



## Using LaTeX3's xtemplate

Asked 9 years, 10 months ago   Modified 2 years ago   Viewed 3k times



46



Relevant are the answers to [Moving to LaTeX3 for package authors](#) and [What new bits have already been implemented in LaTeX3? Will my current documents \(with many packages\) still compile with LaTeX3?](#)

How is [xtemplate](#) properly used? I've read the package documentation and Lars Hellström's [Some notes on templates](#), and I've tried to make sense of the [xfrac](#) implementation.

What I've gleaned is that there are "objects", "templates", "keys", and "instances". But how they interact is confusing, and `xfrac` has too many options to grasp easily (and is written assuming the reader understands `xtemplate`, rather than as an `xtemplate` tutorial).

Can the concepts of `xtemplate` be described in a simple-but-complete worked example?

latex3

xtemplate

[Edit tags](#)

Share Edit Follow Close

Flag

edited Apr 13, 2017 at 12:34



Community Bot

1

asked Jun 6, 2013 at 18:58



J. C. Salomon

4,838

23

47

3



I know this is a lot to ask for. I'm also aware that the LaTeX3 team considers the current implementation of templates to be unready. This question is asking for clarification about the concepts. – [J. C. Salomon](#) Jun 6, 2013 at 18:59

4



This is a nice question. I am looking forward to reading the answers. – [Gonzalo Medina](#) Jun 6, 2013 at 19:04




this is indeed a difficult one to crack because some of the ideas as implemented in the current version are "wrong" in my opinion by now. But since it is more about "concepts" for that level, I guess I could give it a try ;-) – [Frank Mittelbach](#) Jun 6, 2013 at 19:19



@FrankMittelbach now I am curious! Would your answer refer to some of those things that you consider to be wrong in the current implementation? – [Gonzalo Medina](#) Jun 6, 2013 at 19:23

4 Answers

Sorted by:  
[Reset to default](#)

Date modified (newest first) 

## Full example

49

In the first implementation of the template ideas there has been a full example that tried to work through the underlying ideas. One can still find the documentation of `template.dtx` on the web, e.g., [here](#) (in the directory `/doc/latex/xpackages/xbase/`).

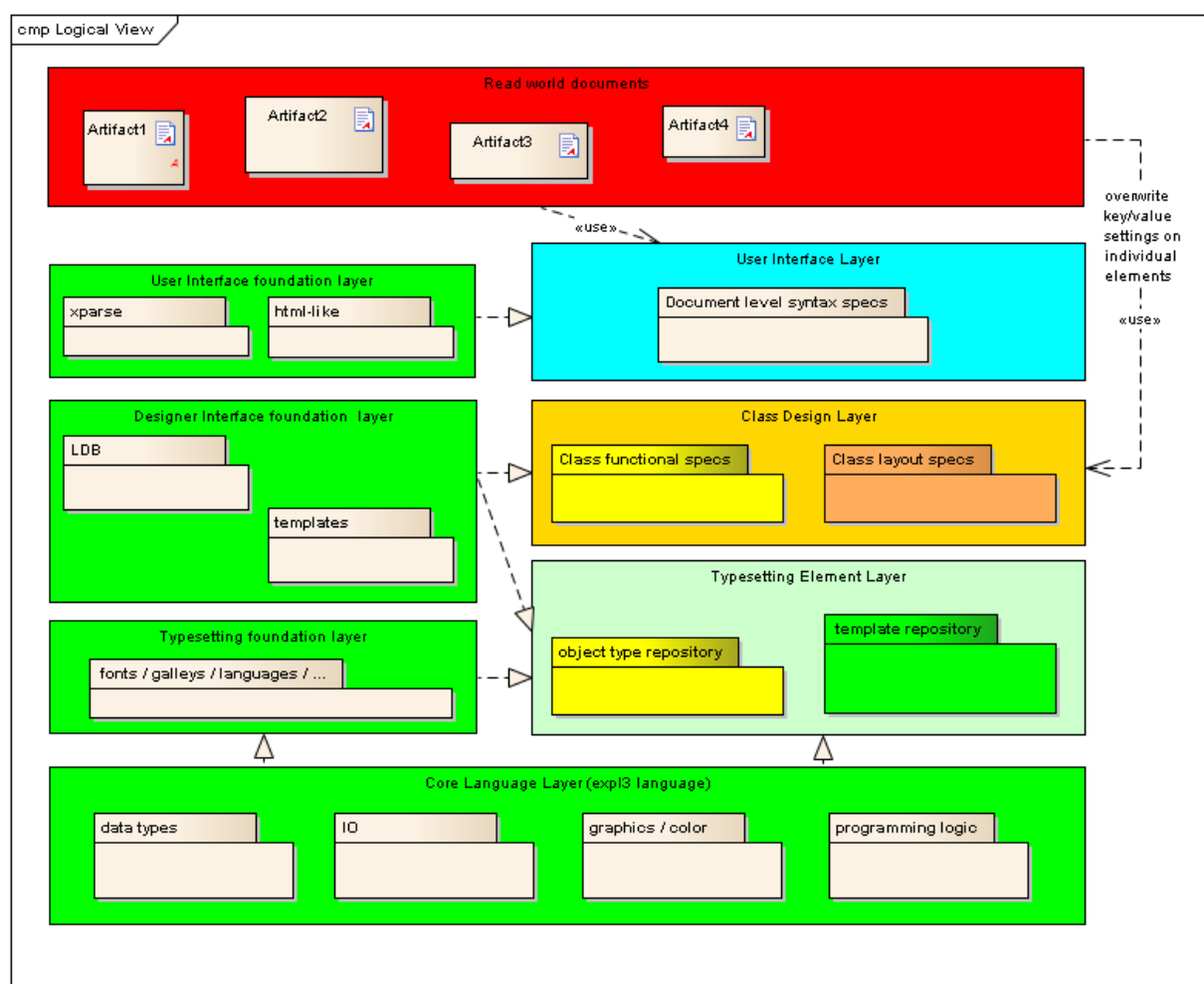


This is for a different implementation but the basic concepts haven't changed so it might be helpful still. Somehow during the reimplementation as `xtemplate.dtx` this part of the documentation was dropped (right now that section is "empty").



## The Concepts explained

The concepts implemented in `xtemplate` (which I consider these days as a second-level proto-type) are best explained using some pictures. Here is my rough sketch of what I think the LaTeX3 architecture should look like:



The `xtemplate` package builds out the concepts for the "object type repository", the "template repository", (parts of) the "Class functional specs", and the "Class layout specs". What it doesn't do, though, is interfacing with what is called "LaTeX Database (LDB)" because at the time the main work for `xtemplate` was done we thought that the concepts behind the LDB are not realistically implementable, so we put it aside and tried to implement a solution without it, which is largely what is the current `xtemplate` code.

Okay, so this doesn't tell you any better what this is all about, I guess : ) So let's try to

Okay, so this doesn't tell you any better what this is all about, I guess. :-) So let's try to explain it from the middle part outwards. There we have the "Typesetting Element Layer" and the idea is that this describes all the "typesetting elements", i.e., objects that take inputs (0 or more) and do something with it to produce a "typeset result".

## Typesetting Design Layer

The idea now is that these "typesetting elements" can be abstractly described by

- the inputs they take (and what they semantically mean)
- and the semantic of what they produce (for the sake of a better word)
- but not the visual representation

This is what we call the `ObjectType` and they end up in the "object type repository". As an example think of a "heading" object type that takes a number of inputs, e.g., the heading title, the TOC title, whether or not it is numbered, etc. (some of those inputs could be special values like "NoValue") but for sake of speed etc the current implementation assumes each type has always a fixed number of arguments and they are positional.

Now so far this says nothing about how things should be visualized. It just gives abstract functional elements.

For each `ObjectType` there are a number of templates (one or more) in the "template repository" that all implement a visualization of the element, ie, they do the typesetting. They are called templates because they are intended to offer a certain flexibility in providing design choices. Whether designs are split over different templates or are implemented in a very complex template is a bit of a matter of taste and practicality, e.g., one may have a template for generating display headings and one to generate run-in headings or attempt to have a more complex template that can produce both layouts (like LaTeX2e's `\@startsection` tries).

`xtemplate` provides a command (`\DeclareTemplateInterface`) to declare a template interface which is basically the description of the knobs and wistles the template offers to manipulate the design it implements. This is given as a list of keys which can have a number of input types. This is, if you like the designers side of the house: to instantiate a template the designer would need to give such keys values.

It also provides a declaration for the template code (`\DeclareTemplateCode`) that implements the design taking the interface key values as input and the document arguments from the `ObjectType`. This coding would be done using the `expl3` layer (i.e., the "CoreLanguage Layer") with the help additional help of what I termed the "Typesetting foundation layer" (of which not much exists so far).

All this is in the box "template repository".

Now if none of the available templates implement the design the designer is looking for,

then another template has to be coded that implements the desired design for the `ObjectType` first, otherwise the task of a designer is to select a suitable template and initiates it with some values to achieve the wanted result.

## Class Design Layer

Moving up the stack we have two bricks in the "Class Design Layer": The "Class functional specs" and the "Class layout specs". Note that this layer doesn't deal with the document level syntax. We are still on the level of (named) `ObjectTypes` and those take inputs in a standardized form.

### Class functional spec

The "Class function spec" describes what are the logical elements are that a document class provides, e.g., that it provides the following heading levels: "A-head", "B-head", down to "F-head" say, or in 2e terms "section", "subsection", etc, down to "subparagraph".

It should also describe the relationships (not really there (yet)) so in essence the idea of this is to provide a kind of "class DTD".

The idea is that if a document belongs to a certain document class (as described by this functional spec) its visualization could be changed from one form to another by replacing on layout spec conforming to the functional spec with another conforming to this functional spec.

In other words a certain functional spec should be common for a number document class layouts, which would make those layouts interchangeable without otherwise touching the document. As we know this is partially possible in 2e but often not quite, e.g. `article.cls` can be replaced by `amsart.cls` but some things will fail because the DTDs are not fully the same unfortunately (and in 2e there is no separation of the DTD from the visualisation both is done in the same place, i.e. in the `.cls` file so it is not that surprising).

Technically speaking this block is not properly covered by `xtemplate` at the moment it only exists conceptually, see discussion of deficiencies below. The closest you get here is `\UseInstance` but this is not a declaration that can stand on its own, but something to be used one layer up.

### Class layout spec

The other building block handles the real instantiation for a specific class, e.g., if the functional spec says there is an "A-head" and a "B-head" then here we would have

```
\DeclareTemplateInstance{heading}{A-Head}{some-template}
  {key1=val1, key2=val2 ...}
\DeclareTemplateInstance{heading}{B-Head}{maybe some other template}
```

```
\DeclareTemplateInstance{heading}{A-head}{some-other-template}
{key1=val1, key2=val2 ...}
```

So both together describe what a specific document class provides in terms of elements and how these elements are formatted.

The only thing that is still missing is how such elements are encoded in real documents, i.e. what syntax is used to input them. That is subject to the next layer up.

## User Interface Layer

The top-level layer turns document input syntax into the format used on the "Class Design Layer (functional spec)". An example of this would be the `xparse` implementation that provides parsing functionalities a la LaTeX2e with some extra bells and whistles. But instead of this one could provide a different user syntax, e.g., some xml interface or ...

Putting the stuff together, using the current possibilities we would then have something like

```
\DeclareDocumentCommand \section { * o m } % we implement 2e's interface
{ \UseInstance{heading}{A-head} {#1} {#2} {#2} {#3} }
```

If you wonder why this looks so funny: For the sake of an argument :-) the above assumes that the ObjectType "heading" was defined to require 4 arguments:

- a boolean value (indicating if the heading should not get numbers)
- a text for the TOC or "NoValue"
- a text for the running heading or "NoValue"
- a text for the heading title

As the 2e document level interface has only one optional argument we duplicated it for both TOC and running-head (as it is also done in 2e) Instead we could have offered a different top-level syntax like

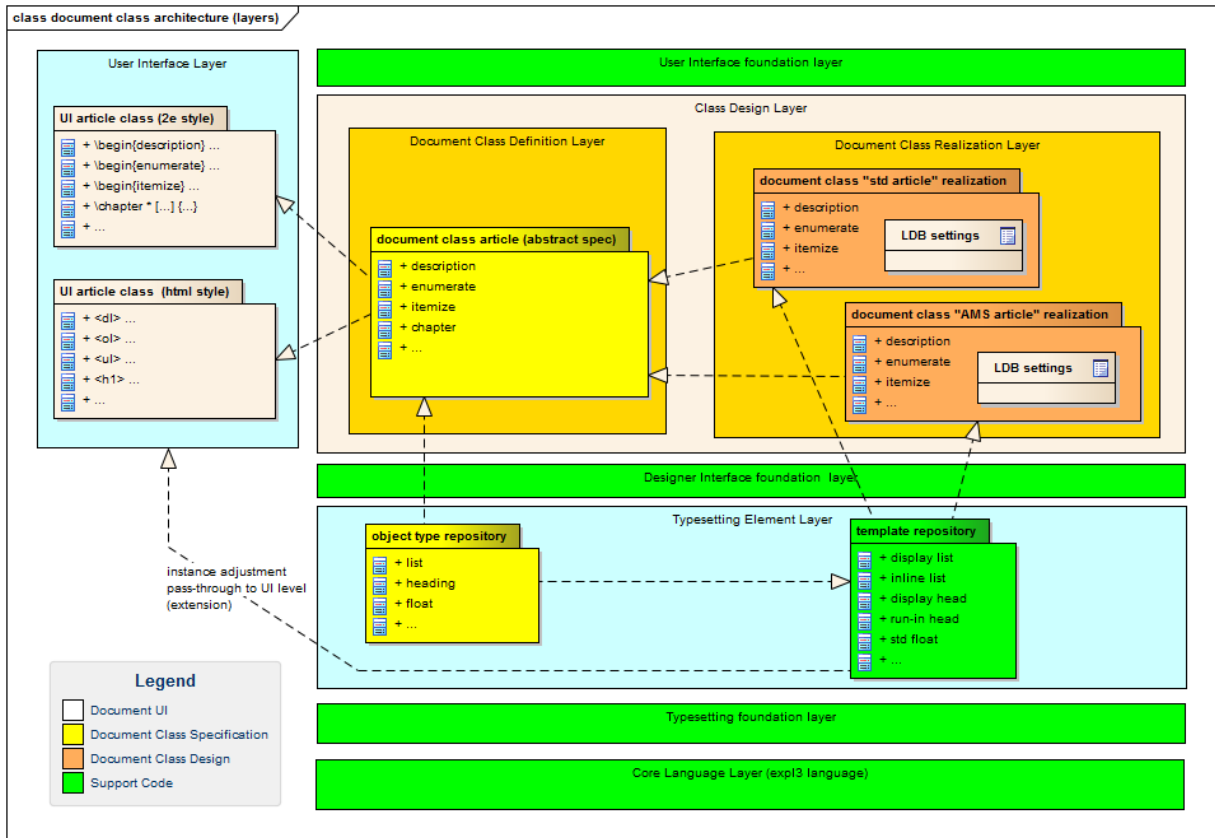
```
\DeclareDocumentCommand \section { * o m o }
{ \UseInstance{heading}{A-head} {#1} {#2} {#4} {#3} }
```

that does make use of all input arguments the object type "heading" has (only in a different order)

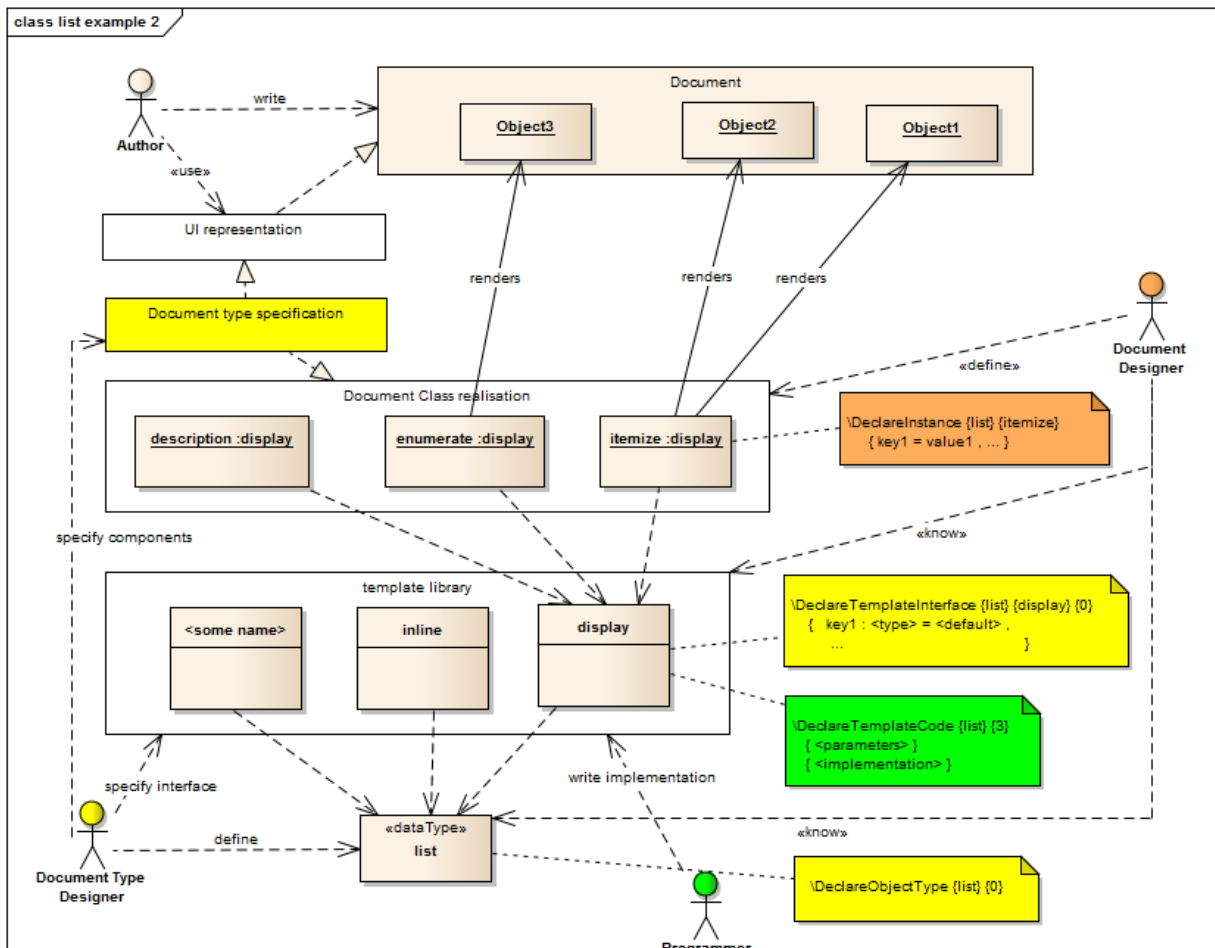
or ...

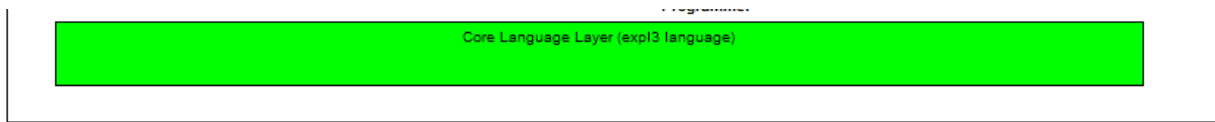
## Summary

To summarize here is a slightly different view on the architecture just described (ignore the LDB part for now):



And here yet another one looking at it more from the perspective of the different roles and how things are be specified using the declaration possibilities of `xtemplate` :





You can see four different roles:

- the **Document Type Designer** defines what kind elements are in a document class (and possibly defines ObjectTypes if they are not yet in the object repository). He is also listed as the one who defines the template interface, but I guess I marked that wrong -- that should probably be the designer doing that.
- if a template actually needs to be produced that would be the task of a **Programmer** who really understands how to program in `expl3` and all the goodies :-) but hopefully for most requirements usable templates exist (after a while)
- the of the **Designer** is then to select suitable templates and decided on the right kind of values for their keys, and if no suitable template exist, ask a **Programmer** to build one.
- and finally there is the **Author** whose task is to concentrate on writing a real document and not worrying about all the other stuff

Of course, in real life those roles might end up being played by a single person.

## The deficiencies

... or what is wrong in my opinion with the current proto-type. Well, what the current implementation provides is an approach to static design:

- here is a abstract object (type), say a heading that takes a defined set of inputs
- here is a bunch of template that typeset such an object
- so to make a design, select a template and select the right parameters and you are done

Simple, right? Well not quite, because often document design needs flexibility depending on context. Take for example display headings:

- they have a defined space above the heading (say `preskip` )
- they have a possibly different space below the heading ( `postskip` )

But what happens if one heading directly follows another one as in

```

\section{A-head}
\subsection{B-head}
  
```

What is the space between them?

- the sum `postskip(A-head)` and `preskip(B-head)` ?



- the maximum of both?
- the minimum?
- something else?

LaTeX2e solves this trivially by using `\addvspace` which is implementing the second solution, i.e., the maximum. But this is fairly limiting the design possibilities.

One can think of other possibilities, e.g. solving this somewhat adhoc so that a heading can detect that it is directly following another heading and then using a different key, e.g. `between-heading-skip`. But this is conceptually rubbish because

- all decisions would be placed into the second heading and not really on the knowledge that it is an A-head followed by a B-head
- it would suddenly require the templates to understand and manage context information which makes them complex and if a single template is not obeying to some special protocol all the identification, etc. would break down like a house of cards.

So this is an example "context" because some elements are in sequence. Another example is that elements behave quite differently because they are in different areas of the document. For example

- headings are differently numbered in frontmatter, mainmatter and backmatter
- lists like `itemize` need totally different spacing if used in contexts with different font sizes, or in special regions (e.g., you may want a totally different design for them if such a list appears in a boxed area in the margin).

Basically none of this is supported properly by `xtemplate`.

Move the film back: in the early '90s (yes it is that long ago) we had grand ideas with something that we called LDB back then that would allow to manage context dependencies of high complexity. To some extent it was a pre-CSS (not quite but CSS came up with some of those ideas a couple of years later) and that would have fairly perfectly managed all this complexity. However back then these ideas were simply years too early and we finally came to the conclusion that it is just technically not working.

Times have changed: the problem is still not solved but computers got so much faster that I'm now fairly determined to retry this, see [my talk given at TUG 2011 in India \[YouTube\]](#) (not surprisingly you will see some of the above pictures there in the slides).

So back then we implemented the LDB (in two competing approaches but both were too slow and took too much memory, so be it onto the shelf and instead we came up with the "simpler" approach in `template.dtx` in 199x which then later was reimplemented by Joseph to become `xtemplate.dtx`. And for the "static" design part the template approach is really good and getting you somewhere. But the dynamic part remained and so we tried to solve this in a simpler fashion.

One idea we had was to provide "collection instances", basically that you can define several "collections" of instances and then switch between them. But this is such a crippled and restricted way of describing context that I'm totally convinced now this is a dead end. So consider this as the "rubbish part" in the concept which is also why I didn't describe it above (but it is still documented in `xtemplate` even with some warning, so be aware).

Instead what I think needs happening is to bring together the ideas from `template` and from LDB for context and offer them in a coherent way --- and nowadays this is possible and I believe I have a good vision on how this should look like.

Share Edit Follow Flag

edited Apr 6, 2021 at 12:35

answered Jun 6, 2013 at 22:37



Marijn

34.6k

2

25

69



Frank Mittelbach

75.8k

9

202

273



Thank you; and I'm looking forward to the rest of the post. I think there may already be more information collected here than in all papers I could find online. – J. C. Salomon Jun 6, 2013 at 23:24

1



@J.C.Salomon actually in the original `template` implementation (which one can still find on the web, e.g., <ftp://ftp.nyu.edu/pub/ftp.dante.de/pub/tex/macros/latex/contrib/...>) there is a longer complete example describing the ideas back then. It is on an earlier implementation but might give some additional insights. For some reason when `template` got reimplemented and became `xtemplate`, that section was not kept/readded.

– Frank Mittelbach Jun 7, 2013 at 11:40



Thank you for that link; this document is exactly what I was hoping for in my question. When you get around to fleshing out the rest your answer, do you suppose you could add this link somewhere near the top? – J. C. Salomon Jun 7, 2013 at 17:13



@J.C.Salomon done, but the correct solution would be to adjust the example there and add it back into `xtemplate` – Frank Mittelbach Jun 7, 2013 at 21:59

1



I sincerely hope you hide this stuff somewhere really deep in the docs before the time comes to shout 'LaTeX3 is ready, people!' Otherwise regular users will be scared away by the complexity. – marczellm Feb 4, 2014 at 21:09

|

9

Frank Mittelbach and Dave Carlisle outlined well the ideas behind the `xtemplate` package. In the MWE following, we wish to emulate the `html` element `<span>`. This is normally used to style in-line text elements.

One of the benefits of the ideas behind `xtemplate`, which might not have come out very clearly in the other answers, is that the Programmer or Designer can expose an API to the users. This is a fundamental way good programs are developed. If for example LaTeXE packages followed this concept, we would have been loading instances of the packages and key conflicts could be reduced, if not totally eliminated.

The process in defining templates follows concepts of *object orientated* programming. An object is first defined and then instances of this object are created and used. The idea of an interface is that it exposes an interface to the external world. I am sure Frank at the time he developed the package, was using Java as it was very popular at the time. Each language has its own idiosyncracies and patterns. As `expl3` is becoming a computer language on its own, these are good concepts. So the `xtemplate` is not just another bunch of macros that define keys.

So the first thing we need to define is the object.

```
\DeclareObjectType{inlineobj}{1}
```

The name we give to the object is immaterial, but I think it is probably better to have the name of your package, to ensure it is unique. This just stores the number of parameters in a property store.

The interface part is coded next:

```
\DeclareTemplateInterface{inlineobj}{span}{1}
{
  font-face: tokenlist,
  font-shape: choice {italic, slanted, normal}=normal,
  font-weight: choice {bold, normal}=normal,
  font-color: tokenlist=black,
  quote: choice {none, enquote}=none,
}
```

Don't be confused with the LaTeX3 way of typing out macros. This part just call a macro with three parameters `\DeclareTemplateInterface {#1}{#2}{#3}`.

Next we declare the code using `\DeclareTemplateCode{inlineobj}{span}{1}{}`. Where the `\AssignTemplateKeys` starts is where the code implementation starts.

The instance is created using similar code, in our case `tn`.

```
\DeclareInstance {inlineobj}{tn}{span}
```

```

    {
      font-face=arial,
      font-shape=normal,
      font-weight=normal,
      font-color=red!80!black,
      quote=none,
    }

```

Here is the not so short MWE.

```

\documentclass{article}
\usepackage{expl3, xtemplate, xparse, csquotes, xcolor, fontspec, xspace}
\newfontfamily\arial{Arial}

\begin{document}
\ExplSyntaxOn
\DeclareObjectType{inlineobj}{1}
\DeclareTemplateInterface{inlineobj}{span}{1}
{
  font-face: tokenlist,
  font-shape: choice {italic, slanted, normal}=normal,
  font-weight: choice {bold, normal}=normal,
  font-color: tokenlist=black,
  quote: choice {none, enquote}=none,
}

\DeclareTemplateCode{inlineobj}{span}{1}
{
  font-face = \l_font_tl,
  font-shape = {
    italic      = \cs_set_nopar:Nn \afontshape: {\itshape},
    slanted    = \cs_set_nopar:Nn \afontshape: {\itshape},
    normal     = \cs_set_nopar:Nn \afontshape: {\upshape}
  },
  font-weight = {
    bold       = \cs_set_nopar:Nn \afontseries: {\bfseries},
    normal     = \cs_set_nopar:Nn \afontseries: {\mdseries}
  },
  font-color = \l_tmpa_tl,
  quote = {
    none       = \cs_set_nopar:Npn \quotemacro:n #1 {\detokenize{#1}},
    enquote    = \cs_set_nopar:Npn \quotemacro:n #1 {\enquote{\detokenize{#1}}},
    unknown    = \cs_set_nopar:Npn \quotemacro:n #1 {\detokenize{#1}}
  },
}
{
% the implementation part
\AssignTemplateKeys

\group_begin:
\color\l_tmpa_tl
\cs:w \l_font_tl \cs_end:

```

```

\fontshape:
\fontseries:\selectfont
\quotemacro:n{#1}
\group_end:
}

\DeclareInstance {inlineobj}{docFunction}{span}
{
  font-face=arial,
  font-shape=italic,
  font-weight=bold,
  font-color=green!40!black,
  quote=enquote
}

\DeclareDocumentCommand\docFunction { m } {
  \IfInstanceExistTF {inlineobj}{docFunction}
  {\UseInstance{inlineobj}{docFunction}{#1}}
  {ERROR}
}

\DeclareInstance {inlineobj}{tn}{span}
{
  font-face=arial,
  font-shape=normal,
  font-weight=normal,
  font-color=red!80!black,
  quote=none,
}

\DeclareDocumentCommand\tn{ m }{%
  \IfInstanceExistTF {inlineobj}{tn}
  {\UseInstance{inlineobj}{tn}{#1}}
  {ERROR}
}
\ExplSyntaxOff

```

The function `\docFunction {get_string ( )}` is used throughout to get a string in LuaTeX, where macros in text paragraphs are shown as `\docFunction{\my_macro}` in green. Typewrite text is obtained by using `\tn{\tn}`.

```
\end{document}
```

Improvements to the code are welcome, especially around the `\detokenize` part to remove the space at the end. I would also welcome any ideas as how to emulate `pgfkeys handlers`.

Share Edit Follow Flag

edited May 13, 2015 at 18:23

answered May 13, 2015 at 18:14



**Yiannis Lazarides**

**115k** 33 282 553



7

Can the concepts of xtemplate be described in a simple-but-complete worked example?



TUGBoat 33:3 has an article by Clemens Niederberger titled [The xtemplate package: An example](#) (now publicly-accessible).

It takes [the author's answer](#) to [Macro for formatting names \(initials or full name\)](#) and uses this as a worked example of defining xtemplate "objects", "interfaces", and "instances". By the author's admission, " xtemplate is overkill for this [exercise]"—but this makes for a more comprehensible example.

Share Edit Follow Flag

edited Apr 13, 2017 at 12:35

answered Jul 1, 2013 at 3:00



Community Bot

1



J. C. Salomon

4,838 23 47

- 
- 1 I used [tex.stackexchange.com/a/57641/5049](https://tex.stackexchange.com/a/57641/5049) as an inspiration for the example. xtemplate is overkill for this so it is not a *real world* use case :) – [cgnieder](#) Jul 1, 2013 at 10:56
- 
- 1 @cgnieder, I'd expect any tutorial example to be at the edge of "is this tool overkill for this case?" `xfrac` might have served, but it's written as an example for people who already understand the template concept. I look forward to reading your article. – [J. C. Salomon](#) Jul 1, 2013 at 19:19
-

▲  
27  
▼

The *idea* is to have a formalised key=value interface to setting a set of parameters on some part of the typographic layout.

So if you look at article's table of contents for example the customisation options vary from setting some internal @ named macros



```
\newcommand\@pnumwidth{1.55em}
\newcommand\@tocrmarg{2.55em}
\newcommand\@dotsep{4.5}
```

To setting top level counters

```
\setcounter{tocdepth}{3}
```

To just defining chunks of internal code

```
\newcommand*\l@section[2]{%
  \ifnum \c@tocdepth > \z@
    \addpenalty\@secpenalty
    \addvspace{1.0em \@plus\p@}%
    \setlength\@tempdima{1.5em}%
    \begingroup
      \parindent \z@ \rightskip \@pnumwidth
      \parfillskip -\@pnumwidth
      \leavevmode \bfseries
      \advance\leftskip\@tempdima
      \hskip -\leftskip
      #1\nobreak\hfil \nobreak\hb@xt@\@pnumwidth{\hss #2}\par
    \endgroup
  \fi}
```

The idea is that a better interface would be that a document class could choose one of a number of basic *templates* for (say) table of contents which would then have a documented set of parameters which may be set within a standard code-free declarative interface.

the syntax in the class may look something like

```
\DeclareInstance { toc } { this-toc } { basic }
{
  toplevel = 2,
  font = \normalsize,
  defaultstyle = dotted,
  chapterstyle = undotted,
  chapterfont=\large\bfseries,
}
```

where separately in the template definition the declaration of the possible keys is made and the mapping somewhere to actual code.

Given such parametrised templates for tables of contents, page geometry, headings, lists, it should be possible to implement a wide variety of designers layout specifications without really using any internal TeX code.

However the current template design and implementation is very old (I've not really looked at it for 10 years:-) ( *apparently parts of it are newer than I thought:-)* ) and really needs a re-think and a re-implementation, so while the ideas are sound the actual syntax should not be considered at all stable.

Share Edit Follow Flag

edited Jun 6, 2013 at 19:29

answered Jun 6, 2013 at 19:22



**David Carlisle**

**709k** 66 1520  
2356

---

1 Can you provide an example of the usage. The idea is explained great. – **Marco Daniel** Jun 6, 2013 at 19:38

Let me see whether I've understood correctly: There will be an *object-type* of "listable" stuff (sections, figures, theorems, *etc.*), a *template* for sectioning stuff (*i.e.*, a list of parameters), and *instances* of the template for the several sectioning commands (*e.g.*, `\Section` titles are set in *this* particular font, set so far from the following text). Basically, `memoir` will be included and the `KOMA` classes will be just a list of alternate instances of the templates. Does that sound right? – **J. C. Salomon** Jun 6, 2013 at 20:15

@J.C.Salomon it might not be possible to make a class *just* a list of template instances but that would be something to aim for, one could even then imagine some kind of GUI class instance generator where you could set the parameters in some kind of graphical view. hard to do now as there isn't really any division between parameter and code. – **David Carlisle** Jun 6, 2013 at 20:20

Did I understand correctly the rôles of object-type, template, and instance? – **J. C. Salomon** Jun 6, 2013 at 20:22

@J.C.Salomon probably (but wait for Frank to reply, this is his idea really:-) – **David Carlisle** Jun 6, 2013 at 20:32

---

|