How to develop your own document class — our experience

Niall Mansfield
UIT Cambridge Ltd.
PO Box 145
Cambridge, England
`tug08 (at) uit dot co dot uk`

Abstract

We recently started re-using LaTeX for large documents — professional computing books — and had to convert an old (1987) LaTeX 2.09 custom class to work with LaTeX 2ε. We first tried converting it to a stand-alone `.cls` file, which the documentation seemed to suggest is the thing to do, but we failed miserably. We then tried the alternative approach of writing an "add-on" `.sty` file for the standard `book.cls`. This was straightforward, and much easier than expected. The resulting style is much shorter, and we can use most standard packages to add extra features with no effort.

This paper describes our experience and the lessons and techniques we learned, which we hope will encourage more people to write their own styles or classes.

1 Where we started from

Years ago I wrote a book *The Joy of X* [1], about the X window system. It was in an unusual format called STOP [5], as enhanced by Weiss [6], summarized graphically in Figure 1. In 2008 I wanted to write another book in the same format [2, 3]. It has several interesting features that make it excellent for technical books, although those details are not relevant here. Suffice it to say that STOP required us to change how parts, chapters, sections and sub-sections are handled, and to provide extra sectional units at the beginning and end of each chapter. We also had to provide a summary table of contents, and for each chapter a per-chapter table of contents (TOC) on the first page of the chapter, and use PostScript fonts, which in 1987 was a non-trivial task.

Back in 1987 a colleague of mine, Paul Davis, very kindly wrote the necessary style file for this format, and it worked very well. However, in the meantime the world had moved on from LaTeX 2.09 to LaTeX 2ε. The challenge was to provide the functionality of the old style, but under LaTeX 2ε.

2 First attempt — failure

Where do you start when developing a new style or class? The document *LaTeX 2ε for class and package writers* says:

> if the commands could be used with any document class, then make them a package; and if not, then make them a class.

I took this to mean "We should write a class". I wrongly went one step further, and thought it also meant we should start our own class from scratch.

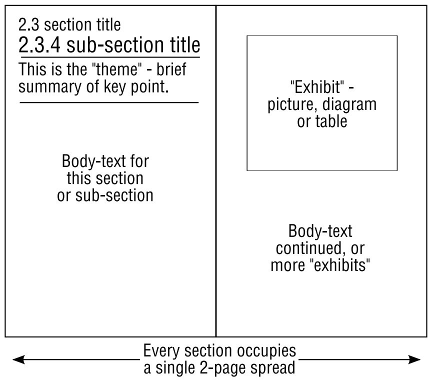
2.3 section title
2.3.4 sub-section title
This is the "theme" - brief summary of key point.
Body-text for this section or sub-section
"Exhibit" - picture, diagram or table
Body-text continued, or more "exhibits"
Every section occupies a single 2-page spread

Figure 1: A STOP sub-section or "module"

(Another reason for thinking this was that at least one major publisher seems to have gone this route.)

In fact the same document continues "There are two major types of class: those like `article`, `report` or `letter`, which are free-standing; and those which are extensions or variations of other classes — for example, the `proc` document class, which is built on the `article` document class." What I ought to have done is started work on an "extension" or "variation" class, but I didn't realize it.

So I tried to convert the old `.sty` to LaTeX 2ε, and failed miserably. (This wasn't surprising, because LaTeX 2.09 style files consisted of plain TeX code, which I have always found *very* difficult to understand.) The end-product was something that almost worked, but had lots of small bugs, and when I fixed one problem, the change caused new problems elsewhere.

Another, equally large, disadvantage of this approach was that even if it had worked, the effort to maintain it would have been huge. None of the standard LaTeX 2ε packages would have worked, so if we needed any changes — even "simple" things like changing the page size — we'd have had to code them by hand ourselves.

> **Lesson A:** whether you call your file a class or a style doesn't matter much — it's just a matter of a name. What *is* important is not to start from scratch, but to build, as far as possible, using existing code.

At this point we almost gave up and considered using Quark Xpress or InDesign. But luckily someone noticed the `minitoc` package, which looked like it might give us exactly what we needed for our per-chapter TOC, if only we could use it. We decided to try again.

3 Second attempt — a short blind alley

We threw away everything we had and started again from scratch. We tried `book.cls` plus `minitoc`. This addressed one of our most difficult requirements — the per-chapter TOC — and did it so well that we were encouraged to persevere, thank goodness.

We copied the `book.cls` file as `uitbook.cls`, and started adding our own modifications to this. After a few days this became messy, especially when bug-fixing: it wasn't obvious which was our code (where the bugs were likely to be) and which was the original code.

> **Lesson B:** in LaTeX, the way to modify standard code is usually not to modify the original file. Instead, extract just the piece that you want to change, save it as *something*`.sty` and modify just that little file. Then do `\usepackage{`*something*`.sty}`.

4 Third attempt — success!

So we started again, leaving `book.cls` unchanged, and created our own file `uitbook.sty` to contain all our changes. The convention we settled on is:

- If something is just a convenience — e.g. a macro that is merely a shorthand to save typing but doesn't add any new functionality — we create a small `.sty` for it, and then `\RequirePackage` that. In this way we can re-use the same convenience tools with other classes.

  For example, we defined about 12 macros for including graphics or verbatim examples of program code, with or without captions, and with captions in the usual place below the figure or alternatively beside the figure (to save vertical space). These don't do anything new, but they all take the same number of arguments in the same order; if a particular variant doesn't actually need them all, we can just leave the irrelevant ones empty. This makes it easy to change a figure from "no caption" to "side caption" or to "normal caption" with a couple of keystrokes. All these macros are in `uit-figures.sty`.

- Where we make substantial changes, e.g. to the sectioning mechanism or to the format of page headings, we include it directly in our file `uitbook.sty`.

To cheer ourselves up after previous failures, we did all the easy bits first. Those included the convenience macros mentioned above, and the dozens of `\RequirePackage` calls to the packages that we needed:

```
caption   chngpage  color     colortbl
courier   crop      endnotes  fancyvrb
framed    geometry  graphicx  helvet
hhline    ifthen    latexsym  layout
makeidx   mathpazo  mcaption  minitoc
nextpage  paralist  relsize   showidx
sidecap   ulem      url       wrapfig
```

At this point things were looking good. We had a style that worked. However, several STOP-specific features were still missing, so that's what we had to implement next.

The document *LaTeX 2ε for class and package writers* describes the boilerplate for a class or package — analogous to telling a C programmer that he needs a `main()` function, and how to use `#include` statements. What it doesn't tell you is how the standard classes work, and the common techniques they use. In the following sections we'll explain the techniques that we came across.

5 The hard bits 1 — over-riding existing functionality

We needed to change the TOC entry for Parts. This is handled in the `\l@part` function in `book.cls`. We copied this function to our `uitbook.sty`, and modified it there. The change involved was only a single line — to use a different font, and insert the word "Part" — but it illustrates a couple of important points:

> **Lesson C:** copying a piece of standard code in LaTeX, and changing your own version of it is a bit like over-riding a method in object-oriented programming. Everything that you haven't changed continues to work as before, but as soon as the relevant function (macro)

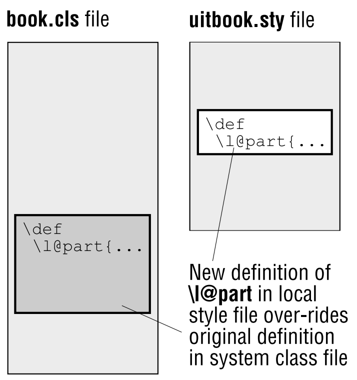
book.cls file
uitbook.sty file
\def \l@part{...
\def \l@part{...
New definition of \l@part in local style file over-rides original definition in system class file

Figure 2: How LaTeX hooks work

is called — `\l@part` in our example — the new code is used instead of the old (Figure 2).

Lesson D: you don't have to understand *everything* to make a change to something relatively small. As long as you change as little as possible, you probably won't break anything else. In our case, even though `\l@part` contains lots of complex stuff, we were confident that our minor format changes would work, because we didn't modify anything else.

By the way, many functions or macros in the package and class files are defined using the TeX primitive `\def`, instead of LaTeX's `\newcommand`. If you redefine an existing command with `\def`, you don't get an error, unlike `\newcommand`'s behavior.

6 The hard bits 2 — LaTeX hooks

The first clue we got about how LaTeX2 packages work with class files, i.e. how they modify their behavior, was reading `ltsect.dtx` — the documented version of the sectioning code in core LaTeX2. It has a comment: "Why not combine `\@sect` and `\@xsect` and save doing the same test twice? It is not possible to change this now as these have become hooks!"

What's a hook? In the Emacs programmable editor, hooks are used to customize the editor's behavior. For example, `before-save-hook` is a list of Lisp functions that should be run just before a file is saved. By default the list is empty, but by adding your own functions to the list, you can have Emacs perform any special actions you want, such as checking the file into a version control system as well as saving it, etc. Emacs provides about 200 hooks, letting you customize most aspects of its behavior.

In LaTeX a hook is slightly different. It's a named function or macro that some other part of the system is going to call. For example, in Section 5 we used `\l@part` as a hook. As we saw, by redefining `\l@part`, you can change how the TOC entries for your Parts are printed. The hook mechanism and the "over-riding functionality" technique above are more or less the same thing.

Hooks are fundamental to how LaTeX packages work: they let the package over-ride the standard operation with something different. As an example, consider the `shorttoc.sty` package, which is useful if you want a one-page summary table of contents before the main TOC, for example. The package contains only about 40 lines of code, and in essence, all it does is call the standard table of contents, having first redefined the variable `\c@tocdepth` to a small value to show only the top levels of contents. In effect, `shorttoc.sty` is using all the standard table-of-contents macros as hooks, although it hasn't changed any of them.

Lesson E: hooks aren't documented (as far as we've been able to see). In fact they can *never* be exhaustively documented, because any package author can just copy any function (as we did with `\l@part` earlier) and over-ride it with their own code, thus using that function as a hook. In real life, the only way you can determine the important hooks is by looking at the important packages, to see which functions they over-ride.

Lesson F: when you copy a chunk of standard code, change the absolute minimum you can get away with. The reason is you don't really know what parts of it might be used as hooks, or what might happen if you remove a call from it to some other hook. Resist the temptation to tidy or "improve" the code.

7 The hard bits 3 — adding extra functionality

(This section describes a technique that you will often come across, and which you might find useful.)

Let's say you want to change some function so that it continues to do exactly what it does at present, but does something extra in addition, i.e. the new is a superset of the old. Here's a real but slightly weird example. The author of a book [4] was using superscripts in his index for a special purpose. We needed a list of the superscripts, and it was difficult to get this from the source files. Using the TeX primitive `\let`, you can assign a whole function to a new variable, and then call the same old function but with the new name. We used this as follows:

```
\let\origsuper=\textsuperscript
\renewcommand{\textsuperscript}[1]
  {\origsuper{XXX(#1)XXX}}
```

This "saves" the original `\textsuperscript` definition as `\origsuper` (Figure 3). Then it redefines `\textsuperscript`, to call the original, unaltered function, but with a modified argument, so that the superscripted text is still superscripted, but is surrounded by the strings `XXX(...)XXX`, which we can then easily search for.[1]

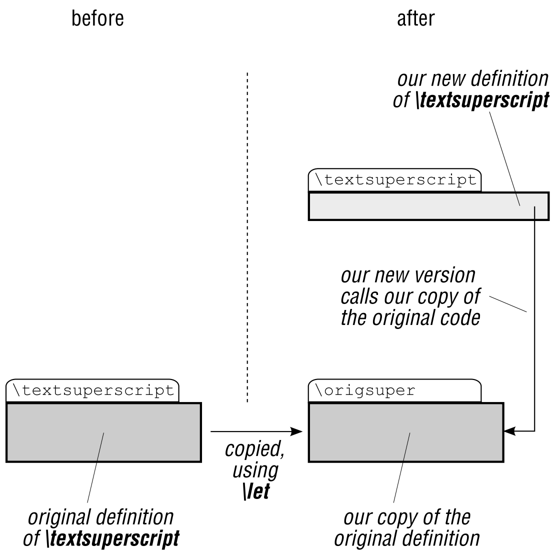
before
after
our new definition of \textsuperscript
\textsuperscript
our new version calls our copy of the original code
\textsuperscript
\origsuper
copied, using \let
original definition of \textsuperscript
our copy of the original definition

Figure 3: Adding extra functionality to a macro

8 The hard bits 4 — plain TeX's syntax

In 1987 I found plain TeX incomprehensible, and nothing has changed. Using LaTeX is non-trivial, but it's powerful and the results are more than worth the effort required. For me, the same is not true of plain TeX: it's too low level, and too complex. Its syntax is weird. Instead of helping you do what you know you need to do, the syntax gets in your way and makes things hard for you. (As an example, we recently found an "off by one" error in a standard package. To fix it, all that was needed was to change '`if X > Y`' to '`if X >= Y`', but plain TeX doesn't let you express things like that, so we had to get someone more experienced in plain TeX to change the code to do the equivalent.)

So, while plain TeX is wonderful and is the foundation on which LaTeX is built, it's not for everyone. (Or, it's for almost no-one?)

Our feeble "solution" to this problem is to avoid it, and when that's not possible, to copy code from packages that work, and hope that LuaTeX (`www.luatex.org`) will eventually make it easier to code complex or low-level macros.

[1] `catdvi` *`file`*`.dvi | tr -s " \t" "\n" | fgrep 'XXX(' | sort -u`

9 The hard bits 5 — indirection in macro names

(Again, this section describes a common technique that you need to understand, although you might not use it often yourself.)

The TeX commands `\csname ...\endcsname` let you construct a "control sequence" name, i.e. a macro, programatically and then invoke it. The following is equivalent to `\textbf{fat cat}`:

```
\newcommand{\mymac}{textbf}
\csname \mymac \endcsname{fat cat}
```

The first line defines the variable `mymac` to be the string `textbf`, and the second line uses the variable to construct a macro name and invoke it, passing the argument '`fat cat`' to it. Being able to invoke a function or macro programatically like this, instead of having to hard-code its literal name in your `.sty` file, makes it possible to handle many similar but slightly different cases compactly and with little duplication of common code.

The sectioning mechanism uses this technique frequently, to construct names of variables or functions related to the level of the current "sectional unit"[2] (SU), such as the macros `\l@part`, `\l@chapter`, `\l@section`, etc. We'll look at this in more detail in the next section, but for now, here's a simple but artificial example of how it works. We define a macro `\T`, whose first argument is the style in which its second argument is to be printed:

```
\newcommand{\T}[2]
   {\csname text#1\endcsname{#2}}
Make stuff \T{bf}{heavy}
   or \T{it}{slanty}. The end.
```

This produces:

Make stuff **heavy** or *slanty*. The end.

10 The hard bits 6 — changing sectioning

The most difficult thing we had to do was change how sectioning works. (We had to do this because our STOP format has to print both section- and sub-section headings on sub-sections.)

For a beginner, sectioning is difficult in three separate ways:

1. There are many functions involved: sections, subsections and lower are defined in terms of `\@startsection`, which then uses `\@sect` (or `\@ssect` if it's a "starred" sectioning command, which in turn calls `\xsect`); all these are complex, and written in plain TeX, which makes life difficult.

[2] A sectional unit is a part, chapter, section, subsection, etc.

The way we got over this was by documenting the functions. This is a work in progress, so we've made the rudimentary documentation available on our Web site (`uit.co.uk/latex`).

2. Sectioning uses indirection a lot. Because the same functions (`\@startsection`, etc.) handle many different levels of sectioning, they use indirection to refer to various parameters for the SU being operated on. For example:
   - The counters `\c@part`, `\c@chapter`, `\c@section`, `\c@subsection`, ... hold the number of the respective SU.
   - The macros `\thepart`, `\thechapter`, `\thesection`, ... specify how the respective counter is formatted. E.g. `book.cls` defines:
     ```
     \renewcommand \thesection
       {\thechapter.\@arabic\c@section}
     ```
     so that the numbering on a section will be of the form "4.9".
   - Similarly, the variables `\l@part`, `\l@chapter`, `\l@section`, ... are what are used to create the table-of-contents entry for the respective SU.

   The first argument to the `\@startsection` and `\@ssect` functions is the type of the current SU, and the functions use this to construct the relevant item they need, as in:
   ```
   \csname l@#1\endcsname
   ```
   This technique isn't intrinsically difficult, but until you're aware of it, the sectioning mechanism can appear incomprehensible.
3. Functions seem to do funny things with their arguments. We cover this in the next section.

11 The hard bits 7 — plain TEX really *is* a macro processor

The file `book.cls` defines:

```
\newcommand\section
  {\@startsection {section} ...
```

i.e. a `\section` is just a call to `\@startsection` with 6 arguments, the first of which is the type of the current SU, as we explained above. However `\@startsection` then calls `\@sect` with 7 arguments, even though `\@sect` is defined to take 8 arguments. And then you realize that `\section` was defined to take no arguments of its own at all! What's happening? Why isn't `\section` defined to take some arguments, like this:

```
\newcommand\section[1]
  {...}
```

since `\section` is always called with a name argument, as in `\section{Thanks}`?

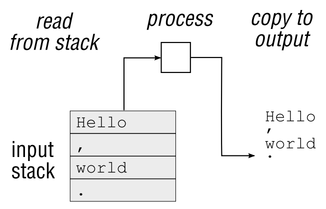

Figure 4: Macro processor — output processing

This starts to make sense only when you realize that plain TEX behaves as a classical, stack-oriented, macro processor (which also typesets!). Initially you can consider the input stack to contain the whole input file. The processor reads input from the file, i.e. removes it from the stack. It just copies the input to the output, unless it's either a macro definition, or a macro invocation, in which case it's evaluated and the result is pushed back onto the input stack, to be re-processed. To make this concrete, let's look at a few simple examples for the `m4` macro processor. The following input has no macros or anything else special, so it's copied to the output without change:

```
% echo 'Hello, world.' | m4
Hello, world.
```

as shown in Figure 4. The slightly more complex:

```
define(`showarg', `my arg is $1')
A showarg(mouse) A
```

defines a simple macro that takes a single argument. Run it and see what you get:

```
% m4 ex1.m4
A my arg is mouse A
```

Now let's have one macro reference another indirectly:

```
define(`concat', `$1$2')
define(`showarg', `my arg is $1')
B concat(quick, brown) B
C showarg(fox) C
D concat(sho, warg)(jumps) D
```

and run this:

```
% m4 example.m4
B quickbrown B
C my arg is fox C
D my arg is jumps D
```

The B and C lines are straightforward, but line D is tricky: `concat(sho, warg)` is read from the stack, leaving only:

```
(jumps) D
```

But what we've just read — `concat(sho, warg)` — evaluates to `showarg`, so the string `{showarg}` is

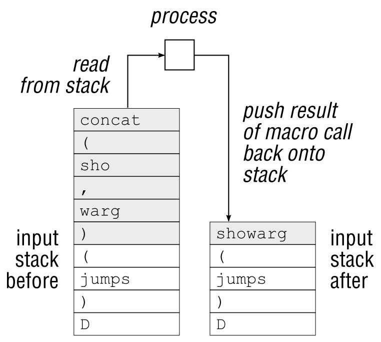

Figure 5: Macro result pushed back onto stack

pushed back onto the stack (Figure 5). The top of the stack now looks like:

```
showarg(jumps) D
```

which is re-evaluated as a call to `showarg` with argument `jumps`. In other words, `(jumps)` was left lying on the stack, and it was picked up as an argument to a macro in due course.

The same thing happens in LaTeX. In the following code:

```
Foo \textbf{cat} bar. (A)
\newcommand{\Ttwo}[2]
  {\csname text#1 \endcsname{#2}}
Foo \Ttwo{bf}{cat} bar. (B)
\newcommand{\Tone}[1]
  {\csname text#1 \endcsname}
Foo \Tone{bf}{cat} bar. (C)
Foo \newcommand{\Tzero}{textbf}
  \csname \Tzero \endcsname{cat} bar. (D)
```

each line produces the same output:

Foo **cat** bar. (A)
Foo **cat** bar. (B)
Foo **cat** bar. (C)
Foo **cat** bar. (D)

However, in lines (C) and (D) the string `{cat}` is not specified as an argument to the `Txxx` macro we defined — it's left lying around, conveniently surrounded by braces, and is picked up later.

This technique is used in the sectioning code. When you write '`\section{Thanks}`', the macro `\section` is invoked with no arguments, leaving the string `{Thanks}` on the stack. `\section` calls `\@startsection`, which calls `\@sect` with 7 arguments; but `\@sect` needs 8 arguments, so it picks up `{Thanks}` from the top of the stack, so it's happy, and we don't get any errors.

12 The results, and lessons learned

Our original LaTeX 2.09 style file had 1400 lines of code, excluding comments.

Our new LaTeX 2_ε style file has 300 lines of code, excluding comments: 34 are `\RequirePackage`s, 150 lines make up 54 `\newcommand`s for convenience-type functions that we ought to have isolated in separate files had we been disciplined enough, and 15 lines make up 11 `\newenvironment`s. The other large chunk of code is 35 lines for our modified version of `\@sect`.

Not only is our new style file much shorter, it's easier to understand and maintain, and is much more flexible than our old one. We can use most standard packages to add extra features with no effort, because we still provide all the hooks that add-on packages rely on. Moreover, the standard class and style files have had years of debugging and are very robust and reliable. By re-using as much as possible, and minimizing the amount of code changed, we've ended up with a stable system. We've had almost no bugs, and the ones we did have we were able to fix quickly and cleanly.

13 Thanks

Lots of people very kindly helped us over the years, with advice and pieces of code. These include, but are not limited to: Donald Arseneau, Barbara Beeton, Timothy Van Zandt, and lots of other patient and helpful people on the `texhax@tug.org` mailing list.

References

[1] Niall Mansfield. *The Joy of X*. Addison-Wesley, 1986.

[2] Niall Mansfield. *Practical TCP/IP — Designing, using, and troubleshooting TCP/IP networks on Linux and Windows (first edition)*. Addison-Wesley, 2003.

[3] Niall Mansfield. *Practical TCP/IP — Designing, using, and troubleshooting TCP/IP networks on Linux and Windows (second edition)*. UIT Cambridge Ltd., 2008.

[4] Jan-Piet Mens. *Alternative DNS Servers — Choice and deployment, and optional SQL/LDAP back ends*. UIT Cambridge Ltd., 2008.

[5] J.R. Tracey, D.E. Rugh, and W.S. Starkey. *STOP: Sequential Thematic Organization of Publications*. Hughes Aircraft Corporation, Fullerton, CA, 1965.

[6] Edmond H. Weiss. *How to Write a Usable User Manual*. ISI Press, 1985.