Towards LaTeX coding standards

Didier Verna

Abstract

Because LaTeX is only a macro expansion system, the language does not impose any kind of good software engineering practice, program structure or coding style. Maybe because in the LaTeX world, collaboration is not so widespread, the idea of some LaTeX coding standards is not so pressing as with other programming languages. Over the years, however, the permanent flow of personal development experiences contributed to shaping our own taste in terms of coding style. In this paper, we report on all these experiences and describe the programming practices that have helped us improve the quality of our code.

1 Introduction

If the notion of coding style is probably almost as old as computer science itself, the concern for style in general is even older. An interesting starting point is the book "The Elements of Style" [16], first published in 1918 (the fourth edition appeared in 1999). This book is a style guide for writing American English and has been a source of inspiration for similar books in computer science later on. It is interesting to mention the fact that this book has been virulently criticized since its very first publication. Although generally considered as a reference book, the authors were also accused of being condescending in tone and of having only a very partial view on what proper American English should be. This is already a strong indication that talking about style, whether in natural of programming languages, can be quite controversial. Indeed, a style, in large part, is a matter of personal taste before anything else. Consequently, what is considered to be good style by one person can legitimately be viewed as bad style by another person.

The first book on style in programming languages was published in 1974 (a second edition appeared in 1978) and was entitled "The Elements of Programming Style" [8], as an explicit reference to its aforementioned predecessor. Although this book was not dedicated to one programming language in particular, it was still largely influenced by the few of that time. Since then, numerous books on programming style appeared, many of them focusing on one specific language, and being entitled "The Elements of `XXX` Programming Style" to follow the tradition. This includes recent languages such as C#.

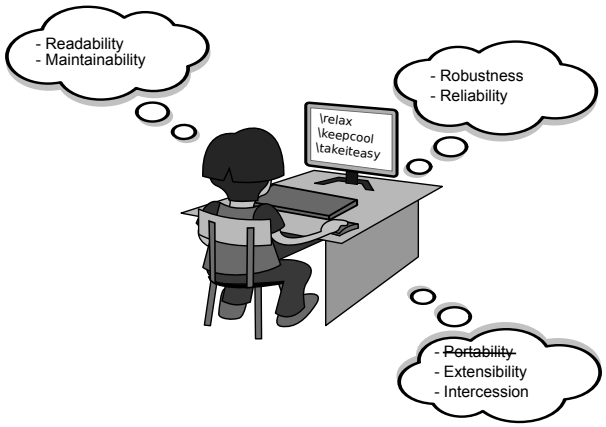

Figure 1: The coding standards many-festos

1.1 The coding standards many-festos

If one looks at the rationale behind most coding styles, the intended purpose is always to

> help programmers read and understand source code, not only their own, but that of others.

An interesting paragraph from the introductory section of the GNU Coding Standards [15] reads as follows:

> Their purpose is to make the GNU system clean, consistent, and easy to install. This document can also be read as a guide to writing portable, robust and reliable programs.

From these widely accepted views on the notion of coding style, we can draw three different points of view on the subject, as depicted in figure 1.

Human From the human point of view, using a proper coding style helps to improve the readability of source code, and as a corollary, its maintainability.

Software From the software point of view, using a proper coding style helps to make the program more robust and reliable. Note that there is a subtle but important difference between robustness and reliability. Reliability means that the program should do what it is expected to do. Robustness means that the program should handle unexpected situations as gracefully as possible.

Man-in-the-middle Third and last, the intermediate point of view is at the interface between humans and programs (note the plural). In this regard, the GNU Coding Standards mention the question of portability. This is essentially due to the fact that the GNU project mostly deals with C code, for which portability is indeed an important problem. This,

however, is much less of a concern to us because practically all LaTeX programs are inherently portable (TeX itself being mostly a virtual machine). A much more important question for us is the question of extensibility and more generally, intercession.

By extensibility, we mean to answer the following question: given a package that does *almost* what a specific user wants, is it possible to make this package provide the requested functionality without modifying its internal implementation? If the answer is yes, then the package (or at least one of its functionalities) can be said to be extensible. In this context, one of the purposes of a coding style is to help provide more, and better extensibility.

Unfortunately, full extensibility is only a utopia because ultimately, the specific desires of a user are completely unpredictable. In such situations, a package may need to be internally modified. This is what we call "intercession". The terminology comes from the more general field of so-called "reflexive" languages [10, 14]. Roughly speaking, a reflexive language provides the ability to reason about the program itself (procedural reflection) or even the language itself (behavioral reflection). Reflection is usually decomposed into "introspection" (the ability to look at yourself) and "intercession" (the ability to modify yourself).

While extension is usually a matter of user–package interaction, intercession is usually due to inter-package interactions. In the LaTeX world, we can identify three major sources of intercession.

1. LaTeX core modification: a package needs to modify LaTeX itself in order to provide the required functionality.
2. Package inter-compatibility: a package needs to co-exist with another package, and this requires modifications in either or both of them.
3. Package conflict: two (or more) packages intercede on the same piece of code but in different ways, or one package modifies some code and another package is not made aware of these modifications. In both cases, compilation breaks.

Every LaTeX user faces the "package conflict nightmare" one day or another [19], to the point that this is probably the major gripe against it these days. Consequently, we would hope that a proper coding style addresses this issue, for example by providing design patterns for graceful inter-package compatibility handling.

1.2 Consistency

One final keyword that appears quite a lot in discussions on coding style is "consistency". Given the fact that there is much personal taste in a coding style, consistency means that the exact coding style that you decide to use is actually less important than the fact of sticking to it. A person not familiar with your coding style can probably get used to it, provided that it is used consistently in the whole source code, and that for example, similar situations are identifiable as such because the same idioms are used in all of them.

1.3 30 years and no style?

Since more or less official coding standards seem to exist for many programming languages and communities, one can't help but wonder why, after 30 years of existence, the LaTeX community still doesn't have any. Several reasons come to mind.

1.3.1 Learning by example

LaTeX is not a real programming language. It is not even a macro expansion system. LaTeX is a *library*: a macro layer written on top of TeX. Because of that, the purpose of LaTeX can be anything you might want to do related to typography, which can eventually be expressed in terms of TeX. Consequently, it is impossible to write "The LaTeX programming language" book and in fact, this book doesn't exist. The things that such a book would have to describe are infinite: they depend on every user's goal. Note that the *LaTeX Companion* [12] is *not* a LaTeX programming book. For the most part, it describes some of the core functionalities, plus a lot of package features.

This explains why learning by example is an important process in the LaTeX community. It is quite easy to backtrack from a particular feature to the way it is done: you just need to look at the implementation. As a result, many LaTeX programmers (especially newcomers) start by actually looking at what other people did before us, copy-pasting or imitating functionality until they reach a satisfactory level of understanding. In doing so, they also implicitly (and unconsciously) inherit the coding style (or lack thereof) of the code they are getting inspiration from. This behavior actually encourages legacy (the good *and* the bad) and leads to a very heterogeneous code base.

1.3.2 Lack of help

Because it is only a macro library, LaTeX is not a structured language but a very liberal one. By providing such paradigms as object-oriented, functional, logic, declarative programming, *etc.*, traditional languages provide support for some "elements of style" by construction: the choice of a specific programming paradigm already imposes a particular design

on your program. On the other hand, when you program in LaTeX, you are essentially on your own. You don't get any help from the language itself.

For the same reason (lack of structure), getting help from your favorite text editor is even more complicated. Even theoretically simple things such as indentation can be an editor's nightmare. Indenting within braces in a traditional language is relatively simple: it's a matter of syntactic analysis. But suppose that you want to indent the contents of `\if<whatever>` conditionals in (La)TeX. First, this is not a syntactic construct but a macro call. Next, the closing `\fi` may be difficult to spot: it may be the result of the expansion of another macro for instance. Worse, its precise location may also depend on a dynamic (run-time) context! This shows that in general, it is impossible to get even simple things right without doing a full semantic analysis of the program, which itself may not even suffice. In a powerful development environment such as the combination of (X)Emacs [3] / AUC-TeX [1], you will typically need to indent the first line of a conditional by hand, and the rest will follow by simply hitting the TAB key. If, on the other hand, you let the editor do everything, you end up with a broken indentation layout scheme.

1.3.3 Lack of need

A survey conducted in 2010 [19] shows that LaTeX is mostly a world of dwarfs. In the TeX Live 2009 distribution, the average size of a package is 327 lines of code, the median being 134. Admittedly, very few people would feel the need for a proper coding style, when it comes to maintaining just a few hundred lines of code. In addition to that, it seems that LaTeX suffers from an anti-social development syndrome: most packages are single-authored and maintained, which leads to the same kind of consequences. When no interaction is required between people, establishing a set of coding standards for working on a common ground is a far less pressing issue.

On the other hand, imagine the difference with an industrial language in which millions of lines of code would be maintained by a team of hundreds of developers. The need for coding standards is obvious. Unfortunately, if you consider the LaTeX code base on CTAN — [2], it *is* a huge one, only maintained in a completely independent and uncontrolled fashion.

1.4 30 years and *almost* no style...

Claiming that there is no coding style for LaTeX turns out to be a slight exaggeration. By looking closely enough, we can spot a few places where the question is indeed addressed.

1.4.1 Tools

TeX itself provides some facilities for stylish programming. The equivalence between blank lines and `\par` encourages you to leave more room in your text, therefore improving its readability. TeX also conveniently ignores blanks at the beginning of lines, a crucial behavior when it comes to indenting your code without leaving spurious blanks in the generated output.

A number of packages (*e.g.* `calc` and `ifthen`) provide additional layers of abstraction on top of the LaTeX kernel, therefore providing structure where it was originally lacking. More abstraction means improved readability. Some packages like `record` even go as far as providing data structures or programming paradigms coming from other, more traditional programming languages. Of course, the difficulty here is to be aware of the existence of such packages.

1.4.2 Conventions

A number of coding conventions have existed for a long time now, including in the LaTeX kernel itself. The use of lowercase letters for user-level macros and mixed up/downcase for extension names (*e.g.* `\usepackage` *vs.* `\RequirePackage`) is one of them. The special treatment of the `@` character in macro names effectively allows one to make a clear distinction between internal and external code.

It is worth mentioning that LaTeX itself does not fully adhere to its own conventions (we see here a typical effect of legacy). For example, the macro `\hbox` is not supposed to be used externally, and hence should have been named with an `@` character somewhere. Conversely, a better name for `\m@ne` would have been `\MinusOne`.

1.4.3 Documentation

The *LaTeX Companion* contains some advice on style. Section 2.1 describes how document files should be structured and Section A.4 does the same for package source code (as we will see later, we disagree with some of the advice given there). It also mentions some of the development tools that help making source code more structured and modular (*e.g.* `doc`, `docstrip`, `ltxdoc`).

Some of these packages are described at length, although this does not count as style advice: only mentioning their existence counts, the rest is technical documentation. Modulo this remark, it turns out that the amount of style advice provided in the *Companion* is extremely limited: it amounts to less than 1% of the book.

1.5 The need for coding standards

Even though we can explain the lack of LaTeX coding standards, and even though some people certainly don't feel any need for them, we still think that they would be a valuable addition to the LaTeX world, especially in a much more developed form than what we have described in the previous section. Some important reasons for this are provided below.

Learning by *good* example We have seen earlier how LaTeX encourages "learning by example". Obviously, the existence of coding standards would help filter out poor quality code and have people learn mostly by *good* example only.

Homogeneity We have also seen how a plethora of small packages with no coding style, or author-specific ones only, contributes to make LaTeX a very heterogeneous world. This is the point at which it is important to make a distinction between coding *style* and coding *standards*. A coding style can be seen as a set of personal tastes and habits in terms of programming. A coding standard, by extension, should be defined as a coding style which multiple people agree to conform to.

In spite of the anti-social aspect of LaTeX development, that is, even if independent package developers rarely talk to each other, we know that the very high level of intercession in LaTeX implies that developers are forced to read and understand other people's code. In that context, it becomes apparent that having more-or-less official coding standards would make it easier for people to read and understand others' code. Homogeneity facilitates interaction.

One important problem here is that a consensus never comes without any concession. Coming up with coding standards that would satisfy everyone is highly unlikely, given the importance of personal taste, even if those coding standards leave room for some degree of flexibility. The question that remains open is hence the following: to what extent would people agree to comply with coding standards that diverge from their own habits or preferences, if it is for the greater good of the community?

Intercession There are many other reasons why having coding standards would be a plus. Intercession is another very important one. The way a particular package handles a particular typesetting problem only affects itself: both the problem and the solution are localized. The situation is however very different when it comes to intercession. The way a particular package handles an extension or a conflict (for example by applying dynamic modifications to another package or to the LaTeX kernel) *does* affect the outside world. As a consequence, one would expect coding standards to help clean up the current "intercession mess" by providing a set of rules, perhaps even some design patterns [4, 5, 6, 7, 9, 13] for intercession management. Intercession would become much less of a problem if every package handled it in the same way.

1.6 Coding style levels

Coding standards are supposed to help with writing better code, although we need to clarify what we mean by "better". In our personal development experience, we have identified four abstraction levels at which it is interesting to consider the notion of style. These four levels, which we are going to explore in the next sections, are the following.

1. **Layout** (low). At the layout level, we are interested in code formatting, indentation, macro naming policies, *etc.*
2. **Design** (mid). The design level deals with implementation: how you do the things that you do. This is where we address software engineering concerns such as modularity, encapsulation, the potential use of other programming languages' paradigms, *etc.*
3. **Behavior** (high). The behavior level is concerned with functionality as opposed to implementation: what you do rather than how you do it. At this level, we are interested in user interfaces, extension, intercession (notably conflict management), *etc.*
4. **Social** (meta). Finally, the social level is a meta-level at which we consider human behavior in the LaTeX community. Notions like reactivity and open development are examined.

1.7 Target audience

In the LaTeX world, it is customary to distinguish the *document* author, writing mostly text, from the *package* author (or LaTeX developer) writing mostly macros. Those two kinds of audience slightly overlap, however. By providing automatically generated text (*e.g.* language-dependent), the package author is a bit of a document author. By using packages, fixing conflicts between them and providing personal macros in the preamble, the document author is also a bit of a LaTeX developer. While this paper is mostly targeted at the package developer, many concerns expressed here (most notably at level 1: layout) are also very important for the document author.

2 Level 1: Layout

In this section, we explore the first level of style, dealing with visual presentation of source code and lexico-syntactic concerns such as naming conventions.

2.1 Formatting

One very important concern for readability is the way you visually organize your code. Most of the time, this boils down to a simple question: how and where do you use blanks. This question is more subtle to address in LaTeX than in other, more syntactically structured languages. We have identified four rules which contribute to better formatting.

2.1.1 The rules

Rule #1: Stay WYSIWYG'ly coherent

LaTeX (or TeX, for that matter) has bits of pseudo-WYSIWYG behavior. The fact that a blank line ends a paragraph is one of them. There are also some commands whose effect can be directly simulated in your source code (or document). In such situations, it is probably a good idea to do so.

The two most prominent examples of this are the `\\` and `\par` commands. Since `\\` effectively ends the current line, we find it reasonable to do so in the source file as well. Put it differently: we find it confusing when a `\\` command is immediately followed by text or code that produces text. The same goes for `\par`, with an additional remark: we have seen code in which `\par` is followed by some text, with the explicit intention of *beginning* a new paragraph. Although ending a paragraph can, in some circumstances, be equivalent to beginning the next one, this use of `\par` is extremely confusing because the its semantics are precisely to *end* the current paragraph.

Tabular-like environments are another situation in which it can be nice to mimic the actual output layout. Although it requires a fair amount of work, probably without the help of your favorite text editor, aligning the various tab commands or columns separators across rows helps readability. If, as we do, you prefer to remain within the bounds of 80 columns, a compromise may be necessary between both constraints.

Rule #2: Be "spacey" in math mode

Surprisingly enough, it seems that many people forget that spaces don't count in math mode. This is a good opportunity to take as much room as you want and make your equations easier to read. Consider the following two alternatives. This:

```
$ f(x) = f(x-1) + f(x-2) $
```

is probably better than this:

```
$f(x)=f(x-1)+f(x-2)$
```

Rule #3: One "logical" instruction per line

This rule may be a wee bit fuzzier than the previous ones. By "logical", we roughly mean something (a code sequence) which makes sense as a whole. In traditional programming languages, a logical instruction is generally a function call along with its arguments, or an operator along with its operands. In LaTeX, the situation is more complicated, notably because of the throes of macro expansion.

Perhaps it is simpler to make this point by providing some examples. We assume here that our logical instructions are small enough to fit on one line, the idea being to avoid putting two of them next to each other.

```
\hskip.11em\@plus.33em\@minus.07em
```

This line constitutes only one logical instruction because the sequence of macros and quantities define a single length.

```
{\raggedleft\foo\bar baz\par}
```

Here, the flushing instruction applies until the closing brace (assuming the paragraph ends before the group), so it would be strange, for instance, to go to the next source line after `\bar`. Note however that for longer contents, not fitting on one line only, we would probably go to the next line right after `\raggedleft`, so that the formatting instruction(s) are distinct from the text to which they apply.

In the same vein, it would be unwise to split things like `\expandafter\this\that`, conditional terms such as `\ifx\foo\bar`, and more generally, everything that can be regarded as an argument to what precedes.

Rule #4: Indent all forms of grouping

This rule is probably the most obvious, and at the same time the most important, when it comes to readability. *All* forms of grouping should have their contents indented so that the beginning and end of the groups are clearly visible. It seems that indenting by 2 columns is enough when you use a fixed width font, whereas 4 or 8 columns (or a tab) are necessary otherwise. In general however, using tab characters for indentation is inadvisable (notably in document sources, but sometimes in package sources as well). Tabs can be dangerous, for instance, when you include code excerpts that should be typeset in a special way.

In LaTeX, grouping can occur at the syntactic level with group delimiters (`{}`, `[]`) or math modes (`$` and `\(\)`, `$$` and `\[\]`), and also at the semantic level (`\bgroup` / `\egroup`, `\begingroup` / `\endgroup` or even `\makeatletter` / `\makeatother`). Your favorite text editor will most likely help you indent at the syntactic level, but you will probably need to do some manual work for semantic grouping. In the case of Emacs for example, manually indenting the first line below a call to `\makeatletter` is usually

```
1  %% Original version:
2  \def\@docinclude#1 {\clearpage
3  \if@filesw \immediate\write\@mainaux{\string\@input{#1.aux}}\fi
4  \@tempswatrue\if@partsw \@tempswafalse\edef\@tempb{#1}\@for
5  \@tempa:=\@partlist\do{\ifx\@tempa\@tempb\@tempswatrue\fi}\fi
6  \if@tempswa \let\@auxout\@partaux \if@filesw
7  \immediate\openout\@partaux #1.aux
8  \immediate\write\@partaux{\relax}\fi
9  % ... \fi :-(
10
11 %% Reformatted version:
12 \def\@docinclude#1{%
13   \clearpage
14   \if@filesw\immediate\write\@mainaux{\string\@input{#1.aux}}\fi
15   \@tempswatrue
16   \if@partsw
17     \@tempswafalse
18     \edef\@tempb{#1}
19     \@for\@tempa:=\@partlist\do{\ifx\@tempa\@tempb\@tempswatrue\fi}%
20   \fi
21   \if@tempswa
22     \let\@auxout\@partaux
23     \if@filesw
24       \immediate\openout\@partaux #1.aux
25       \immediate\write\@partaux{\relax}%
26     \fi
27   % ... \fi :-)
```

Figure 2: The virtues of proper formatting

enough to have the subsequent ones follow the same indentation level automatically. But then again, you will also need to manually *unindent* the closing call to `\makeatother`.

As an illustration of both rules #3 and #4, consider the code in figure 2 in both original and reformatted form. In each case, ask yourself: to which conditional does the upcoming `\fi` belong? This point clearly demonstrates the importance of indentation. Line 14 contains an example of what we called a "logical" instruction, although a longer one this time. The contents of the conditional is a single instruction to write something in the auxiliary file immediately. Also, since there is no `\else` part in this conditional and the whole line doesn't exceed 80 columns, we chose to keep it as a one-liner. The same remark can be made for line 19.

2.1.2 Formatting of syntactic groups

In the case of syntactic groups, various policies can be observed regarding the position of the braces (this is also true in other programming languages). The case of an environment definition could be formatted as follows, as is done on several occasions in the LaTeX standard classes:

```
\newenvironment{env}
               {\opening\code
                \opening\code}
               {\closing\code
                \closing\code}
```

We find this kind of formatting somewhat odd and it doesn't seem to be used so frequently anyway. The conspicuous amount of indentation can be disturbing, and it is also a bit difficult to visually distinguish the opening argument from the closing one.

A more frequent way of formatting this would be more or less as in a piece of C code, as follows:

```
\newenvironment{env}
{%
  \opening\code
  \opening\code
}
{%
  \closing\code
  \closing\code
}
```

This kind of formatting is admittedly more readable, although the two nearly empty lines between the opening and the closing arguments may be considered somewhat spurious. Some people hence take the

```
\newcommand\text{%
  \@nextentry
  \noalign\bgroup
    \gdef\@beforespace{...}%
    \@ifstar{\@stext}{\@text}}

\newcommand\@text[1]{%
    \gdef\@nextentry{}%
  \egroup% end of \noalign
  \multicolumn{3}{@{}p ... \\}}

\newcommand\@stext{%
    \gdef\@nextentry{\egroup\\\par}%
  \egroup% end of \noalign
  \multicolumn{3}{@{}p ...} ...}
```

Figure 3: Inter-macro indentation

habit of joining those two lines as follows:

```
\newenvironment{env}
{%
  \opening\code
  \opening\code
}{%
  \closing\code
  \closing\code
}
```

Other people choose a more compact formatting by closing a group, and possibly opening the next one on the same line, as follows:

```
\newenvironment{env}{%
  \opening\code
  \opening\code}{%
  \closing\code
  \closing\code}
```

Again, this leads to quite compact code that makes it difficult to visually distinguish the opening argument from the closing one. In such a case, a possible workaround is to introduce comments, also an opportunity for documenting the macro's prototype (imagine that in a text editor with fontification, you might also have different colors for code and comments):

```
\newenvironment{env}{%
  %% \begin{env}
  \opening\code
  \opening\code}{%
  %% \end{env}
  \closing\code
  \closing\code}
```

2.1.3 Inter-macro indentation

The case of semantic grouping introduces an additional level of complexity because groups may be opened and closed in different macros (worse: the opening and closing instructions may themselves be the result of macro expansion). When possible, it is a good idea to preserve the amount of indentation corresponding to the current group nesting level, even if the group in question is not syntactically apparent.

Consider for example the code in figure 3 taken from the *CurVe* class [17]. The `\text` command calls `\noalign`, but the argument passed to `\noalign` (enclosed in `\bgroup`/`\egroup`) starts here and ends in either `\@text` or `\@stext`. You can see that this group's indentation level is preserved across all three macros.

2.1.4 Exceptional situations

No rule goes without exception. Sometimes, and for the greater good, one might be tempted to go against the established rules. Here are two examples.

Consider the following call to `\@ifnextchar`:

```
\@ifnextchar[%] syntax screwup!
  {\@dothis}{\@dothat}
```

The left square bracket, which is in fact the first argument of `\@ifnextchar`, confuses Emacs because it thinks it's the opening of a group, and expects this group to be closed somewhere. In order to compensate for this problem, we usually virtually close the fake group by putting a right square bracket within a comment on the same line. This forces us, however, to provide the "then" and "else" arguments to `\@ifnextchar` on the next line, something that we would normally not do.

Another exceptional situation is the case of empty macro arguments, where we prefer to stay on the same line rather than consuming another one just for an empty pair of braces, as illustrated below:

```
\@ifundefined{#1note}{}{%
  \@fxpkgerror{a short explanation}{%
    a longer one}}
```

2.1.5 Corollary

As a corollary to the rules described in this section, it is essential to note that the `%` character is your "worst best friend". A very important problem when writing macros (and even documents) is the risk of spurious blank spaces. When you indent your code properly, many blanks are inserted, which are not supposed to appear in the final document. TEX helps you with that in several ways: spaces are eaten after a control sequence, consecutive blanks are treated as only one (this includes the newline character), and leading / trailing spaces are discarded on every line.

That alone, however, is not sufficient for a liberal indentation scheme. In the previous examples, we have seen many places (notably after braces) where

it is required to create an end-of-line comment with the `%` character, so that the final newline character is not taken as a textual one (see for example figure 3 on the previous page).

In that sense, the `%` character is your best friend. It is also your worst friend because determining the exact places at which an end-of-line comment is required is far from trivial. There are even cases where it could be necessary after opening an environment in a final document! In any case, when there are blanks in your source that you *know* you don't want in the output, and you're unsure whether TEX will skip them on its own, you can safely always insert a comment character at the end of the line.

2.2 Naming

The second concern we want to address in this section is that of naming schemes. Naming conventions are obviously important for readability, but also for backward compatibility. Once you get a name, it's for life. Starting with bad naming conventions can become a major headache, both for your clients (using your API) and yourself (maintaining your own code).

2.2.1 The rules

Rule #1: Use prefixes

Because LATEX lacks a proper notion of module, package, or even namespace, the use of a specific prefix for every package that you write should be a rule of thumb. For example, our FiXme [18] package uses `fx` as a prefix, which means that every command (but see rule #3) starts with those two letters.

The choice of the prefix is also important. In theory, the prefix that would guarantee a minimal risk of name clash between packages would be the full package name. In practice however, this can lead to very long macro names, cumbersome to type. Therefore, a trade-off must be made between the prefix length and its uniqueness (a possible idea is to start by removing the vowels). `fx` for example has the defects of its qualities: it is practical because it is very short, but the risk of collision with only two letters is not negligible.

Once you have chosen a prefix, it is also important to stay consistent and stick to it. Recently, we discovered that for some obscure (and forgotten) reason, our ***FiNK*** package uses a prefix of `fink` for its user-level commands, but only `fnk` for its internal ones. This is not only rinadvisable but also unnecessary since LATEX already provides the `@` character convention for making such a distinction (*cf.* rule #3).

One situation where the prefix rule should probably be relaxed is the case of classes (as opposed to styles). Classes, by definition, are mutually-exclusive and perform similar, very general tasks, although in different ways. It would hence be silly to have to name similar things differently (imagine for instance that the sectioning commands were named `\artsection`, `\rprtsection` and `\bksection`!). On the other hand, the risk of collision is still high, precisely because of the broad spectrum of class functionality. This problem has already befallen us in the ***CurVe*** class, which provides a `\text` macro, also implemented (to do something different) by `siunitx` and probably other packages. `\text` is the perfect example of a very poor choice of name because it is far too general and doesn't really mean anything. This demonstrates that choosing a pertinent, unique and concise name for a macro is an important but tricky exercise.

Rule #2: Use postfixes

In a very analogous way, there are situations in which the use of a postfix may be a good idea in order to avoid name clashes, although this time not with other packages, but with yourself. LATEX provides a number of concepts, loosely related to data types or structures, such as counters and saveboxes. Unfortunately, the provided interfaces are rather inconsistent.

In some situations like counters, you are only required to provide a name, and LATEX constructs the underlying, opaque macros with a specific naming scheme. What's more, you are not supposed to use those macros explicitly. Suppose for example that you want to maintain a counter of "items". There is no need to name this counter `myitemscount` because the standard interface makes things perfectly readable without the postfix:

```
\newcounter{myitems}
... \value{myitems} % not a very good name
... \stepcounter{myitems}
```

Besides, the risk of name clash is minimal because under the hood, LATEX has used a specific and hopefully unique naming scheme for naming the counter macro (`\c@myitems`).

Suppose now that you want to save your items in a box. In that case, you are requested to provide a macro name yourself, and choosing `\myitems` is for sure a bad idea because that name is too general (there is no indication that you're talking about the *box* of them, and not the number, list or whatever else of them). What you need to do, therefore, is decide on a specific naming scheme for boxes, just as LATEX does under the hood for counters. Using a `box` postfix appears to be a good solution:

```
\newsavebox\myitemsbox
... \savebox\myitemsbox{...}
... \sbox\myitemsbox{...}
```

Of course, there is some naming redundancy in this code, but that is what you get from an interface that is not as opaque as it should be.

If you are developing a package (as opposed to a document) and want to maintain an *internal* list of items, you may also be tempted to follow LaTeX's own convention for, say, counters, and call your macro `\b@myitems` or something like that. We advise against that, however, because it conflicts with the prefix rule described previously, and also because it would make your code less readable (remember that you need to use the macro explicitly, not just the "name of the thing").

Finally, note that the ultimate solution to this kind of problem would be to develop another, properly abstracted layer on top of the original one, in which the actual macro names are never used explicitly, and standardize on it...

Rule #3: Obey the *Companion*

The *LaTeX Companion* provides some advice on naming in section A.1. Modulo a substantial amount of legacy code, LaTeX itself tries to adhere to the naming conventions described there so it is a good idea to honor them in your packages as well. For starters, you are invited to name your external macros with lowercase letters only, and reserve a mixture of lowercase and uppercase names for extension APIs. FiXme, for example, follows this convention by providing an end-user command named `\fxuselayout`, and at the same time an equivalent command named `\FXRequireLayout` for theme authors.

The other important and well known naming convention adopted by LaTeX is the use of an `@` character in internal macro names. By turning this character into a *letter* (category code 11) only internally and in packages, LaTeX effectively prevents the document author from using such macros directly (one would have to intentionally enclose a call to an `@`-macro within `\makeatletter` / `\makeatother`).

Third-party packages should obviously follow this convention in order to separate internal from external macros. Package authors should however do a better job at naming internal macros than LaTeX itself (again, we see here the effect of a long legacy). The LaTeX kernel seems to enjoy making fun of the `@` character, using it in place of different vowels (*e.g.* `\sixt@@n` or `\@filef@und`) and with no apparent rationale in terms of number and position (*e.g.* `\@input`, `\@@input` but `\@input@`).

Although we underst@nd how this c@n be fun, it is better for readability to keep a more systematic approach to naming internal macros. Typically, we find that using the `@` character is useful in two situations:

```
\DeclareRobustCommand\fxnote{%
  %% ...
  \@ifstar{%
    %% \fxnote*
    \@ifnextchar[%]
      {\@fxsnote{#2}}
      {\@@fxsnote{#2}}}{%
    %% \fxnote
    \@ifnextchar[%]
      {\@fxnote{#2}}
      {\@@fxnote{#2}}}}

\long\def\@fxsnote#1[#2]#3#4{%
  %% ...
  \@@fxsnote{#1}{#3}{#4}}

\long\def\@@fxsnote#1#2#3{%
  \implement\me}
```

Figure 4: Nesting levels

- as a prefix to indicate an internal implementation of an external functionality,
- as a word separator.

For example, the current (language-dependent) value for the "List of FiXme's" section name is stored in a macro named `\@fxlistfixmename` (an acceptable alternative would be `\fx@listfixmename`).

In some situations, the implementation of a particular feature may go through different levels of indirection. In such cases, we like to use multiple `@` characters to give an indication of the current implementation level. Figure 4 illustrates this. The macro `\fxnote` supports an optional `*` postfix as well as a regular optional first argument provided in square brackets. The implementation goes through a first sub-level that detects the presence of a `*` character (`\@fxnote` / `\@fxsnote`), plus another sub-level which handles the presence of an optional argument (`\@@fxnote` / `\@@fxsnote`).

A final example is the case of "polymorphic" macros (see section 3.3 on page 319), that is, macros whose implementations depend on some context. As mentioned earlier, the `@` character can be used to separate words. For instance, FiXme has a macro named `\@@@fxnote@early`. This macro is polymorphic in the sense that its actual implementation varies according to the document's draft or final mode. The two corresponding effective implementations are named `\@@@fxnote@early@draft` and `\@@@fxnote@early@final`.

2.2.2 Exceptional situations

From time to time, the naming rules exhibited in the previous section may be bypassed for the sake of

readability. Here are three typical situations where exceptions are in order.

Conforming to *de facto* standards LaTeX itself has some naming conventions that may impact a package or even a document author. Lists are one such case. For example, the behavior for the list of figures depends on two macros named `\listoffigures` and `\listfigurename`. FiXme supports its own list facility, and for the sake of coherence, provides analogous macros named `\listoffixmes` (instead of `\fxlist` or some such) and `\listfixmename` (instead of `\fxlistname`). Following the usual convention makes it much easier for your users to remember your own API.

Another example is that of conditionals. *All* conditionals in (La)TeX are named `\if`⟨*something*⟩. So here again, given that you need to implement `mycondition`, it is better to name your conditional `\ifmycondition` than `\myifcondition`.

Forced exceptions There are times where LaTeX itself will force you to depart from your own rules, although this is seldom critical. The case of counters is one of them. When creating a counter for `myitems`, LaTeX creates a macro named `\c@myitems` which is not how you would have named this macro. However, this is not such a big deal because in general, you don't need to use this macro directly.

A slightly more intrusive exception is when LaTeX requires that you implement a specific macro, following its own naming scheme. For instance, supporting a list of FiXme's involves implementing a macro named `\l@fixme`. The `l@` prefix is LaTeX's choice, not ours.

Finally, if you implement an environment named `myenv`, LaTeX will eventually turn this into a macro named `\myenv` and another one named `\endmyenv`. Here again, the names are LaTeX's choice, not yours. And by the way, it is unfortunate that the environment opening macro is not named `\beginmyenv` instead of just `\myenv` because it means that you can't have both a command and an environment with the same name. In the FiXme package, we use a nice naming trick for this kind of situation: environments corresponding to macros are prefixed with "`a`" or "`an`". For example, there is a macro named `\fxnote` and the corresponding environment is named `anfxnote`. This contradicts our own naming conventions but it makes the actual environment usage as readable as if it were plain English:

```
\begin{anfxnote}
...
\end{anfxnote}
```

3 Level 2: Design

In this section, we explore the second level of style, dealing with design considerations such as modularity and other programming paradigms. From a more practical point of view, *design* here is concerned with how to implement a particular feature, rather than the feature itself.

3.1 Rule #1: Don't reinvent the wheel

3.1.1 Feature libraries

In many programming languages, so-called "standard libraries" provide additional layers of functionality, typically functions that perform useful and frequently needed treatments. Browsing CTAN [2] clearly demonstrates that LaTeX is no exception to this rule. People have created packages for making slides, curricula vitae, split bibliographies, tables that span across several pages, *etc.*

When you develop a package, it is important, although not trivial, to be aware of what's already existing in order to avoid reinventing the wheel. For instance there are currently at least half a dozen different solutions for implementing key-value interfaces to macros (`keyval`, `xkeyval`, `kvoptions`, `pgfkeys`, *etc.*). This is very bad because each solution has its own strengths and weaknesses, so the choice of the most appropriate one for your personal needs can be very complicated and time-consuming (in fact, there might not even be a *best* choice).

One rule of thumb is that when you feel the need for implementing a new functionality, someone most probably had the same idea before you, so there is a good chance that you will find a package doing something close to what you want. In such a case, it is better to try and interact with the original author rather than to start over something new on your own. Doing this, however, also requires some rules in terms of social behavior (*cf.* section 5 on page 326).

3.1.2 Paradigm libraries

Furthermore, in the LaTeX world the notion of standard library goes beyond common functionality: it goes downwards to the language level. TeX was not originally meant to be a general purpose programming language, but TeX applications today can be so complex that they would benefit from programming paradigms normally found in other languages. Because of this, there are packages that are meant to extend the language capabilities rather than providing a particular (typesetting) functionality. The two most prominent examples of this are `calc` and `ifthen`. These packages don't do anything useful in terms of typesetting, but instead make the pro-

grammer's life easier when it comes to arithmetic calculations or conditional branches. Another one, called `record`, even goes as far as providing data structures for LaTeX.

It is always a good idea to use these packages rather than doing things at a lower level, or reinventing the same functionality locally. The more abstract your code, the more readable. The *LaTeX Companion* advertises some of them (notably `calc` and `ifthen`). Of course, the difficult thing is to become aware of the existence of such packages (CTAN contains literally thousands of packages).

3.2 Rule #2: Duplication/Copy-paste is evil

This rule is well-known to every programmer, although the "evilness" threshold may be a bit subtle to calculate. It is also interesting to provide some insight on the distinction we make between "duplication" and "copy-paste". The two examples below will shed some light on these matters.

3.2.1 Duplication

Consider the case of FiXme which uses the `xkeyval` package for defining several layout-related package options. The bad way of doing it would be as follows:

```
\define@key[fx]{layout}{morelayout}{...}
\define@cmdkey[fx]{layout}{innerlayout}{...}
\define@key[fx]{envlayout}{envlayout}{...}
```

This is bad because the `[fx]` optional argument (the prefix in `xkeyval` terminology) is *duplicated* in every single call to the `xkeyval` package (and it is rather easy to forget).

This is a typical case where duplication should be abstracted away in order to avoid redundancy. We can improve the code by providing *wrappers* around `xkeyval` as follows:

```
\newcommand\@fxdefinekey{\define@key[fx]}
\newcommand\@fxdefinecmdkey{\define@cmdkey[fx]}
\@fxdefinekey{layout}{morelayout}{...}
\@fxdefinecmdkey{layout}{innerlayout}{...}
\@fxdefinekey{envlayout}{envlayout}{...}
```

It should be noted that this new version is actually *longer* than the previous one. Yet, it is clearer because more abstract. Using such wrappers is like saying "define a FiXme option". This is more abstract than "define an option which has an `fx` prefix".

Note also that in this example, two "layout" options are defined. One could hence be tempted to abstract the layout family, for example by providing an `\@fxdefinelayoutkey` command. We decided not to do this but it could be a legitimate choice. This is an illustration of the flexibility and perhaps also the difficulty there is to decide on the exact "evilness duplication threshold" mentioned earlier.

3.2.2 Copy-paste

Consider again the case of FiXme which defines several Boolean options. For each Boolean option `foo`, FiXme also provides a corresponding `nofoo` option, as a shortcut for `foo=false`. *E.g.* the `langtrack` / `nolangtrack` option can be defined as follows:

```
\@fxdefineboolkey{lang}{langtrack}[true]{}
\@fxdefinevoidkey{lang}{nolangtrack}{%
  \@nameuse{fx@lang@langtrack}{false}}
```

Defining the `silent` / `nosilent` option can be lazily done by *copy-pasting* the previous code and only modifying the relevant parts (the option and family names):

```
\@fxdefineboolkey{log}{silent}[true]{}
\@fxdefinevoidkey{log}{nosilent}{%
  \@nameuse{fx@log@silent}{false}}
```

This way of doing things obviously screams for abstraction. It is better to make the concept of "extended Boolean" explicit by providing a macro for creating them:

```
\newcommand*\@fxdefinexboolkey[3][]{%
  \@fxdefineboolkey{#2}{#3}[true]{#1}
  \@fxdefinevoidkey{#2}{no#3}{%
    \@nameuse{fx@#2@#3}{false}}}

\@fxdefinexboolkey{lang}{langtrack}
\@fxdefinexboolkey{log}{silent}
```

3.3 Rule #3: Conditionals are evil

This rule may sound surprising at a first glance, but experience proves that too many conditionals can hurt readability. In fact, this is well known in the object-oriented community. After all, object-orientation is essentially about removing explicit conditionals from code.

There are two main reasons why conditionals should be avoided whenever possible.

- First, too many conditionals, especially when they are nested, make the program's logic difficult to read.
- Second, the presence of multiple occurrences of the *same* conditional at different places is a form of duplication, and hence should be avoided.

One particular design pattern that helps a lot in removing explicit conditionals is to centralize the logic and use polymorphic macros. This is explained with the following example.

Figure 5 on the next page implements a macro `\doeverything`, the behavior of which depends on whether the document is in draft or final mode. This macro is in fact decomposed in three parts: the "do this" part, a middle part (left as a comment) and a final "do that" part. Because the same conditional

```
\newif\ifdraft

\def\doeverything{%
  \ifdraft
    \dothis\this\way
  \else
    \dothis\this\other\way
  \fi
  %% ...
  \ifdraft
    \dothat\that\way
  \else
    \dothat\that\other\way
  \fi}

\DeclareOption{draft}{\ifdrafttrue}
\DeclareOption{final}{\ifdraftfalse}
\ExecuteOptions{final}
\ProcessOptions
```

Figure 5: Conditional duplication

```
\def\dothis@draft{\this\way}
\def\dothis@final{\this\other\way}

\def\dothat@draft{\that\way}
\def\dothat@final{\that\other\way}

\def\doeverything{%
  \dothis
  %% ...
  \dothat}

\DeclareOption{draft}{
  \let\dothis\dothis@draft
  \let\dothat\dothat@draft}
\DeclareOption{final}{
  \let\dothis\dothis@final
  \let\dothat\dothat@final}
\ExecuteOptions{final}
\ProcessOptions
```

Figure 6: Centralized logic

branch clutters the code in two different places, the three-step nature of this macro is not very apparent.

A better and clearer implementation of the same functionality is proposed in figure 6. Here, the two mode-dependent parts of the `\doeverything` macro are explicitly implemented in different macros, with a postfix indicating in which mode they should be used. In the `\doeverything` macro, the three parts are now clearly visible. This new version of the macro is obviously much more concise and readable. The important thing to understand here is that when you read the code of `\doeverything`, you are in fact not concerned with implementation details such as how `\dothis` and `\dothat` vary according to the document's mode. It is more important to clearly distinguish the three steps involved.

Finally, you can also note that the logic involving conditionals is centralized at the end, where the actual `draft` or `final` options are processed. As a side note, the `\ifdraft` conditional is not needed anymore and the total amount of code is smaller in this new version. This time, clarity goes hand in hand with conciseness.

You may still be wondering what we meant by "polymorphic macros". Although slightly abusive, this term was coined because of the resemblance of this design pattern with object-oriented polymorphism, encountered in virtual methods à la C++ or generic functions à la Lisp. The macros `\dothis` and `\dothat` are polymorphic in the sense that they don't have a regular implementation (in other words, they are only *virtual*). Instead, their actual implementation varies according to some context.

3.4 Rule #4: Be modular

Modularity is another final principle which is rather obvious to follow, although it is perhaps even more crucial in LaTeX than in other programming languages. Modularity affects all levels of a document, from the author's text to the packages involved.

At the author's level, it is a good idea to use LaTeX's `\include` command and split your (large) source files into separate chunks. When used in conjunction with `\includeonly`, compilation may be considerably sped up by making LaTeX process only the parts on which you are currently working.

From a package development perspective, modularity is important at different levels. In terms of distribution, the `docstrip` package is an essential component in that it allows you to split your source code into separate files, provides conditional inclusion and (and perhaps most importantly) lets you generate separate distribution files from a centralized source. This is important because splitting a package across different files allows you to subsequently load only the relevant ones. Less code loaded into LaTeX means reduced memory footprint and improved performance. Imagine for instance if Beamer had to load every single theme every time it is run!

At a lower level, the modularity principle dictates that it is better to have 10 macros of 10 lines each rather than one macro of 100 lines. Every programmer knows this but perhaps LaTeX programmers don't realize that this is even more critical for them. There is indeed one LaTeX-specific reason for keeping your macros small. That reason is, again, interces-

sion. Since other package developers may need to tweak *your* code for compatibility reasons, it is better to let them work on small chunks rather than big ones.

To illustrate this, let us mention the case of CurVe in which, at some point, we decided to support the `splitbib` package. In order to do so, we needed to override some parts of `splitbib`'s macro `\NMSB@writeentry`. This macro was originally 203 lines long. After dead branch removal, that is, after cutting out pieces of code that we knew would never be executed in the context of CurVe, we ended up with 156 lines that needed to be imported into CurVe, only to modify 5 of them. Our modifications consequently involve only 3% of the code that needed to be imported. One can easily imagine how bad this is in terms of maintainability. We need to keep track of potential modifications on 203 lines of `splitbib` just to make sure our 5 keep functioning correctly.

4 Level 3: Behavior

In the previous section, we claimed to be more concerned with how to implement particular features, rather than the features themselves. In this section, we focus on features through the lens of behavior. What we are interested in is the impact of your package features on the people that may interact with it.

4.1 Rule #1: Be user-friendly

The first category of people who will interact with your package is its end users. Hopefully, you belong to this category as well. There is, however, one major difference between you and other users: you know the package much better than they do, since you wrote it. Being user-friendly means doing everything possible to make your package easy to use. This can mean many different things, but two important aspects are documentation and backward compatibility.

4.1.1 Documentation

Nowadays, the vast majority of LaTeX packages comes with documentation. The combination of `doc`, `ltxdoc` and `docstrip`, by allowing for literate programming, has greatly helped the community in this respect. Nevertheless, there remains a huge difference between documentation and *good* documentation.

The difficulty in writing good documentation is to put yourself in the position of the casual user — which you are not because you know the package so well already. Thinking from a user perspective is probably the most difficult thing to do, but it can also be a very rewarding experience (we will get back to this later).

One of the major pitfalls to avoid when writing documentation is forgetting that a user manual is not the same thing as a reference manual. Just doing literate programming is not enough. Documenting your macros around their implementation is not enough. The casual user is not interested in the brute list of commands, nor in the internals of your package. The casual user wants an overview of the package, what it is for, what it can do, what it can't, probably a quick start guide describing the entry points and the default behavior, with examples. Only then, when the major concepts are understood, you may delve into complexity and start talking about customization, additional but less important features, and so on.

A good user manual will sacrifice sufficiency to the benefit of gradualness and redundancy. You shouldn't be afraid of lying by omission to the readers. It is for their own good. They don't want to be overwhelmed by information. A typical hint that you are reading a bad manual is when the documentation starts with the full list of package options. There is no point in introducing an option dealing with a concept that the reader does not understand yet (that would be a reference manual). Another hint is when a manual starts referring to another package (that it happens to use internally) and assumes that the reader knows everything about it already. The end user shouldn't have to read two or three other manuals to understand yours, especially if in the end, they will never use those other packages directly.

Why, as we said earlier, can it be rewarding to write a good manual? Because writing documentation is in fact a *feedback loop*. The difficult thing, again, is to put yourself in the position of someone who knows nothing about the things you are going to talk about, and ask yourself: "what do I need to say first?" If you can do that, you will discover that many times, the answers to that question reveal design flaws in your package, its design or its APIs. Things that a casual user would want to do but can't, things that should be simple to do but aren't, default behavior that shouldn't be by default, concepts that are not apparent enough, not distinct enough, names that are not sufficiently self-explanatory. *Etc.* other words, writing or improving the quality of your manual often helps you improve the quality of your code, and *vice-versa*.

4.1.2 Backward compatibility

Documentation is an important feature. Backward compatibility is another. Users can get very frustrated when a package breaks their documents from one version to another, or more generally, when a document doesn't compile anymore after a couple

of years. This was in fact a concern that Donald Knuth had in mind at the very beginning of TEX and which had a considerable influence on the design of the LATEX Project Public License [11], the LPPL.

Maintaining backward compatibility often goes against the "greater good". The natural evolution of a design might require a complete change of API, or at least an important amount of hidden trickery in order to stay compatible with the "old way". That is why it is all the more important to take great care with the design right from the start.

Just as in the case we made for modularity, the very high level of intercession in LATEX makes backward compatibility an even more important concern. Because other developers will interfere with your code in order to fix compatibility or conflict problems between your package and theirs, the changes you make in your internals *will* affect them as well. So it turns out that backward compatibility is not only a surface concern, but also something to keep in mind even when working on the inner parts of your code. In the LATEX world, nothing is really private... Of course, you may decide not to care about that, pretending that it's the "other guy's responsibility" to keep up to date with you, as he's the one who messes up with your code. But this is not a productive attitude, especially for the end user of *both* packages. In that regard, the following excerpt from `hyperref`'s README file is particularly eloquent:

> *There are too many problems with varioref. Nobody has time to sort them out. Therefore this package is now unsupported.*

In order to balance this rather pessimistic discourse, let us mention two cases where the burden of backward compatibility can be lightened. The first is the case of packages focused on the development phase of a document. FiXme is one of them. As it is mostly dedicated to handling collaborative annotations to draft documents, the cases where you would want to keep traces of it in a finished document are rare. Under those circumstances, we would not care about backward compatibility in FiXme as much as in other packages. For a document author perspective, it is very unwise to upgrade a LATEX distribution in the middle of the writing process anyway...

When you decide that backward compatibility is too much of a burden, it is still possible to smooth the edges to some extent. Here is an idea that we are going to use for the next major version of *CurVe*: change the name of the package, possibly by postfixing the (major) version number. In our case, the current version of *CurVe* (the `1.x` series) will be declared deprecated although still available for download, and the next version will be available in a package named `curve2`. This way, former *CurVe* documents will still compile in spite of backward incompatible changes to the newest versions.

```
\ExecuteOptionsX[my]<fam1,...>{opt1=def1,...}
\ProcessOptionsX*[my]<fam1,...>

\newcommand*\mysetup[1]{%
  \setkeys[my]{fam1,...}{#1}}

\newcommand\mymacro[2][]{%
  \setkeys[my]{fam1,...}{#1}%
  ...}
```

Figure 7: `xkeyval` programming example

Even if you do so, as a convenience to your users, it might still be a good idea to decorate your manual with a transition guide from one version to the next.

4.1.3 Key-value interfaces

We mentioned already the importance of feature design and the effect it can have on backward compatibility. The case of key-value interfaces is a typical example. Implementing package or macro options in a `key=value` style is a feature that every package should provide nowadays.

Key-value options are user-friendly because they are self-explanatory and allow you to provide a flexible API in a uniform syntax. It is better to have one macro with two options and 5 values for each rather than 25 macros, or 5 macros with 5 possible options.

As usual, the difficulty is in knowing all the existing alternatives for key-value interface, and choosing one. For that, Joseph Wright wrote a very useful paper that might help you get started [20]. Once you get used to it, programming in a key-value style is not so complicated.

Figure 7 demonstrates how easy it is to empower your package with key-value options both at the package level and at the macro level with `xkeyval`. Assuming you have defined a set of options, you can install a default behavior with a single macro call to `\ExecuteOptionsX`, and process `\usepackage` options with a single call to `\ProcessOptionsX`. Several packages provide a "setup" convenience macro that allows you to initialize options outside the call to `\usepackage`, or change the default settings at any time in the document. As you can see, such a macro is a one-liner. Similarly, supporting a key-value interface at the macro level is also a one-liner: a single call to `\setkeys` suffices.

In order to understand how key-value interfaces provide more flexibility and at the same time make

backward compatibility less of a burden, consider one of the most frequently newbie-asked questions about LaTeX: how do I make a numbered section which does not appear in the table of contents (TOC)? The general answer is that you can't with the standard interface. You need to reprogram a sectioning macro with explicit manipulation of the section counter, *etc.*

We know that `\section` creates a numbered section which goes in the TOC. We also know that `\section*` creates an unnumbered section that does *not* go in the TOC. Finally, the optional argument to `\section` allows you to provide a TOC-specific (shorter) title. So it turns out that there's no standard way to extend the functionality in a backward-compatible way, without cluttering the syntax (either by creating a third macro, or by providing a new set of options in parentheses for instance). In fact, two macros for one sectioning command is already one too many.

Now imagine that key-value interfaces existed when `\section` was designed. We could have ended up with something like this:

```
% Number and TOC:
\section{Title}

% TOC-specific title:
\section[toctitle={Shorter Title}]{Title}

% Unnumbered and not in the TOC:
\section[numbered=false]{Title}
```

Obviously here, we intentionally reproduce the same design mistake as in the original version: assuming that unnumbered also implicitly means no TOC is suboptimal behavior. But in spite of this deficiency, when somebody wanted a numbered section not going in the TOC, we could have added a new option without breaking anything:

```
\section[toc=false]{Title}
```

What's more, we could also handle the opposite request for free: an unnumbered section still going in the TOC:

```
\section[numbered=false,toc=true]{Title}
```

4.2 Rule #2: Be hacker-friendly

The second category of people who will interact with your package is its "hackers", that is, the people that may need to examine your code or even modify it for intercession purposes. Of course, you are the first person to be in this category. Being hacker-friendly means doing everything possible to make your package easy to read, understand and modify. Note that as the first person in this category, you end up doing yourself a favor in the first place. Being hacker-friendly can mean many different things, including concerns that we have described already, such as modularity. In this section we would like to emphasize some higher level aspects, notably the general problem of code organization. In our experience, we find that organizing code in a bottom-up and feature-oriented way works best.

4.2.1 From bottom to top

Organizing code in a bottom-up fashion means that you build layers on top of layers and you organize those layers sequentially in the source file(s). The advantage in being bottom-up is that when people (including yourself) read the code, they can rely on the fact that what they see only depends on what has been defined above (previously). Learning seems to be essentially an incremental process. Reading is essentially a sequential process. Being bottom-up helps to conform to these cognitive aspects.

The bottom-up approach is sometimes confused with the design of a hierarchical model in which one tries to establish nested layers (or rings) of functionality. These are different things. For example, not all problems can be modeled in a hierarchical way and trying to impose hierarchy leads to a broken design. Sometimes, it is better to be modular than hierarchical. Some concepts are simply orthogonal to each other, without one being on top of the other. The bottom-up approach allows for that. When you have two orthogonal features to implement, you can just write them down one after the other, in no particular order. The only rule is that one feature depends only on the preceding, or more precisely, a subset of the preceding.

As a code organization principle, the bottom-up approach will also inevitably suffer from a few exceptions. Any reasonably complex program provides intermixed functionality that cannot be implemented in a bottom-up fashion. Macro inter-dependency (for instance, mutual recursion) is one such case. Another typical scenario is that of polymorphic macros (*cf.* figure 6 on page 320): you may need to use a polymorphic macro at a time when it hasn't been `\let` to its actual implementation yet. Those exceptions are unavoidable and are not to be feared. A short comment in the code can help the reader navigate through those detours.

4.2.2 Feature-oriented organization

In terms of code organization, the second principle to which we try to conform is arranging the code by feature instead of by implementation. This means that we have a tendency to think in terms of "what it does" rather than "how it does it" when we organize code sections in source files. In our case, this is a

relatively recent change of perspective which, again, comes from the idea of putting oneself in the "hacker" position. When people need to look at your code, they are most of the time interested in one particular feature that they want to imitate, extend, modify or adapt for whatever reason. In such a situation, acquiring an understanding of the feature's inner workings is easier when all the code related to that feature is localized at the same place in the source.

To illustrate this, we will intentionally take an example which may be controversial: the case of internationalization. The *CurVe* class has several features which need to be internationalized: rubrics need a "continued" string in case they extend across several pages, bibliographic sections need a "List of Publications" title, *etc.* In *CurVe* 1, the code is already organized by feature, except for multi-lingual strings which are all grouped at the end, like this:

```
%% Implement rubrics
%% ...

%% Implement bibliography
%% ...

\DeclareOption{english}{%
  \continuedname{continued}
  \listpubname{List of Publications}}
\DeclareOption{french}{%
  \continuedname{suite}
  \listpubname{Liste des Publications}}
%% ...
```

These days, we find this unsatisfactory because the code for each feature is scattered in several places. For instance, the `\continuedname` macro really belongs to the rubrics section and hence should not appear at the end of the file. This kind of organization is indeed implementation-oriented instead of feature-oriented: we grouped all multi-lingual strings at the end because in terms of implementation, the idea is to define a bunch of `\<whatever>name` macros.

In *CurVe* 2, we will take a different approach, as illustrated below:

```
%% Implement rubrics
\newcommand*\continuedenglishname{%
  continued}
\newcommand*\continuedfrenchname{%
  suite}
%% ...

%% Implement bibliography
\newcommand*\listpubenglishname{%
  List of Publications}
\newcommand*\listpubfrenchname{%
  Liste des Publications}
%% ...

\DeclareOption{english}{%
  \def\@currlang{english}}
\DeclareOption{french}{%
  \def\@currlang{french}}
%% ...
```

After that, using the appropriate "continued" string is a matter of calling

```
\csname continued\@currlang name\endcsname
```

This new form of code organization has several advantages. First, *all* the code related to one specific feature is now localized in a single place. Next, it conforms better to the bottom-up approach (no forward reference to a multi-lingual string macro is needed). Finally, and perhaps unintentionally, we have improved the flexibility of our package: by implementing a macro such as `\@currlang`, we can provide the user with a means to dynamically change the current language right in the middle of a document, something that was not possible before (language processing was done when the package was loaded).

Earlier, we said that this example was taken intentionally because of its controversial nature. Indeed, one could object here that if someone wants to modify the multi-lingual strings, or say, support a new language, the first version is better because all internationalization macros are localized in a single place. It is true that if you consider internationalization as a *feature*, then our very own principle would dictate to use the first version. This is simply a demonstration that in general, there is no single classification scheme that can work for all purposes. However, we think that this argument is not really pertinent. If you would indeed want to modify all the multi-lingual strings, you would open the source file in Emacs and use the `occur` library to get all lines matching the regular expression

```
^\\newcommand\*\\.+name{
```

From the occurrence buffer, you can then reach every relevant line directly by hitting the `Return` key. This is really not complicated and in fact, could be more good programming advice: *know your tools*.

4.3 Rule #3: Use `filehook` for intercession

The final behavioral rule we would like to propose in this section deals more specifically with the intercession problem. We recently came up with a design pattern that we think helps smooth the implementation of inter-package compatibility.

4.3.1 Standard tools

The first thing we need to realize is that in general, the standard LaTeX tools are too limited.

`\@ifpackageloaded` allows you to detect when a package has been used or required, and possibly take counter-measures. However, this is only a *curative* way of doing things: it only lets you provide post-loading (*a posteriori*) code. What if you need to take precautionary measures instead?

`\AtBeginDocument` allows one to massively defer code execution until the beginning of a document, that is, after every package has been loaded. This is obviously a very gross granularity. For example, it doesn't provide any information on the order in which the packages have been loaded, something that might be needed even for post-preamble code.

Consider for example the following scenario:

- Style S calls `\AtBeginDocument{\things}`
- Class C loads style S

And ask yourself the following question: how does class C intercede on `\things`? There is no simple way to sort this out with the standard LaTeX tools.

4.3.2 `filehook`

Like probably almost every package developer, we have fought against these problems for years with intricate and obfuscated logic to fix inter-package compatibility. We think however that the very recent appearance of Martin Scharrer's `filehook` package is (should be) a crucial component in cleaning up the current intercession mess.

The `filehook` package provides pre- and post-loading hooks to files that you input in every possible way (`\include`'d files, packages, even class files). Thanks to that, one can now handle intercession in a context-free way, which is much better than what was possible before. For example, you can take both *a priori* and *a posteriori* counter-measures against any package, without even knowing for sure if the package is going to be loaded at all. This notably includes the possibility of saving and restoring functionality, much like what OpenGL does with its `PushMatrix` / `PopMatrix` or `PushAttrib` / `PopAttrib` functions (although OpenGL uses real stacks for this).

Eventually, the existence of `filehook` allowed us to come up with a particular design pattern for intercession management that can be summarized as follows. So far, this pattern works (for us) surprisingly well.

- First of all, start by writing your code as if there were no intercession problem. In other words, simply implement the default behavior as usual, assuming that no other package would be loaded.
- Next, handle compatibility problems with packages, one at a time, and only locally: use pre- and post-hooks *exclusively* to do so.
- Remember that hook code is only *potential*: none of it will be executed if the corresponding package is not loaded.
- Also, note that getting information on package loading order is now trivial if you use `\@ifpackageloaded` in a pre-hook.
- Avoid using `\AtBeginDocument` for intercession, unless absolutely necessary; for instance, if you need to take a counter-measure against a package that already uses it. Again, in such a case, calling `\AtBeginDocument` in a post-hook will allow you to plug in the relevant code at exactly the right position in the `\@begindocumenthook` chain.

4.3.3 Bibliography management in *CurVe*

To provide a concrete example of these ideas, let us demonstrate how recent versions of *CurVe* handle compatibility with various bibliography-related packages. A summarized version is given in figure 8 on the following page. Roughly speaking, what this code does is:

- install the default, *CurVe*-specific behavior first,
- step back if `bibentry` is loaded,
- merge with `multibib`,
- step back before `splitbib` and re-merge afterwards,
- render `hyperref` inoperative.

Knowing where we came from, that is, how this logic was done before `filehook`, it is amazing how readable, clear, concise, and in fact simple, this new implementation is. We cannot be sure how striking this will be for the unacquainted reader, but in case it is not, you are invited, as an exercise, to try an implement this only with the standard LaTeX macros (hint: it is practically impossible).

Earlier, we claimed that `filehook` would allow us to program in a context-free way. Let us explain now what we meant by that. First of all, note that because we use hooks exclusively to plug our intercession code, the five special cases could have been implemented in *any* order in the *CurVe* source file. We can move the five blocks around without modifying the semantics of the program. This is what we mean by being "context-free": the current dynamic state of the program has no effect on the code we put in hooks. Another instance of context freedom is in the specific case of `hyperref`. The important thing to notice here is that we save (and restore) *whatever* state we had just before loading `hyperref`.

```
%% Step 1: implement the default bibliographic behavior
%% ...

%% Backup LaTeX's original macros and replace them by our own:
\let\@curveltx@lbibitem\@lbibitem
\def\@curve@lbibitem[#1]#2{...}
\let\@lbibitem\@curve@lbibitem
%% ... do the same for \@bibitem, \bibitem etc.

%% Step 2: special cases

%% Bibentry. Restore standard definitions because bibentry just inlines
%% bibliographic contents.
\AtBeginOfPackageFile{bibentry}{
  \let\@lbibitem\@curveltx@lbibitem
  ...}

%% Multibbl. Merge its definition of \bibliography with ours.
\AtEndOfPackageFile{multibbl}{
  \def\bibliography##1##2##3{...}}

%% Splitbib.
%% Before: restore standard definitions because ours are only used as part of
%% the \endthebibliography redefinition.
\AtBeginOfPackageFile{splitbib}{
  \let\@lbibitem\@curveltx@lbibitem
  ...}
%% After: Modify \NMSB@writeentry and re-modify \endthebibliography back.
\AtEndOfPackageFile{splitbib}{
  \def\NMSB@writeentry##1##2##3##4##5,{...}%
  \def\endthebibliography{...}}

%% Hyperref. Currently, we don't want hyperref to modify our bibliographic
%% code, so we save and restore whatever bibliographic state we had before
%% hyperref was loaded.
\AtBeginOfPackageFile{hyperref}{
  \let\@curveprevious@lbibitem\@lbibitem
  ...}
\AtEndOfPackageFile{hyperref}{
  \let\@lbibitem\@curveprevious@lbibitem
  ...}
```

Figure 8: Intercession management

In other words, here again we don't need to know our exact context (the specific definition for the macros involved). We just need to save and restore it. And again, it is impossible to do that without the ability to hook code before and after package loading — which `filehook` now provides.

5 Level 4: Social

This section, shorter than the others, addresses a final level (or rather, a meta-level) in coding standards: the social level. Here, we are interested in how the human behavior may affect the LaTeX world and in particular the development of packages. We only provide two simple rules, and a rather unfortunate, but quite illustrative story.

We mentioned in the introduction the anti-social development syndrome that LaTeX seems to suffer from. In our opinion, this behavior is what leads to wheel-reinvention (*cf.* section 3.1 on page 318) and hence redundancy (for instance, the existence of half a dozen packages for key-value interfaces). In an ideal world, the situation could be improved by following the two simple rules described below.

5.1 Rule #1: Be proactive

The first rule of thumb is to permanently try to *trigger* collaboration. Package development frequently comes from the fact that you are missing a particular functionality. However, there is little chance that you are the first person to miss the functionality in question. Therefore, the first thing to do is to look for an existing solution instead of starting your own. By the way, we *know* that it is fun to start one's own solution. We have done that before, but it is nothing to be proud of!

Once you find an already existing solution (and you will, most of the time), it will probably not be an exact match. You will feel the need for implementing a variant or an extension of some kind. Here again, don't take this as an excuse to start your own work, and don't keep your work for yourself either. Try to be proactive and trigger collaboration: contact the original author and see if your ideas or your code could be merged in some way with the upstream branch. This is the first key to avoid redundancy.

5.2 Rule #2: Be reactive

Of course, this can only work if there is a response from the other side. And this is the second rule of thumb: if collaboration is proposed, *accept* it. Maintaining a package should be regarded as a certain responsibility towards its users. People frequently escape from their maintenance responsibility by hiding behind the free software banner (free as in freedom and/or as in beer). This is of course legitimate but also abused to the point of leading to the anti-social syndrome we have been discussing.

Being reactive means reviewing and accepting patches from other people in a reasonable time frame (for some definition of "reasonable"). It also means listening to other people's suggestions and implementing them within the same reasonable time frame. We understand that this is not always possible, but when you feel that there is some kind of pressure on you, there is also an alternative: *trust* people and *open* the development. Use a version control system (VC) of some kind. Put your code on `github` or a similar place and let people hack on it. The advantage to using a VC is that it is always possible to revert to an earlier state in the history of the package.

5.2.1 *FiNK* and `currfile`

We realize these considerations may be somewhat idealistic. In order to illustrate why they are important anyways let us tell a short story.

Sometime in 2010, we were contacted by Martin Scharrer, the author of `filehook`, about another package of his named `currfile`. This package maintains the name of the file currently being processed by LaTeX. Martin was inquiring about a potential cross-compatibility with ***FiNK***, one of our own packages, which does exactly the same thing.

We answered politely with the requested information and continued the email conversation for a little while, not without a wee bit of frustration however. Why yet another package for this? Wasn't ***FiNK*** good enough? Couldn't its functionality have been extended rather than duplicated?

Interestingly enough, we were recently sorting out some old mail when we dug up a message from this very same Martin Scharrer, providing a patch against ***FiNK*** in order to ground it onto `filehook`. This message was one year and thirty eight weeks old. Of course, we had completely forgotten all about it. In terms of time frame, one year and thirty eight weeks is way beyond "reasonable". So much for being reactive, lesson learned, the hard way...

6 Conclusion

In this paper, we addressed the notion of LaTeX coding standards. We started by analyzing the reasons why no such thing seems to exist as of yet. In short, the lack of coding standards for LaTeX can be justified by a mostly anti-social development syndrome, a not so pressing need for them in the view of the developers and the lack of help and support from the usual text editors. We however demonstrated that having a set of coding standards would be extremely beneficial to the community. First, they would help make the most of a programming language that is far less structured than the more casual ones. Next, they would also help in terms of code homogeneity and readability, both key components in collaboration. This is especially important in LaTeX because even if *intentional* collaboration is not so widespread, there *is* a very frequent form of *forced* collaboration, which is intercession (inter-package compatibility and conflict management).

We then reported on our own development experience and proposed a set of rules and design patterns that have helped improve our own code over the years. Those rules were organized in four different abstraction levels: layout (formatting and naming policies), design (modularity and other programming paradigms), behavior (interfaces and intercession management) and finally the meta-level (social behavior).

We don't expect that everyone would agree to every one of these rules, as we know that a coding style is above all a matter of personal taste. In fact, a coding style is important, but it is even more important to stick to it, that is, to stay coherent with yourself. Developing a coding style is also a matter of

keeping the problem in mind permanently, not unlike a daemonized process running in the background of one's head. Every time you write a line of code, you need to ask yourself, "is this the proper way to do it?" This also means that a coding style may be a moving target, at least partially. It will evolve along with the quality of your code. Finally, one should remember that there can be no rule without exceptions. Knowing when to escape from your style for the greater good is as important as conforming to it.

The ideas, rules and design patterns proposed in this article are those that work best for us, but our hope is that they will also help you too. Many other ideas have not been tackled in this paper, both at the level of the document author and at the level of the package developer. Much more could be said on the matter, and if there is enough interest in the community, maybe it is time for an "Elements of LaTeX Programming Style" book which remains to be written. Perhaps this article could serve as a basis for such a book, and we would definitely be willing to work on such a project.

References

[1] AUC-TeX. http://www.gnu.org/s/auctex.

[2] The comprehensive TeX archive network. http://www.ctan.org.

[3] The XEmacs text editor. http://www.xemacs.org.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture*, volume 1. Wiley, 1996.

[5] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*, volume 4. Wiley, 2007.

[6] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*, volume 5. Wiley, 2007.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[8] B.W. Kernighan and P.J. Plauger. *The Elements of Programming Style.* McGraw-Hill, 1974.

[9] Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture: Patterns for Resource Management*, volume 3. Wiley, 2004.

[10] Patty Maes. Concepts and experiments in computational reflection. In *OOPSLA*. ACM, December 1987.

[11] Frank Mittelbach. Reflections on the history of the LaTeX Project Public License (LPPL) — A software license for LaTeX and more. *TUGboat*, 32(1):83–94, 2011. http://tug.org/TUGboat/tb32-1/tb100mitt.pdf.

[12] Frank Mittelbach, Michel Goossens, Johannes Braams, David Carlisle, and Chris Rowley. *The LaTeX Companion,* second edition. Addison Wesley, 2004.

[13] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. Wiley, 2000.

[14] Brian C. Smith. Reflection and semantics in Lisp. In *Symposium on Principles of Programming Languages*, pages 23–35. ACM, 1984.

[15] Richard M. Stallman. The GNU coding standards. http://www.gnu.org/prep/standards.

[16] William Strunk Jr. and E.B. White. *The Elements of Style.* W.P. Humphrey, 1918.

[17] Didier Verna. The CurVe class. http://www.lrde.epita.fr/~didier/software/latex.php#curve.

[18] Didier Verna. The FiXme style. http://www.lrde.epita.fr/~didier/software/latex.php#fixme.

[19] Didier Verna. Classes, styles, conflicts: The biological realm of LaTeX. *TUGboat*, 31(2):162–172, 2010. http://tug.org/TUGboat/tb31-2/tb98verna.pdf.

[20] Joseph Wright and Christian Feuersänger. Implementing key–value input: An introduction. *TUGboat*, 30(1):110–122, 2009. http://tug.org/TUGboat/tb30-1/tb94wright-keyval.pdf.

◇ Didier Verna
EPITA / LRDE
14-16 rue Voltaire
94276 Le Kremlin-Bicêtre Cedex
France
didier (at) lrde dot epita dot fr
http://www.lrde.epita.fr/~didier