>
</cmdsynopsis>
```

This creates the definition:

```
\newcommand{\classorpackage}{...}
```

where [...] is the type of the current document. The `arch` element in this case has no content, but uses the `remap` attribute to specify the name of an attribute (here, `arch`) on the `book` root element. This results in the command `\classorpackage` being set equal to `"class"` or `"package"`; this value is used to provide a value for the entity `&doctype;`. This entity can then be used in shared XML documentation to refer to the current document by type, knowing that it will be correctly translated to the type of document when the `.dtx` file is created.

Environments: The same principle applies to environments as to commands, but there are two arguments: one for the 'before' and one for the 'after':

```
<cmdsynopsis>
  <command remap="environment">panel</command>
  <arg wordsize="1" condition="\relax">\begin{Sbox}%
        \begin{minipage}{3in}%
        \if#1\relax\else\subsubsection*{#1}\fi</arg>
  <arg>\end{minipage}\end{Sbox}%
        \begin{center}\fbox{\theSbox}\end{center}%</arg>
</cmdsynopsis>
```

This creates the definition:

```
\newenvironment{panel}{%
    \begin{Sbox}%
        \begin{minipage}{3in}%
        \if#1\relax\else\subsubsection*{#1}\fi
}{%
    \end{minipage}\end{Sbox}%
        \begin{center}\fbox{\theSbox}\end{center}%
}
```

The controls for the number of arguments and any default argument must go on the first `arg` element. The same rule about setting the `role` attribute to `"renew"` applies as for generating commands.

2.3.4 Additional setup commands

Quite separately from the business of defining new commands (or redefining existing ones) dealt with in section 2.3.3 on page 25, there is also usually a need to issue commands that establish or reset a value needed for the documentation.

Commands that you want implemented every time you use a particular package should go in your `prepost.xml` file, as described in section 2.3.2 on page 21. This section is for commands or settings that only refer to the documentation for the current class or package being written.

These commands go in the third of the types of `constraintdef` element, with the `xml:id` value of `"startdoc"` because they are output at the start of the documentation (*after* the `\begin{document}`).

They follow exactly the same syntax as those in the `prepost.xml` file:

```
<constraintdef xml:id="startdoc">
  <procedure>
    <step>
      <cmdsynopsis>
        <command>\setcounter{tocdepth}{5}</command>
        <command>\setcounter{secnumdepth}{5}</command>
        <command>\def\@@doxdescribe#1#2{\endgroup
   \ifdox@noprint\else\marginpar{\raggedleft
   \textcolor{DarkRed}{\@nameuse{PrintDescribe#1}{#2}}}\fi
   \ifdox@noindex\else\@nameuse{Special#1Index}{#2}\fi
   \endgroup\@esphack\ignorespaces}</command>
      </cmdsynopsis>
    </step>
  </procedure>
</constraintdef>
```

The `\@@doxdescribe` command is an oddity here: it appears not to work if placed in the `prepost.xml` document, where it gets issued in the Preamble; instead it goes here, where it gets issued after the `\begin{document}`.

2.3.5 The Manifest

The fourth and last variant of the `constraintdef` element (with the `xml:id` attribute of `"manifest"`) is very simple. It is a list of the names of any separate files that you want included in the Zip file that the `build` command produces. This means anything other than the `.dtx`, `.ins`, and `.pdf` files that get included automatically.

The content of this element is a `simplelist`, containing `member` elements, one per file:

```
<constraintdef xml:id="manifest">
  <simplelist>
    <member>doctexbook.dtd</member>
    <member>db2dtx.xsl</member>
    <member>db2bibtex.xsl</member>
    <member>prepost.xml</member>
    <member>lppl.xml</member>
    <member>decommentbbl.awk</member>
  </simplelist>
</constraintdef>
```

2.3.6 The README file

One additional output file is produced automatically by the XSLT2 program: the plaintext README file which accompanies all classes and packages, with a brief description of usage and installation, for the benefit of people who cannot or will not read the PDF documentation.

This is generated automatically from the file `readme.xml`, which is a DocBook5 `chapter` document with some changes to the character entities to accommodate plain text. Note that this document does *not* use the `doctexbook.dtd` used for your normal class or package XML document. A sample is included in the `classpack` distribution.

The `readme.xml` file uses the `olink` element type to act as a placeholder for transcluded atomic information from the main document (pending implementation of the proposed DocBook Transclusions method). This element must have a `targetptr` and a `type` attribute specifying (respectively) the name of the element type and the relevant attribute in the main document. For example, to include the name of the class or package, we use:

```
<olink targetptr="book" type="xml:id"/>
```

Two other element types have also had `targetptr` and `type` attributes added: `sect1` and `anchor`. These are used to specify the inclusion of whole sections or fragments of the main document, such as the Abstract or the Copyright.

The text is reformatted in plain text, omitting all markup. Only a few element types have been implemented for this in this version: see the ancillary XSLT2 routines in `db2plaintext.xsl` for details.

The resulting `README` file includes the Abstract from your class or package XML document as the first section. The `db2plaintext.xsl` program uses a template called `normtext` to reformat text. This handles the conversion of entities which occur in your Abstract (those declared in `doctexbook.dtd`): by default, `&TeX;`, `&LaTeX;`, and ` ` are catered for, but if you use any others, you must modify the code in this template to deal with them, using a nested `replace()` function.

normtext

replace()

3 Using *ClassPack*

The body of your docmentation is held in a `part` element with the `xml:id` attribute set to `"doc"`.

The tag-set of *DocBook* is very large, and only a part of it is needed for this purpose, although support for additional elements is easily added in the XSLT2 program.

The following sections describe the elements that are currently supported, for the hierarchical structure (chapters, sections, subsections, etc); the block-level structure (tables, figures, lists, etc; what *DocBook* refers to as the 'pool') and for the inline markup (element types in mixed content, used mostly in paragraph-like situations).

3.1 Hierarchical markup

The outline top-level structure is described in section 1.2 on page 7 and section 2.1 on page 11.

The `part`s do not have any title or direct textual content themselves: they just act as containers to keep the user documentation separate from the documented code and any other files that may be stored and extracted.

Within a `part`, the major subdivision is the `chapter`, which translates to a `\section` in the `.dtx` file. You should use the `chapter` element as your major structural division. Each part (user documentation, documented code, and additional files) must have at least one chapter.

Within chapters, the sections, subsections, and lower structural divisions are identified with `sect1`, `sect2`, and so on. You can use as many or as few of these as you feel you need to organise your writing. In the documented code, it is a good idea to modularise the class or package, so that you can describe each part of it in a logical and consistent manner.

The nested arrangement of chapters containing sections conaining subsections should already be familiar to LaTeX users, although LaTeX itself only uses headings as separators, and has no physical 'containment' of the hierarchical divisions of a document in the way that it does for the block-level structures (environments).

Each of these hierarchical divisions must have a `title` element (see '**title**', the first item in the list in section 3.2), and can also have an `xml:id` attribute to act as a target (like a LaTeX `\label`) for cross-referencing (see '**xref**', the last item in the list in section 3.3 on page 37).

```
<part xml:id="doc">...
  <chapter xml:id="ui">
    <title>User interface</title>
    ...
    <sect1>
      <title>Font selection</title>
      ...
    </sect1>
    <sect1 xml:id="margins">
      <title>Margins and spacing</title>
      ...
    </sect1>
  </chapter>
  ...
</part>
```

3.2 Structural markup (block-level elements)

Within chapters, sections, subsections, etc, you can have any arrangement or mixture of paragraphs, tables, lists, figures, quotations, code samples, and other conventional structures that will be familiar to you from LaTeX environments.

The only requirement is that each hierarchical division must start with a title, and must contain at least one other structural component. Supported element types are:

title: a title, used in all chapters, [sub]sections, figures, and tables (where it equates to a caption); and also optionally in lists, sidebars, and other block-level element types.

para: for normal paragraphs.

itemizedlist: for bulleted lists, containing `listitem`s which contain paragraphs.

orderedlist: for numbered lists; the structure is identical to an `itemizedlist`.

variablelist: for description lists like this one (the `term` element in each `varlistentry` holds the reference term; the descriptive part is in the same `listitem` structure as for itemized and bulleted lists.

simplelist: for plain unnumbered, unbulleted lists; each item goes in a `member` element.

programlisting: for listings of code: use without attributes in the Code section. In the Documentation section, the basic style is in `\small` type, black, `\ttfamily`, and the following attributes control the appearance:

`wordsize`	either a size command or a pointsize/baseline like `8/9`
`language`	LaTeX (default), `DocBook`, `bash`, or another language supported by the listings package
`arch`	`framed` will box the listing
`remap`	LaTeX styling commands for tokens to emphasise
`annotations`	comma-separated list of tokens to emphasise

figure: for Figures, containing a `caption` and a `media` element.

table: for Tables; the structure is explained in detail in **??** on page ??.

sidebar: for sidebars.

warning: for warnings.

3.2.1 Ancillary files documented inline

These are files which you want extracted at installation time, which you describe *and* show in the user documentation (the `doc` part).

Files which you want extracted which are not *documented or shown in the user documentation must go in the `files` part*
(see section 1.3.3 on page 9).

3.2.2 Bibliography

If bibliographic citations and reference is required, the references themselves must be stored in a `bibliography` element immediately

after the last `chapter` element in *either* the 'doc' part *or* the 'code' part. This must contain a `biblioentry` element for each entry you wish to cite (for how to cite, see '**`biblioref`**', the first item in the list in section 3.3 on the next page).

```
<bibliography xreflabel="apacite" label="apacite">
  <biblioentry xml:id="tb97" xreflabel="article">
    <biblioset>
      <author>
        <personname>
          <surname>Flynn</surname>
          <firstname>Peter</firstname>
        </personname>
      </author>
      <title>Typographers' Inn</title>
      <subtitle>Where have all the flowers gone?</subtitle>
    </biblioset>
    <artpagenums>21-22</artpagenums>
    <title>TUGboat</title>
    <volumenum>31</volumenum>
    <issuenum>1</issuenum>
    <date YYYY-MM-DD="2010"/>
  </biblioentry>
</bibliography>
```

The `bibliography` element must have a `label` attribute giving the name of the BibTeX style file to use (without the `.bst` filetype). The `apacite` style is recommended.

If the specified style requires a LaTeX style package for formatting (often called by the same name, eg `apacite`, `natbib`, `chicago`, etc), this must be given in an `xreflabel` attribute (without the `.sty` filetype).

There may also be an `xlink:href` attribute giving the name of the BibTeX file (without the `.bib` filetype) to which the references should be written: the default is the name of the class or package itself (as defined in '**`xml:id`**', the first item in the list in section 2.1 on page 11).

Each `biblioentry` element must have both an `xml:id` attribute by which it can be cited with the `biblioref` element; and a `type` attribute classifying it with one of the standard BibTeX document types (article, book, incollection, etc)

3.3 Inline markup (elements in mixed content)

`biblioref:` a citation (bibliographic reference) to an item in the Bibliography; the `linkend` attribute must be the value of the `xml:id` of a `biblioentry` element (see section 3.2.2 on page 34); this value is passed to a `\cite` command.

`citetitle:` the title of a document being mentioned; usually formatted in italics or quotes; may be empty, with a `linkend` attribute pointing to an entry in the Bibliography (as for `biblioref`), in which case the title is automatically extracted and formatted, or passed to a `\citefield` command.

`code:` a fragment of computer or data code, formatted in monospace type.

`emphasis:` emphasis according to style, usually italics.

`exceptionname:` used for the keywords of RFC2119 in formal admonishments.

`filename:` name of a file, a full filepath, or just a part of the name (eg a filetype).

`firstterm:` the defining instance of a specialist term; this may or may not actually be the first occurrence.

`footnote:` a footnote; contains a paragraph.

`foreignphrase:` for foreign-language expressions; identify the language with the `xml:lang` attribute if the phrase is long enough to need the babel package.

`guibutton:` represents a GUI Button.

`guilabel:` represents a GUI Label.

`guimenu:` represents a GUI **Menu**.

`guimenuitem:` represents a GUI *Menu item*.

`guisubmenu:` represents a GUI Sub-menu item.

`literal:` marks a `literal string` on which no interpretation is to be performed (markup characters like backslashes and curly braces will not be escaped; the `xml:lang` attribute can be set to the name of the language (eg `"TeX"`, `"LaTeX"`, etc.

phrase: marks a phrase used as-is (ie not a quote from anyone in particular) by enclosing it in quotation marks.

productname: marks a product or program name, eg *Emacs*.

quote: marks a quote from someone by putting it in quotation marks.

replaceable: identifies text, commands, or keywords to be typed, for which the user must substitute a meaningful value, eg *password* (in italics).

systemitem: identifies generic computer-related strings such as system commands, hostnames, Regular Expressions, etc which need to be printed in monospace to eliminate any confusion over 1/l/I, 0/1/I, etc.

type: marks a span for which special typographical treatment is needed. The `role` attribute must be set to 'font' and the `remap` attribute must be set to the NFSS2e three-character fontname.

uri: marks a URI (formats it with the `\url` macro).

wordasword: marks a word that is being used as itself (usually for purposes of clarification), so it goes in quotation marks.

xref: a cross-reference using the `linkend` attribute to point at some other part of the document, which must have the matching `xml:id` value; the mechanism is identical to LaTeX's `\label...\ref`.

In addition, there are six special-purpose element types used for functional documentation, that create the special dox package commands for adding LaTeX (and XML) terms to the index, and highlighting them in the left-hand margin:

classname: a LaTeX document class name like article

command: a LaTeX or other computer command, such as `\parskip`; the backslash is added automatically for the default case of LaTeX; other languages require the `xml:lang` attribute giving the name of the language

envar: a LaTeX environment name like `enumerate`

option: a LaTeX option to a class, package, or command, like **a4paper**

package: a LaTeX package name like fancybox

tag: an XML element, attribute, attribute value, or entity name: the type is specified in the `class` attribute and is one of the

predetermined list provided automatically by *DocBook* (so your XML editor will guide you)

3.4 Producing your class or package

The XSLT2 program generates a number of output files, principally the `.dtx` and `.ins` files which are the package or class itself. A third output is a `build` file, which is a *bash* shell script customised for the production of the class or package you are writing. A fourth is the `MANIFEST` file, used for zipping everything up for distribution,

You should therefore keep each class or package development in a separate directory, otherwise the `build` file generated by one will overwrite that generated by others.

```
#! /bin/bash
#
# Bourne shell script to build the class file and documentation
# Note the following line is wrapped here to fit the width
java -jar /usr/local/saxon/saxon9he.jar \
     -o:classpack.dtx classpack.xml \
     /home/peter/texmf/dev/db2dtx.xsl \
     processor=/usr/local/saxon/saxon9he.jar \
     appdir=/home/peter/texmf/dev \
     cpdir=/home/peter/texmf/dev
yes|latex classpack.ins
pdflatex classpack.dtx
bibtex classpack
awk -f /home/peter/texmf/dev/decommentbbl.awk classpack.bbl >classpack.bdc
mv classpack.bdc classpack.bbl
pdflatex classpack.dtx
makeindex -s gind.ist -o classpack.ind classpack.idx
makeindex -s gglo.ist -o classpack.gls classpack.glo
pdflatex classpack.dtx
echo Copying files into dev tree...
mkdir -p doc/latex/classpack
mkdir -p source/latex/classpack
mkdir -p tex/latex/classpack
cp README MANIFEST classpack.pdf doc/latex/classpack
cp classpack.dtx classpack.ins source/latex/classpack
cp classpack.cls tex/latex/classpack
cp db2bibtex.xsl source/latex/classpack
cp db2dtx.xsl source/latex/classpack
cp db2plaintext.xsl source/latex/classpack
```

```
cp decommentbbl.awk source/latex/classpack
cp doctexbook.dtd source/latex/classpack
cp lppl.xml source/latex/classpack
cp prepost.xml source/latex/classpack
cp readme.xml source/latex/classpack
echo Zipping up files from dev tree...
zip -r --exclude=*.svn* --exclude=*.DS_Store* \
    classpack-0.73.tds.zip doc/latex/classpack \
    source/latex/classpack tex/latex/classpack
echo Installing working copy...
unzip -o -d ~/texmf classpack-0.73.tds.zip
```

Because this file will not exist the very first time you process a new class or package, you will need to type that first (`java`) command by hand. The arguments are:

1. {⟨*jar*⟩} the location of your copy of the Saxon XSLT2 processor, a `.jar` file;
2. **`-o:`** the name of the `.dtx` file you are producing;
3. the name of the XML file you are processing;
4. the full path to the DB2DTX program;
5. the location of your copy of the Saxon XSLT2 processor (again) as the `processor` parameter;
6. the directory you use for this class or package as the `appdir`;
7. the directory where your copy of the XSLT2 program is stored as the `cpdir` parameter (along with the DTD, `prepost.xml`, `readme.xml`, `db2plaintext.xsl`, and `lppl.xml` files) .

For subsequent runs, you just type `./build` and the values and parameters will be re-used automatically. If you ever need to run the XSLT2 process by itself, use the command `grep java build | bash`

The remainder of the `build` tests the extraction of the class or package, and compiles the full documentation in the standard sequence, including any bibliography, index, or glossary.

The use of the `decommentbbl.awk` script on the BibTEX output is to defeat the use of terminal percent comment characters, which upset the ltxdoc package because of the special use of that character there.

The final stage is to create a Zip file of the class or package, which is placed in the current working directory and then unzipped into your

personal TEX tree.

3.5 Maintaining your class or package

The things to control each time you make an update are:

1. on the `book` root element, update the `version` and `revision` attributes
2. add a new `revision` element in the Revision History, setting the `version` attribute to the compound of the version and revision specified above, and setting the `date` subelement's `conformance` attribute to today's date in ISO format
3. after processing the document once, set the `book` root element's `security` attribute to the checksum displayed by LATEX
4. process the document again (run the `build` script again) — the security checksum should now match

4 The *db2dtx* program

While the core of your class or package is the *DocBook* XML document, the core of the *classpack* system is the program that turns your XML into `.dtx` and `.ins` files for distribution as combined code and documentation.

The *db2dtx* program is written in XSLT2, a declarative language for processing XML. It consists of a set of templates, each of which matches a pattern in the XML document, usually an element type, or an element type in a particular position or with a particular attribute value or subelement.

For example, there is a template which matches the `biblioref` element whenever it occurs. This puts three things into the output:

1. the command `\cite{` with its opening curly brace;
2. the value of the link to the bibliographic entry;
3. the closing `}` curly brace.

```
<xsl:template match="db:biblioref">
  <xsl:text>\cite{</xsl:text>
  <xsl:value-of select="@linkend"/>
  <xsl:text>}</xsl:text>
</xsl:template>
```

The advantage of using a declarative language is that you don't need to know when and where each element will occur: XSLT2 will find them as they come up in processing, and apply the template when it happens. It's basically a case of 'when you see one of *these*, do *this*'.

The next few sections of this document describe each part of the program and how it produces your class or package files.

4.1 XML Declaration and Namespace declarations

The program starts in the usual way with the XML Declaration and the `xsl:stylesheet` start-tag with the Namespace declarations.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:db="http://docbook.org/ns/docbook"
                xmlns:xlink="http://www.w3.org/1999/xlink"
```

```
                version="2.0">
```

Note that this is an XSLT2 program and requires an XSLT2 processor.

```
<!-- db2dtx.xsl
     XSL script to transform DocBook5 documentation and code of a
     LaTeX package or class file into a DocTeX (.dtx and .ins)
     distribution.
     Full processing command chain is output to file 'build'
     Note this requires an XSLT2 processor (eg Saxon9 or above)
-->
```

We identify the version of the program, output methods, parameters, and the single `xsl:include` file: the `db2bibtex.xsl` module for handling bibliographic formatting.

```
<xsl:variable name="thisversion">
  <xsl:text>14.7</xsl:text>
</xsl:variable>

<xsl:output method="text"/>
<xsl:output method="text" name="textFormat"/>

<xsl:include href="db2bibtex.xsl"/>
<xsl:include href="db2plaintext.xsl"/>
```

This is incomplete: the remainder of the program is not yet included here.

5 Service commands

As ClassPack itself is not a document class or package *per se*, there is no operating code.

However, there are some ancillary commands commonly used in documentation which should be expected by authors of classes and packages using ClassPack.

This section therefore implements `classpack.sty`, which gets invoked automatically via its entry in `prepost.xml`.

`IndexColumns` The doctex package uses a default three-column index, which is too narrow for most purposes. We therefore make the index in two columns, and space them slightly farther apart.

```
\setcounter{IndexColumns}{2}
\setlength{\columnsep}{3pc}
```

5.1 TEX and other logos

TEX and LATEX are defined in the LATEX kernel, but most of the others are not. The following definitions are taken from the ltugboat package, used for typesetting the TUGboat journal.

`\ConTeXt` ConTEXt is a typography and typesetting system meant to provide users easy and consistent access to advanced typographical control (Anon, n.d.).

```
\def\ConTeXt{C\kern-.0333emon\-\kern-.0667em\TeX\kern-.0333emt}
```

References

Anon. (n.d.). ConTEXt. *Wikipedia*. Retrieved 27 March 2013, from http://en.wikipedia.org/wiki/ConTeXt

A The XML vocabulary

There are currently no changes to the *DocBook* element structure.

`book` The DTD is a driver implementing a number of entity declarations to ease the transformation to LaTeX.

```
<!ENTITY % db5dtd SYSTEM "/dtds/docbook/docbook-5.0/dtd/docbook.dtd">
<!ATTLIST date YYYY-MM-DD CDATA #IMPLIED>
<!ATTLIST blockquote units CDATA #IMPLIED
                     begin CDATA #IMPLIED
                     end CDATA #IMPLIED>
<!ATTLIST quote units CDATA #IMPLIED
                begin CDATA #IMPLIED
                end CDATA #IMPLIED>
<!ELEMENT html:form EMPTY>
<!ENTITY ampers "\&#38;#38;">
<!ENTITY BiBTeX "\BibTeX{}">
<!ENTITY BibTeX "\BibTeX{}">
<!ENTITY BIBTeX "\BibTeX{}">
<!ENTITY ConTeXt "\ConTeXt{}">
<!ENTITY LaTeX "\LaTeX{}">
<!ENTITY LaTeX2e "\LaTeXe{}">
<!ENTITY XeTeX "\XeTeX{}">
<!ENTITY LyX "\LyX{}">
<!ENTITY METAFONT "\MF{}">
<!ENTITY METAPOST "\MP{}">
<!ENTITY TeX "\TeX{}">
<!ENTITY bsol "{\texttt{\textbackslash}}">
<!ENTITY date "\filedate{}">
<!ENTITY degree "\textdegree{}">
<!ENTITY doctype "\classorpackage{}">
<!ENTITY filler "\hfil{}">
<!ENTITY frac12 "\nicefrac12">
<!ENTITY frac13 "\nicefrac13">
<!ENTITY frac23 "\nicefrac23">
<!ENTITY hellip "\dots{}">
<!ENTITY mdash "~--- ">
<!ENTITY mldr "\dotfill{}">
```

```
<!ENTITY nbsp "~">
<!ENTITY ndash "--">
<!ENTITY percnt "\&#x0025;">
<!ENTITY square "\raisebox{-1pt}{\Square}">
<!ENTITY thinsp "\thinspace{}">
<!ENTITY times "×">
<!ENTITY specialUuml '{\normalfont\"{\fontfamily{cdr}\selectfont U}}'
<!ENTITY verbar "\menusep{}">
<!ENTITY version "\fileversion{}">
<!-- call the main DTD --> %db5dtd;
```

B Reusable XML

In the last item in the list in section 1 on page 5, I said that one of the benefits of using XML for software generation and documentation was the re-usability of the data. Here are a couple of simple examples.

```
$ lxprintf -e productname "%s\n" . classpack.xml |\
  sort | uniq -c | sort -k 1nr
```

Checking that all element types have been described!

C The LaTeX Project Public License

> Everyone is allowed to distribute verbatim copies of this license document, but modification of it is not allowed.

C.1 Preamble

The LaTeX Project Public License (LPPL) is the primary license under which the LaTeX kernel and the base LaTeX packages are distributed.

You may use this license for any work of which you hold the copyright and which you wish to distribute. This license may be particularly suitable if your work is TeX-related (such as a LaTeX package), but it is written in such a way that you can use it even if your work is unrelated to TeX.

The section *Whether and How to Distribute Works under This License*, below, gives instructions, examples, and recommendations for authors who are considering distributing their works under this license.

This license gives conditions under which a work may be distributed and modified, as well as conditions under which modified versions of that work may be distributed.

We, the LaTeX3 Project, believe that the conditions below give you the freedom to make and distribute modified versions of your work that conform with whatever technical specifications you wish while maintaining the availability, integrity, and reliability of that work. If you do not see how to achieve your goal while meeting these conditions, then read the document `cfgguide.tex` and `modguide.tex` in the base LaTeX distribution for suggestions.

C.2 Definitions

In this license document the following terms are used:

Work: Any work being distributed under this License.

Derived Work: Any work that under any applicable law is derived from the Work.

Modification: Any procedure that produces a Derived Work under any applicable law — for example, the production of a file containing an original file associated with the Work or a significant portion of such a file, either verbatim or with modifications and/or translated into another language.

Modify: To apply any procedure that produces a Derived Work under any applicable law.

Distribution: Making copies of the Work available from one person to another, in whole or in part. Distribution includes (but is not limited to) making any electronic components of the Work accessible by file transfer protocols such as FTP or HTTP or by shared file systems such as Sun's Network File System (NFS).

Compiled Work: A version of the Work that has been processed into a form where it is directly usable on a computer system. This processing may include using installation facilities provided by the Work, transformations of the Work, copying of components of the Work, or other activities. Note that modification of any installation facilities provided by the Work constitutes modification of the Work.

Current Maintainer: A person or persons nominated as such within the Work. If there is no such explicit nomination then it is the 'Copyright Holder' under any applicable law.

Base Interpreter: A program or process that is normally needed for running or interpreting a part or the whole of the Work.

A Base Interpreter may depend on external components but these are not considered part of the Base Interpreter provided that each external component clearly identifies itself whenever it is used interactively. Unless explicitly specified when applying the license to the Work, the only applicable Base Interpreter is a 'LaTeX-Format' or in the case of files belonging to the 'LaTeX-format' a program implementing the 'TeX language'.

C.3 Conditions on Distribution and Modification

1. Activities other than distribution and/or modification of the Work are not covered by this license; they are outside its scope. In particular, the act of running the Work is not restricted and no requirements are made concerning any offers of support for the Work.
2. You may distribute a complete, unmodified copy of the Work as you received it. Distribution of only part of the Work is considered modification of the Work, and no right to distribute such a Derived Work may be assumed under the terms of this clause.
3. You may distribute a Compiled Work that has been generated from a complete, unmodified copy of the Work as distributed under Clause item 2 above above, as long as that Compiled Work is distributed in such a way that the recipients may install the Compiled Work on their system exactly as it would have been installed if they generated a Compiled Work directly from the Work.
4. If you are the Current Maintainer of the Work, you may, without restriction, modify the Work, thus creating a Derived Work. You may also distribute the Derived Work without restriction, including Compiled Works generated from the Derived Work. Derived Works distributed in this manner by the Current Maintainer are considered to be updated versions of the Work.
5. If you are not the Current Maintainer of the Work, you may modify your copy of the Work, thus creating a Derived Work based on the Work, and compile this Derived Work, thus creating a Compiled Work based on the Derived Work.
6. If you are not the Current Maintainer of the Work, you may distribute a Derived Work provided the following conditions are met for every component of the Work unless that component clearly states in the copyright notice that it is exempt from that condition. Only the Current Maintainer is allowed to add such statements of exemption to a component of the Work.

   (a) If a component of this Derived Work can be a direct

replacement for a component of the Work when that component is used with the Base Interpreter, then, wherever this component of the Work identifies itself to the user when used interactively with that Base Interpreter, the replacement component of this Derived Work clearly and unambiguously identifies itself as a modified version of this component to the user when used interactively with that Base Interpreter.

(b) Every component of the Derived Work contains prominent notices detailing the nature of the changes to that component, or a prominent reference to another file that is distributed as part of the Derived Work and that contains a complete and accurate log of the changes.

(c) No information in the Derived Work implies that any persons, including (but not limited to) the authors of the original version of the Work, provide any support, including (but not limited to) the reporting and handling of errors, to recipients of the Derived Work unless those persons have stated explicitly that they do provide such support for the Derived Work.

(d) You distribute at least one of the following with the Derived Work:

i. A complete, unmodified copy of the Work; if your distribution of a modified component is made by offering access to copy the modified component from a designated place, then offering equivalent access to copy the Work from the same or some similar place meets this condition, even though third parties are not compelled to copy the Work along with the modified component;

ii. Information that is sufficient to obtain a complete, unmodified copy of the Work.

7. If you are not the Current Maintainer of the Work, you may distribute a Compiled Work generated from a Derived Work, as long as the Derived Work is distributed to all recipients of the Compiled Work, and as long as the conditions of Clause item 6 above, above, are met with regard to the Derived Work.

8. The conditions above are not intended to prohibit, and hence do not apply to, the modification, by any method, of any component so that it becomes identical to an updated version of that component of the Work as it is distributed by the Current Maintainer under Clause item 4 above, above.

9. Distribution of the Work or any Derived Work in an alternative format, where the Work or that Derived Work (in whole or in part) is then produced by applying some process to that format, does not relax or nullify any sections of this license as they pertain to the results of applying that process.

10. (a) A Derived Work may be distributed under a different license provided that license itself honors the conditions listed in Clause item 6 above above, in regard to the Work, though it does not have to honor the rest of the conditions in this license.

    (b) If a Derived Work is distributed under a different license, that Derived Work must provide sufficient documentation as part of itself to allow each recipient of that Derived Work to honor the restrictions in Clause item 6 above above, concerning changes from the Work.

11. This license places no restrictions on works that are unrelated to the Work, nor does this license place any restrictions on aggregating such works with the Work by any means.

12. Nothing in this license is intended to, or may be used to, prevent complete compliance by all parties with all applicable laws.

C.4 No Warranty

There is no warranty for the Work. Except when otherwise stated in writing, the Copyright Holder provides the Work ‘as is’, without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the Work is with you. Should the Work prove defective, you assume the

cost of all necessary servicing, repair, or correction.

In no event unless required by applicable law or agreed to in writing will The Copyright Holder, or any author named in the components of the Work, or any other party who may distribute and/or modify the Work as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of any use of the Work or out of inability to use the Work (including, but not limited to, loss of data, data being rendered inaccurate, or losses sustained by anyone as a result of any failure of the Work to operate with any other programs), even if the Copyright Holder or said author or said other party has been advised of the possibility of such damages.

C.5 Maintenance of The Work

The Work has the status ‘author-maintained’ if the Copyright Holder explicitly and prominently states near the primary copyright notice in the Work that the Work can only be maintained by the Copyright Holder or simply that it is ‘author-maintained’.

The Work has the status ‘maintained’ if there is a Current Maintainer who has indicated in the Work that they are willing to receive error reports for the Work (for example, by supplying a valid e-mail address). It is not required for the Current Maintainer to acknowledge or act upon these error reports.

The Work changes from status ‘maintained’ to ‘unmaintained’ if there is no Current Maintainer, or the person stated to be Current Maintainer of the work cannot be reached through the indicated means of communication for a period of six months, and there are no other significant signs of active maintenance.

You can become the Current Maintainer of the Work by agreement with any existing Current Maintainer to take over this role.

If the Work is unmaintained, you can become the Current Maintainer of the Work through the following steps:

1. Make a reasonable attempt to trace the Current Maintainer (and the Copyright Holder, if the two differ) through the means of an

Internet or similar search.

2. If this search is successful, then enquire whether the Work is still maintained.
   (a) If it is being maintained, then ask the Current Maintainer to update their communication data within one month.
   (b) If the search is unsuccessful or no action to resume active maintenance is taken by the Current Maintainer, then announce within the pertinent community your intention to take over maintenance. (If the Work is a LaTeX work, this could be done, for example, by posting to `news:comp.text.tex`.)
3. (a) If the Current Maintainer is reachable and agrees to pass maintenance of the Work to you, then this takes effect immediately upon announcement.
   (b) If the Current Maintainer is not reachable and the Copyright Holder agrees that maintenance of the Work be passed to you, then this takes effect immediately upon announcement.
4. If you make an 'intention announcement' as described in item 2b above above and after three months your intention is challenged neither by the Current Maintainer nor by the Copyright Holder nor by other people, then you may arrange for the Work to be changed so as to name you as the (new) Current Maintainer.
5. If the previously unreachable Current Maintainer becomes reachable once more within three months of a change completed under the terms of item 3b above or item 4 above, then that Current Maintainer must become or remain the Current Maintainer upon request provided they then update their communication data within one month.

A change in the Current Maintainer does not, of itself, alter the fact that the Work is distributed under the LPPL license.

If you become the Current Maintainer of the Work, you should immediately provide, within the Work, a prominent and unambiguous statement of your status as Current Maintainer. You should also announce your new status to the same pertinent community as in item 2b above above.

C.6 Whether and How to Distribute Works under This License

This section contains important instructions, examples, and recommendations for authors who are considering distributing their works under this license. These authors are addressed as ‘you’ in this section.

C.6.1 Choosing This License or Another License

If for any part of your work you want or need to use *distribution* conditions that differ significantly from those in this license, then do not refer to this license anywhere in your work but, instead, distribute your work under a different license. You may use the text of this license as a model for your own license, but your license should not refer to the LPPL or otherwise give the impression that your work is distributed under the LPPL.

The document `modguide.tex` in the base LaTeX distribution explains the motivation behind the conditions of this license. It explains, for example, why distributing LaTeX under the GNU General Public License (GPL) was considered inappropriate. Even if your work is unrelated to LaTeX, the discussion in `modguide.tex` may still be relevant, and authors intending to distribute their works under any license are encouraged to read it.

C.6.2 A Recommendation on Modification Without Distribution

It is wise never to modify a component of the Work, even for your own personal use, without also meeting the above conditions for distributing the modified component. While you might intend that such modifications will never be distributed, often this will happen by accident — you may forget that you have modified that component; or it may not occur to you when allowing others to access the modified version that you are thus distributing it and violating the conditions of this license in ways that could have legal implications and, worse,

cause problems for the community. It is therefore usually in your best interest to keep your copy of the Work identical with the public one. Many works provide ways to control the behavior of that work without altering any of its licensed components.

C.6.3 How to Use This License

To use this license, place in each of the components of your work both an explicit copyright notice including your name and the year the work was authored and/or last substantially modified. Include also a statement that the distribution and/or modification of that component is constrained by the conditions in this license.

Here is an example of such a notice and statement:

```
%%% pig.dtx
%%% Copyright 2005 M. Y. Name
%%
%% This work may be distributed and/or modified under the
%% conditions of the LaTeX Project Public License, either version 1.3
%% of this license or (at your option) any later version.
%% The latest version of this license is in
%%   http://www.latex-project.org/lppl.txt
%% and version 1.3 or later is part of all distributions of LaTeX
%% version 2005/12/01 or later.
%%
%% This work has the LPPL maintenance status ‘maintained’.
%%
%% The Current Maintainer of this work is M. Y. Name.
%%
%% This work consists of the files pig.dtx and pig.ins
%% and the derived file pig.sty.
```

Given such a notice and statement in a file, the conditions given in this license document would apply, with the ‘Work’ referring to the three files `pig.dtx`, `pig.ins`, and `pig.sty` (the last being generated from `pig.dtx` using `pig.ins`), the ‘Base Interpreter’ referring to any ‘LaTeX-Format’, and both ‘Copyright Holder’ and ‘Current Maintainer’ referring to the person M. Y. Name.

If you do not want the Maintenance section of LPPL to apply to your Work, change ‘maintained’ above into ‘author-maintained’. However, we recommend that you use ‘maintained’ as the Maintenance section was added in order to ensure that your Work remains useful to the community even when you can no longer maintain and support it yourself.

C.6.4 Derived Works That Are Not Replacements

Several clauses of the LPPL specify means to provide reliability and stability for the user community. They therefore concern themselves with the case that a Derived Work is intended to be used as a (compatible or incompatible) replacement of the original Work. If this is not the case (e.g., if a few lines of code are reused for a completely different task), then clauses 6b and 6d shall not apply.

C.6.5 Important Recommendations

C.6.5.1 Defining What Constitutes the Work The LPPL requires that distributions of the Work contain all the files of the Work. It is therefore important that you provide a way for the licensee to determine which files constitute the Work. This could, for example, be achieved by explicitly listing all the files of the Work near the copyright notice of each file or by using a line such as:

```
%% This work consists of all files listed in manifest.txt.
```

in that place. In the absence of an unequivocal list it might be impossible for the licensee to determine what is considered by you to comprise the Work and, in such a case, the licensee would be entitled to make reasonable conjectures as to which files comprise the Work.

Change History

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.