Object-Oriented Programming, Descriptive Markup, and TEX

Arthur Ogawa
TEX Consultants, P.O. Box 51, Kaweah, CA 93237-0051, U.S.A.
ogawa@orion.arc.nasa.gov

Abstract

I describe a synthesis within TEX of descriptive markup and object-oriented programming. An underlying formatting system may use a number of different collections of user-level markup, such as LATEX or SGML. I give an extension of LATEX's markup scheme that more effectively addresses the needs of a production environment. The implementation of such a system benefits from the use of the model of object-oriented programming. LATEX environments can be thought of as objects, and several environments may share functionality donated by a common, more general object.

This article is a companion to William Baxter's "An Object-Oriented Programming System in TEX."

I believe that the key to cost-effective production of TEX documents in a commercial setting is *descriptive markup*. That is, the document being processed contains *content* organized by *codes*, the latter describing the structure of the document, but not directly mandating the format.

The formatting of such a document is embodied in a separate module (usually a file of definitions of formatting procedures) which represents the implementation of a typographic specification (*typespec*). Thus, descriptive markup achieves the separation of document instance from formatting engine.

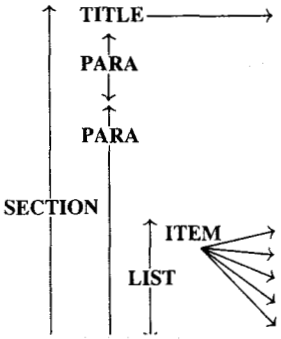
At the same time, the key to cost-effective generation of document formatters in TEX is found in the paradigms of object-oriented programming (OOP). Typographic elements are treated as *objects*, with data and methods. The formatter is a collection of code modules with well-defined boundaries and communication pathways. The programmer can take advantage of OOP techniques such as object encapsulation, data-hiding, and inheritance to create robust, easy-to-maintain, powerful formatters.

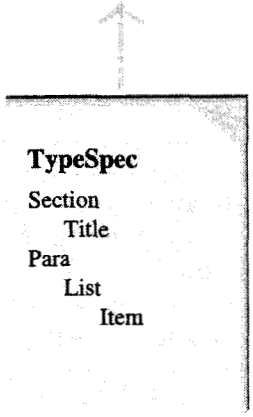
For the purposes of this article, LATEX and SGML will be used as specific instances of descriptive coding schemes, but other methods that cleave to the standards of descriptive markup are not excluded. In particular, databases are very descriptive in nature, and the processing engine described in this and the next paper will process such data well.

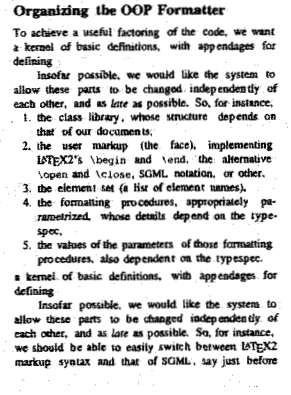
The present article discusses issues of descriptive markup and object-oriented programming as relate to TEX and document processing. The next article gives implementation details of the processing engine.

Commercial Typesetting with LATEX

Advantages of LATEX's Descriptive Markup. The descriptive markup of LATEX bestows numerous advantages on this document processing system, making it the predominant TEX macro package.






Organizing the OOP Formatter

To achieve a useful factoring of the code, we want a kernel of basic definations, with appendages for defining

Insofar possible, we would like the system to allow these parts to be changed independently of each other, and as *late* as possible. So, for instance,

1. the class library, whose structure depends on that of our documents,
2. the user markup (the face), implementing LATEX2's `\begin` and `\end`, the alternative `\open` and `\close`, SGML notation, or other,
3. the element set (a list of element names),
4. the formatting procedures, appropriately parametrized, whose details depend on the typespec,
5. the values of the parameters of those formatting procedures, also dependent on the typespec.

a kernel of basic definitions, with appendages for defining

Insofar possible, we would like the system to allow these parts to be changed independently of each other, and as *late* as possible. So, for instance, we should be able to easily switch between LATEX2 markup syntax and that of SGML, say just before

Simple Syntax. LaTeX's environments and commands provide a simple system of user-level markup; there are only the *environments* (with content) and the *commands* (with argument).

Completeness. LaTeX's public styles are of sufficient richness to accommodate many of the structures required for a typical book. Modest extensions enable one to code fairly technical books.

Context-sensitive formatting. An enumerated list may contain yet another list: the latter is formatted differently than when it appears at the topmost level. The same environment can be used in numerous contexts, so there are fewer markup codes for the author or typesetter to remember.

Authoring versus formatting. Even though using the same set of markup codes as the author, the typesetter may employ a different set of formatting procedures, allowing the author to concentrate on content and structure while leaving the typesetter to deal with the thorny production problems (e.g., float placement, line- and pagebreaks).

Limitations of LaTeX's Markup. Despite the aforementioned advantages, LaTeX has a number of problems.

Inconsistencies. Some of LaTeX's codes introduce syntax beyond the environment and command mentioned above, e.g., the `\verb` command.

Architecture. LaTeX's *moving arguments* and *fragile commands* consititute annoying pitfalls. That the `\verb` command must not appear within the argument of another command has bitten numerous unwary users.

Debasement with procedural markup. When an author inevitably conceives of new markup elements, he or she will commonly be disinclined to simply define new environments to go with them. Instead the author is likely to introduce them in the document instance itself with explicit formatting codes.

The awkward optional argument. Even though many LaTeX commands and environments have a variant (*-form) or an optional argument (within brackets `[]`), not all do, and those that do not are unable to parse a * or optional argument if one does appear in the document. This increases LaTeX's syntactic complexity. Furthermore, the existing scheme is inadequate to accommodate much demand for options, because any one command may have at most one *-form and one optional argument.

User-interface software. Using TeX to code a document is problematic because only TeX can validate the document — and TeX does not perform well as a document validator (nor was it intended for such a use).

Software exists to help generate a valid LaTeX document; the emacs LaTeX mode and TCI's Scientific Word are two such. But neither can assert (as an SGML validator can) that the document has no markup errors.

Limitations of LaTeX Styles. Separating core processing functionality from design-specific formatting procedures is embodied in LaTeX's style (`.sty`) files. It is a useful idea, allowing the considerable investment in LaTeX's kernel to be amortized over a large body of documents, but it has limitations.

Excessive skill requirements for style writers. Because LaTeX exposes TeX's programming language within the style files, only someone skilled in programming TeX can create the style file for a new document typespec. Less daunting is the task of customizing an existing style, but this remains out of the reach of professional designers as a class. This situation stands in sharp contrast to commercial applications such as Frame Maker, which possess what I call a *designer interface.*

Designer-interface software. Some progress has been made to supply software that will generate the code of a LaTeX style, notably TCI's Scientific Word. One can think of a fill-in-the-blank approach that allows one to specify the values of dimensions that parametrize a typespec. But there is currently no method of extending an existing body of styles to accommodate new formatting procedures and parametrizations.

Incomplete Implementation. Much work remains to be done in separating style-specific code from kernel code: LaTeX2's core definitions as they now stand make numerous decisions about document structure and formatting metrics.

Commercial Typesetting with SGML

Because a Standard Generalized Markup Language (SGML) parser can verify the validity of the markup of a document, and because SGML markup is purely descriptive (to first order), it supplies an effective "front-end" to a TeX-based formatter. A number of commercial systems have implemented this idea. At the same time, SGML is not prey to LaTeX's limitations.

Documents in Classes. In an SGML system, a document instance belongs to a class defined by a Document Type Definition (DTD), which specifies the concretes of the markup scheme, the name of each *element*, or tag (in LaTeX: environment), its *attributes* (modifiers) and their allowable values, and the element's *content model.* The latter specifies

what elements may or must appear within a given element, and what order they must appear in. For example,

```
<!ELEMENT theorem - -
 (title, paragraph*)
 >
<!ATTLIST theorem
 id ID #REQUIRED
 kind (theorem|lemma|corollary) #IMPLIED
 >
```

defines the "theorem" element and specifies that it has to be given a key called "id" (like LaTeX's `\label` command) and may carry an attribute, "kind", whose value, if specified, must be either "theorem", "lemma", or "corollary". Its content must have an element called "title", followed by any number of paragraphs. The DTD is thus the basis for SGML document validation.

Elements with attributes. SGML has just one syntax for its descriptive markup, namely the element. An element instance may specify the values of its attributes, or may accept a default; this allows the value to be determined effectively by the formatter, or by inheritance from some containing element (discussed in more detail below). A typical instance of an SGML element in a document might be:

```
<theorem ID="oops1" kind=Corollary>
 <title>OOPS, A Theorem</title>
 ⟨content of the theorem⟩
 </theorem>
```

Note that in SGML we really may not give the title as an attribute, because an SGML attribute can not, for instance, contain math. The practice is rather to put the text of the title in an element of its own.

General and consistent markup. The advantages of such a meager syntax cannot be overstated. An author may generate a relatively complex document with a fairly small set of markup. At the same time, SGML application software may assist in selecting and inserting the codes, thereby removing the onus of verbose markup.

The document as database. It is a common school of thought to treat an SGML document instance as rather a collection of structured data than a traditional book or article. This emphasizes the desirability of descriptive markup and the undesirability of procedural markup. Such a document can be published on numerous different media (paper, CD-ROM) and forms (demand publishing, custom publishing). The value of a document coded this way cannot be overstated.

Face-Independent Procedures

Separating Markup from Formatting Procedures. A core processor is something that will serve equally well as a formatter for SGML, LaTeX2, LaTeX3 and beyond. It must, in fact, be able to parse user markup defined by some external specification, what we call a *face*. At the same time, its style files must not at all determine the input syntax.

Here, I describe the span of user markup that must be parsed. Each one of these markup schemes constitutes a different face of the core processor.

Bestowing Attributes on LaTeX Environments. An extension to the LaTeX2 syntax which provides flexible SGML-like attributes is:

```
\begin{theorem
 \kind{Corollary}
 \number{2.1}
 \prime{}
 \title{OOPS, A Theorem}
 \label{oops1}
 }
 ⟨content of the theorem⟩
 \end{theorem}
```

This notation is such that current LaTeX2 markup simply coincides with default values for all attributes.

SGML Markup. I gave an example of an SGML element instance above. What corresponds to a LaTeX sectioning command might appear as:

```
<section ID="sgmlmarkup">
 <title>&SGML; Markup Syntax</title>
 <title-short>&SGML; Markup</title-short>
 <title-contents>&SGML; Markup</title-contents>
 ⟨content of the section⟩
 </section>
```

Here, the elements `<title-short>` and `<title-contents>` would be optional and would specify a short title for the running head and table of contents respectively. The syntax `&SGML;` is that of a text *entity*, an SGML shorthand. Interestingly enough, in a TEX-based processor for SGML markup, it suffices for the two characters < and & to have catcode active (13), with all others as letter (11) or other (12).

Markup for a Successor to LaTeX2. For LaTeX3 we propose the markup scheme:

```
\open\theorem{
 \number{2.1}
 \prime{}
 \label{oops1}
 }
 \open\title OOPS, A Theorem\close\title
 ⟨content of the section⟩
\close\theorem
```

The options appear in a brace-delimited argument,

while the command name is simply a token. This syntax replaces LaTeX2's environments and commands alike.

Note here that the implementation of the `\title` element could in principle parse its entire content into a TeX macro parameter using the tokens `\close\title` as a delimiter. The same observation also applies to SGML syntax (with `</title>` as the delimiter), but not to LaTeX2's syntax, where the end of the environment contains the brace characters. This observation was evidently not lost on the creators of $\mathcal{A}_{\mathcal{M}}\mathcal{S}$-TeX, who tend to close out their elements with a control sequence name, like `\endtitle`.

The Defining Word. A system that is able to encompass the above markup syntax may be readily extended to other syntax. More important, though, is that all commands defined by such a system share a single, consistent syntax. LaTeX2 would possess this attribute if all environments were defined by means of `\newenvironment`; anyone who has looked inside LaTeX's core macro file or its style files knows otherwise, though.

The `\newenvironment` command of LaTeX's style files is an instance of what we may call a *defining word*, to borrow a phrase from FORTH. We shall see later the relationship between defining words and the OOP concept of class creation.

Benefits in production. As the next talk will also emphasize, the mere existence of a convenient syntax for element attributes bears importantly on production needs. The need is so longstanding that the TeX Users Group-supplied macros for authoring papers submitted to this conference have a syntax for introducing multiple options, and LaTeX users from time immemorial have resorted to their own techniques, e.g.,

```
{\small
\begin{verbatim}
Your text
On these lines
\end{verbatim}
}
```

to reduce the typesize of an environment.

Object-Oriented Programming and TeX

In a rather happy conjunction of requirements and resources, we are now in a position to employ the 20-year old technology of Object-Oriented Programming (OOP) to advance the 16-year old TeX. Here, I introduce certain OOP concepts and show their relationship with the current work.

Object-Oriented Programming Basics.

Data and procedures are encapsulated into objects. To paraphrase a famous formula:

Fields + Methods = Object

That is, an object is a self-contained computing entity with its own data and procedures. For instance, we can have a object called "enumerated list", one of whose attributes tells whether it is an arabic, roman, or lettered list. Other instances of enumerated list have their own value for this attribute, determined by the context of the object, or specified in the instance.

The object is an instance of its class. A class abstracts an object. In the above example of enumerated list, all enumerated list objects are molded on the same form, the enumerated list class. When the formatter encounters an enumerated list within the document, it creates an instance of the class (say, object number 5):

$\exists$ list5 $\Leftarrow$ enumerated list

We can look upon a document as a collection of elements, each being an instance of the related class. The paragraph you are reading falls within a section within a section within a section of an article. Three section objects exist simultaneously, yet distinctly. Each of these sections has a title, as a section must. The title of a section is an attribute which is always defined upon its appearance within a document; there is no (non-trivial) default value determined by the class.

An object's fields are private. Encapsulation refers to the practice of disallowing other objects from directly altering a class's fields; instead, objects pass each other messages. An object may alter one of its own fields in response to another object's message. In a numbered list, for example, the counter is "owned" by the list itself, not by the list item; when the latter is instantiated, it sends a message to the list object to increment the counter.

A derived class inherits from its base class. In what is possibly the most powerful paradigm of OOP, a new class of objects can be created (derived) from an existing (base) class by the addition of new fields and methods. The new, or child, class inherits all the fields and methods of the generating, or parent, class. Some of the added methods may supersede, or *override* those that would otherwise be inherited from the parent.

For instance, we may create an enumerated list class from a basic list class by appending a field which determines whether the list device is an number or letter and by overriding the procedure that formats the list device so that it uses this field

appropriately. All other aspects of the list format are determined by the parent class:

$\forall$ enumerated list $\Leftarrow$ list $\supset$ {counter + device}

A derived class may inherit from more than one parent. In a system with *multiple inheritance*, a new class can be created that inherits simultaneously from two or more exising classes. This is sometimes referred to as *mix-in classes.*

For instance, we may have created a class that numbers its instances, applying this to, say, equations and theorems, but the enumerated list class mentioned above should also be a child of this numbering class. In fact, the enumerated list class inherits from both the list class and the numbering class.

$\forall$ enumerated list $\Leftarrow$ list + counting $\supset$ {device}

The structure of the interrelated classes, including descendents is called the *class hierarchy.*

The object has a context in its document. Since the abovementioned sections are nested, each section has a different hierarchical position within the document. This affects their respective formatting (intentionally so, in order to *reveal* the document's structure). This nesting of elements in the document instance is called the *document hierarchy.*

Note that class hierarchy is independent of any particular document instance, while document hierarchy is not *a priori* related to the class hierarchy. Thus, any two enumerated list objects within a document are instantiated identically (they are "created equal"), regardless of where they might appear. Likewise, within a document, a list item must always appear within a list, but in the class hierarchy discussed in the next paper, the item class is a subclass of a run-in head.

Environments, Elements, and Objects. There seems a fairly straightforward connection between LaTeX environments and SGML elements. But where do classes and objects fit in? We can think of a class as an abstract environment or element, and an object as a specific instance thereof within a document.

The distinction between class and object is important, because an instance of a class within a document is allowed to have instance options: these must not affect the fields' values in the class itself, which remain unaltered while the document is processed.

Advantages of the OOP approach. In other venues, OOP is said to have the advantages of good organization, robustness of code, reuse of code, ease of modification and extension, and ease of comprehension.

In descriptive markup, the OOP approach makes particular sense because of the close correspondence between element and class, and between element instance and object instance.

The modularity of objects implies a decoupling between them, allows the methods of one object to be maintained, changed, and extended without affecting other objects, and allows one to learn a particular class hierarchy by first understanding each of its elements separately, then in relation to each other.

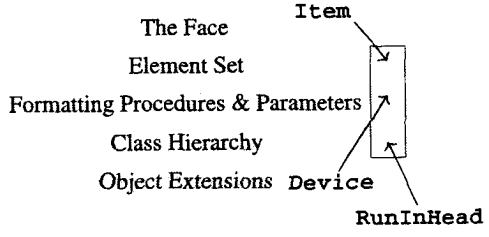
Organizing the OOP Formatter

To achieve a useful factoring of the code, we want a kernel of object extensions, with appendages defining

- the class library, whose structure depends on that of our documents,
- the formatting procedures, appropriately parametrized, whose details depend on the typespec,
- the values of the parameters of those formatting procedures, also determined by the typespec,
- the element set (a list of element names), each bound to a particular formatting procedure. In an SGML formatter, this could be derived automatically from the DTD or some other resource.
- the user markup (the face), implementing LaTeX2's `\begin` and `\end`, the alternative `\open` and `\close`, SGML notation, or other syntax.

The figure shows these modules in relationship to each other. The last aspect to be applied, the face, is seen to be truly a very small module placed on top of the entire stack.

Modularity and Late Binding. Insofar as possible, we would like these parts to be independent of each other, and *late* changes should be permitted. So, for instance, we should be able to switch easily between the LaTeX2 markup syntax and that of SGML, say just before the `\article` command starts the actual



document. Or, we would like to alter the name of an element; in principle, a `\chapter` command by any other name would still format a chapter opener. Equally well, we may wish to revise the details of a formatting procedure or the value of one of its parameters to reflect an alteration to the typespec. All of these changes are *incremental.* In fact, we shall be able to do all these things principally because TEX is an interpreter, not a compiler.

Maintaining the Formatter. There tends to be an additional relationship, an example of which is indicated, in which an element, a formatting procedure, and a class are connected. In this case, the abstract class `RunInHead` is subclassed to provide what will be known as the `Item` element. In the process, a procedure `Device` is donated, which takes care of the formatting of the list device.

This vertical connection is natural and, to the programmer, compelling. But when developing a document formatter, the distinctions between class, formatting procedure, and element name must nonetheless be preserved for ease of maintenance.

Extensive Use of Defining Words. In order to achieve the greatest of uniformity in the code, we will use defining words exclusively to create the class hierarchy, and to bind the user-level markup codes to their respective procedures. When a new class is derived from another, a defining word is invoked. A user-level code will invoke a different defining word to instantiate an object of a class.

Elsewhere, defining words are used to allocate counters and dimensions (as does LATEX's `\newcounter` or Plain TEX's `\newdimen`), as well as other, more complex constructs.

Bibliography

Baxter, William E. "An Object-Oriented Programming System in TEX." These proceedings.

Lamport, Leslie. *LATEX—A Document Preparation System—User's Guide and Reference Manual.* Reading, Mass.: Addison-Wesley, 1985.

Goldfarb, Charles F. *The SGML Handbook.* Clarendon Press, 1990.

Wang, Paul S. *C++ with Object Oriented Programming.* Boston: PWS Publishing, 1994.