Towards accessible document classes

Frank Mittelbach*

Abstract

Document classes remain one of the main barriers to accessible LaTeX output. This article describes a template-based approach to class design in which the major structural elements of a document — such as headings, lists, floats, captions, front matter, and page styles — are all exposed through a consistent, tagging-aware interface.

The same framework can emulate established customization packages such as geometry, fancyhdr, titlesec, titletoc, and enumitem, making it possible to adapt both legacy and newly written classes without extensive manual intervention. The result is a more maintainable class architecture with accessibility built in from the beginning, building on the broader LaTeX tagged-PDF project.

Contents

1 Introduction

Accessibility in LaTeX has advanced significantly, but the progress has not been uniform. At the package level, a large number of existing extensions can now be used in tagged documents without breaking the accessibility layer. Document classes, however, still present a much harder problem. A class is not merely a collection of formatting options; it defines the structure of headings, lists, floats, page styles, front matter, and many other document elements that directly affect the semantic structure of the output.

This makes the class the natural place where accessibility needs to be designed-in from the start. The central idea explored here is therefore not merely that a class should be taggable in principle, but that it should remain functional and produce semantically correct structures when accessibility-related declarations are switched on. If a class is designed so that its structural interfaces remain compatible with tagging and other accessibility-related declarations, then accessibility is part of the class design rather than an afterthought.

The broader project to make this possible is described in the LaTeX Project's tagged-PDF feasibility work and follow-up articles [8, 3, 7].

2 What makes a class accessible?

For the purpose of this discussion, an accessible class is one that continues to behave correctly when declarations such as tagging and PDF/UA-oriented setup are added. The standard classes provide a useful example: article, report, and book [4] are among the best-known cases whose semantic elements have been made accessibility-aware by adding appropriate tagging interface commands in the right places. However, this has been done through low-level interfaces, due to the fact that these classes are very old and contain code that dates back to the early nineties [1]. A modern rewrite of these classes using the proposed template approach is still outstanding. In contrast, the beamer replacement ltx-talk, by Joseph Wright, is one of the first classes to have been designed with preservation of structure for accessibility in mind.

The important aspect here is that all the semantic features a class offers are expressed through interfaces that produce the intended layouts and, at the same time, remain structurally visible to the accessibility layer. In other words, when the document asks for tagging, the class should not collapse into a set of incompatible ad hoc overrides that try to establish accessibility after the fact (and typically fail). Instead it should be able to automatically fulfill this request through appropriate interfaces. This accessibility goal is closely tied to the PDF/UA and well-tagged PDF work undertaken by the LaTeX Project team [8, 3, 7].

3 Why the current situation is difficult

The present situation is still uneven. Many classes produce inaccessible results, and the number of fully compatible classes remains small compared with the number of compatible packages. There are several reasons for this.

First, many class files are not hosted on CTAN, so their existence and status may be unknown so that

* With Perplexity AI providing some editorial help.

doi.org/10.47397/tb/47-2/tb146mittelbach-accessibleclasses

structural problems can't be proactively addressed. Second, class design is often driven by complicated layout requirements that LaTeX does not support directly in a straightforward way. As a result, class designers often resorted to misusing available structural elements for layout purposes (e.g., tables or lists without visible items for positioning material) which badly fails if these elements are suddenly recognized as semantic elements by the accessibility layer. Third, many classes are old, layered, and historically evolved, sometimes with code that effectively dates back decades. This is especially true when classes have been patched repeatedly over time rather than redesigned from a coherent interface.

The original discussion of class- and package-level design issues in LaTeX 2_ε already identified many of the structural and interface-related constraints that continue to shape class design today and complicate retrofitting classes for accessibility [1].

Class code tends to fall into recurring patterns:

- Some classes build on the standard classes and make only a few overrides; these are often the easiest to make accessible or are, in fact, already accessible.
- Others extend the standard classes with interface packages and are therefore more promising but not yet fully accessible because not all of the important interface packages are already tagging-aware.
- A third group relies heavily on old code and customizations, especially in the front matter, and often load long chains of historical packages. These classes do require manual adjustments.
- Finally, there are classes built on top of memoir or KOMA-Script, which are powerful general-purpose classes but require their own interfaces to be made tagging-aware or mapped to the new tagging-aware layer.

4 The proposed solution: A template-based customization layer

The proposed solution is to provide a consistent set of templates for all major aspects of a class. This includes page layout and page styles, text and math font setup, titles and front matter, contents lists, document divisions and headings, lists and block environments, math blocks, floats and captions, footnotes and sidenotes, and paragraph setup.

The key design principle is that new templates should provide a consistent interface and include tagging support from the beginning. That consistency is important for both users and implementors. It makes the system easier to understand, easier to extend, and more likely to remain accessible over time. This design direction goes back to the original LaTeX3 class design interface work, where templates and instances were introduced as the central abstraction for document design [2] — it just took us the usual two decades (waiting for computers to be come powerful enough) to turn these ideas into reality.

A modern template system can retain the useful behavior of existing legacy interfaces while expressing it through a uniform abstraction, making accessibility depend on structure rather than on historical accidents of implementation.

4.1 Existing interfaces and kernel support

LaTeX 2_ε already has some customization points, but they are uneven and not uniformly designed. Existing kernel interfaces such as `\@dottedcontentsline` and `\@starttoc` for table-of-contents elements and structures, the famous `\@startsection` for headings, `\@makecaption` for floats, `\@makefntext` and `\@makefnmark` for footnotes, `\@listi`, `\@listii`, etc. for list spacing — they all provide valuable hooks, but they were not designed as a coherent accessibility-oriented architecture. Individually, it would be difficult to make them tagging-aware, but with the new template-based system in place, most of these old interfaces can be emulated and thus old classes relying on them can continue to use them — and get accessibility support for free.

4.2 Emulating established packages

To make the transition practical, the new template system should also emulate the interfaces of important customization packages. Page layout should remain familiar to users of geometry. Running headers should continue to be expressible in a style similar to fancyhdr. Heading and contents-list customization should be approachable for users accustomed to titlesec and titletoc. List customization should match the expectations set by enumitem. Float and caption customization must also be covered, because figures and tables are central to many classes.

This emulation layer matters because it makes the new system usable in practice. It also creates a bridge for many legacy classes. If a class already relies on one or more of these interfaces, the accessibility layer can often be inserted with relatively little or no effort. If not, the class can still be migrated, but the work is inevitably greater.

The LaTeX tagged-PDF project is precisely about making structural information available in a form that can be used consistently by the tagging machinery [8, 3, 7]. One important part of that work is to bridge, and preferably close, the gap between legacy

class design and a consistent tagging-aware template approach.

4.3 Current development status

The current development status is incremental rather than all-at-once. Lists and block environments are already template-based, and enumitem emulation is nearly finished[1] and already distributed. Heading command templates are in progress [6], and titlesec emulation is scheduled. Work is also underway on templates for floats and captions.

That staged approach is deliberate. It allows individual document elements to be made accessible while the larger class architecture continues to evolve. This is also the best way to validate the design: each newly supported element tests whether the template abstraction is expressive enough and whether the tagging behavior is sufficiently robust.[2]

5 How legacy classes benefit

Legacy classes benefit in two distinct ways. If they already rely on interfaces that can be emulated, then much of their accessibility work becomes automatic. If they use older kernel interfaces, then some parts, such as `\@startsection`, can still be handled automatically, while other parts, for example `\secdef`, may require manual intervention.

The difference is visible in the kinds of interfaces involved. A class that uses titlesec can usually be mapped more easily because its structure is already exposed through a higher-level interface. A class that uses old kernel hooks that require low-level coding, such as `\makecaption`, `\makefntext`, or direct table-of-contents manipulations is harder, because there is less semantic structure to build on and more class-specific code to untangle.

The most difficult cases are not necessarily the largest ones, but the ones where the class has mixed several generations of mechanisms together. In those situations, accessibility cannot simply be layered on top. It requires careful adjustments, often in more than one place, to ensure that the semantic structure matches the visual structure of the document.

5.1 Classes based on memoir or KOMA-Script

If a class is based on memoir or KOMA-Script, the potential for automatic accessibility is especially good, but only after these classes have themselves been adjusted to include the necessary tagging sockets (not impossible but difficult and not really scalable) or after their interfaces have been emulated through the new template layer. At the moment this is not yet the case.

This is important because accessibility does not scale if it has to be added to each class separately. It becomes much easier when classes share a common design layer, since tagging support can then be built into that layer and reused across multiple classes. The more classes share such a common design layer, the easier it becomes to make and keep them all accessible.

6 New and rewritten classes

For new classes or classes rewritten from scratch, the template approach offers a much more consistent designer interface. Instead of implementing every document element by hand, the class author selects a template, instantiates it with suitable parameters, and, where needed, defines a document-level syntax around it. Once that is done, the class has a complete and coherent design layer.

This changes class development from a coding-heavy activity into something much closer to layout design as done by a graphic designer. It also reduces the need for layered code and historical patches, which in turn makes the class easier to maintain. Perhaps most importantly, the class becomes accessible out of the box because the templates already provide the necessary tagging support.

6.1 A heading example

As an example, consider headings for poems in the verse package done with the command `\poemtitle`. The designer selects a heading template (for example, `display`) and creates an instance with suitable parameter values,[3] for example:

```
\DeclareInstance{heading}{poem}{display}  {
  name             = poemtitle            ,
  force-unnumbered = true                 ,
  placement        = normal               ,
  before-vspace    = \beforepoemtitleskip ,
  after-vspace     = \afterpoemtitleskip  ,
  title-decls      = \poemtitlefont       ,
  mark-cmd         = \poemtitlemark{#1}   }
```

[1] At this writing, about 2/3 of the enumitem emulation is complete.

[2] We do find possible improvements in already-existing templates when we apply them to new designs, and also sometimes discover desirable functionality not available in the general template mechanism. See the more recent LaTeX newsletters (that accompany each LaTeX release) in which we describe such improvements [5].

[3] Normally, parameters would take explicit values, not commands that hold such values, but the verse package used `\beforepoemtitleskip`, etc. as an exposed user interface — so this is kept for backward compatibility.

The corresponding command definition should feel natural to the user. Thus, `\poemtitle` in `verse` accepts a star, an optional argument for a table-of-contents title, and a title argument, just like any other heading. For the template-based solution, we extend this slightly by allowing a key–value list in the optional argument, as with all template-based commands or environments. This way, one can make one-off key settings in the document to override the defaults of a template instance. In a modern[4] formulation, this would then look like this:

```
\NewDocumentCommand \poemtitle
                    {s ={shorttitle}O{} m}
 {\ParseLaTeXeHeading{poem}{#1}{#2}{#3}}
```

The `\ParseLaTeXeHeading` command takes the three standard heading inputs and an instance name (`poem` in this case), generates all necessary inputs for the template instance (which has 10 arguments), and then calls the instance. In fact all heading commands with a normal LaTeX 2$_\varepsilon$ document interface are reimplemented like this; e.g., `\section` becomes

```
\NewDocumentCommand \section
                    {s ={shorttitle}O{} m}
 {\ParseLaTeXeHeading{section}{#1}{#2}{#3}}
```

that is, its design is done with a `heading` instance named `section`. In this way, all such commands support the same general mechanisms, e.g.,

```
\section*[toc=Bibliography]{Bibliography}
```

would produce an unnumbered section with the title "Bibliography" that would also go into the table of contents, but not in the running header.

The point is not merely to make a command that works, but to make the interface predictable and expressive while keeping the implementation semantic and tagging-aware.

7 Designing good templates

A template is only useful if its inputs, outputs and controls are well-defined. The template type describes what the mandatory arguments are and, more broadly, what the template is supposed to generate from them. It may also specify that all templates of this type accept a certain set of keys (as a subset of all keys a template supports). Conceptually, two templates with the same template type should be interchangeable in a meaningful way.

That sounds straightforward, but in practice it is one of the most difficult design questions in the system. To define a good template type, one must decide what counts as document input and then find a balance between generality and practicality. Too little structure makes the templates inflexible; but too much structure makes them cumbersome and hard to use.

7.1 A heading template type

Headings are a good stress test for template design. A heading may need a key-value list to override defaults, a flag for requesting an unnumbered heading, the main title, a table-of-contents title, a running title, a bookmark title, a `\nameref` text, labels, subtitle text, and perhaps quotation or epigraph material. In very specialized publications you might even have more structural inputs — so where to draw the line? There is no single obvious answer to the question of what the template inputs should be.

The current design reflects repeated refinement. The goal is to present LaTeX's inputs in a way that is sufficiently general to support many real use cases, while still being practical enough that users and class writers can understand what they are doing. This is one of the reasons template types matter so much: they determine whether a template family becomes a coherent abstraction or a collection of unrelated special cases.[5]

7.2 Providing the right keys

Another problematic area is the key names that are used in templates. Keys need to be understandable and consistent. Users should be able to infer the meaning of a key from its name, and similar concepts should be named in similar ways across templates. This is harder than it sounds, because a system that grows over time often accumulates accidental inconsistencies.[6]

The current naming conventions use hyphens to structure longer names. For example, ⟨*name*⟩`-vspace` is used for vertical spacing, horizontal-space keys typically end in `-hspace` or `-margin`, ⟨*name*⟩`-decls` holds declarations such as font changes, and ⟨*name*⟩`-format` denotes a function expecting one argument. Likewise, `order` or ⟨*name*⟩`-order` denotes a comma list describing a placement or ordering, e.g.,

```
order={number,title,punct,separator,note}
```

as used in "theorem-like" templates.

[4] Well, not that modern: envisioned already in 1999 [2].

[5] And it is the place where actual implementations can help to find suitable answers. In the case of headings we started with the obvious three inputs, that we all know from years of LaTeX usage, and over some period finally settled on the current 10 inputs as a good compromise.

[6] For that reason we are constantly reviewing the key names across different templates as we develop new templates for additional areas and as a result sometimes retract and replace some names. We expect that this will get us to a coherent set over time that then no longer changes, but at present some level of future adjustment is unavoidable.

These conventions are still evolving, but the overall goal is clear: keys should be predictable so that a class designer can work with them without needing to memorize a large number of unrelated special cases.[7]

8 Monoliths or coordinators and workers

There are two broad ways to organize templates. In a monolithic design, one template instance implements an entire document element. That approach is easy to understand because all customization is in one place. The drawback is that similar code often has to be repeated in multiple templates, which creates maintenance problems and makes consistent layout changes harder as value updates have to be made in several places.

In a coordinator-and-worker design, an organizer template handles the overall element and delegates individual tasks to worker templates. Those worker templates can then be reused in multiple organizers, which improves consistency and reduces duplication. The trade-off is that the design becomes more complex, and adjustments must be made in the correct place. There is also a possible user adjustment problem when different worker templates reuse the same key names in different contexts.

This question of what to use is still open. Both approaches have clear advantages and clear costs. Monoliths are simpler to understand, while coordinators and workers are better for consistency and reuse. The right answer may vary by document element and by the amount of customization expected from users — and also on personal preference.

9 Practical implications

The practical benefit of this work is that accessibility becomes part of the design process rather than a later repair step. That matters for both new and legacy classes. New classes can be designed with a clean abstraction layer from the start, while existing classes can migrate through a combination of interface emulation and targeted adjustments.

The broader implication is that accessibility, maintainability, and good class architecture are not separate goals. They reinforce each other. A class with clear structure, reusable templates, and stable interfaces is easier to extend and easier to make accessible. Conversely, a class that relies on ad hoc overrides becomes harder to understand and maintain in every respect.

[7] Of course, good documentation helps too, especially if your memory is bad — I find myself often referring to documents, such as `texdoc blocks-doc`, to remind myself in detail of what we have set up.

10 Conclusion

Document class accessibility is feasible, but it requires a deliberate shift in how classes are designed. A consistent template-based system with tagging support offers a practical route forward. It allows legacy classes to inherit accessibility through emulated interfaces and gives new classes a clean, modern architecture from the outset.

This is still work in progress, and the remaining design questions are real and difficult to resolve. Nevertheless, the direction is promising: accessibility can become a natural consequence of class design rather than an exceptional case that demands special handling.

That is the kind of foundation LaTeX needs if accessible document production is to become routine, and the default rather than the exception.

References

[1] J. Braams. Document Classes and Packages for LaTeX 2ε. *TUGboat* 15(3):255–262, 1994. `tug.org/TUGboat/tb15-3/tb44braams.pdf`

[2] D. Carlisle, F. Mittelbach, C. Rowley. New Interfaces for LaTeX Class Design, 1999. The LaTeX Project, TUG99 slides. `latex-project.org/publications/1999-l3team-TUG-l3-interfaces.pdf`

[3] U. Fischer. On the road to Tagged PDF: About StructElem, Marked Content, PDF/A and Squeezed Bärs. *TUGboat* 42(2):170–173, 2021. `latex-project.org/publications/2021-UFi-TUB-tb131fischer-tagpdf.pdf`

[4] The LaTeX Project. *Standard Document Classes for LaTeX version 2e*, 2025. `latex-project.org/help/documentation/classes.pdf`

[5] The LaTeX Project. *LaTeX News, Issues 1–43*, 2026. `latex-project.org/news/latex2e-news/ltnews.pdf`

[6] The LaTeX Project. *Reimplementation of LaTeX 2ε's heading commands using templates*, 2026. `texdoc.org/serve/latex-lab-sec-template/0`

[7] F. Mittelbach, U. Fischer, et al. Automatically producing accessible and reusable PDFs with LaTeX. In *Proceedings of DocEng 2024*, pp. 1–4, 2024. `doi.org/10.1145/3685650.3685670`

[8] F. Mittelbach, C. Rowley. LaTeX Tagged PDF — A blueprint for a large project, 2020. *TUGboat* 41(3):292–298. `latex-project.org/publications/2020-FMi-TUB-tb129mitt-tagpdf.pdf`

◇ Frank Mittelbach
LaTeX Project
`frank.mittelbach (at) latex-project.org`