An Object-Oriented Programming System in \TeX

William Baxter
SuperScript, Box 20669, Oakland, CA 94620-0669, USA
web@superscript.com

Abstract

This paper describes the implementation of an object-oriented programming system in \TeX. The system separates formatting procedures from the document markup. It offers design programmers the benefits of object-oriented programming techniques.

The inspiration for these macros comes from extensive book-production experience with \TeX. This paper is a companion to Arthur Ogawa's "Object-Oriented Programming, Descriptive Markup, and \TeX".

The macros presented here constitute the fruit of a struggle to produce sophisticated books in a commercial environment. They run under either plain \TeX or \LaTeX, but owe their primary inspiration to \LaTeX, especially in the separation of logical and visual design. The author hopes that future \TeX-based document production systems such as \LaTeX3 and NTS will incorporate these techniques and the experience they represent.

Throughout this paper we refer to book production with \LaTeX. Many of the comments apply equally well to other \TeX-based document processing environments.

Design and Production Perspectives

Certain problems routinely crop up during book production with \LaTeX. The majority fall into two general categories: those related to the peculiarities of a particular job and those regarding the basic capabilities of the production system.

Peculiar documents. Strange, and sometimes even bizarre, element variants often occur within a single document. Without extremely thorough manuscript analysis these surprise everybody during composition, after the schedules have been set. The author received the following queries during production of a single book:

1. What is the proper way to set Theorem 2.1' after Theorem 2.1?
2. Small icons indicating the field of application accompany certain exercise items. How do we accommodate these variations?
3. This book contains step lists numbered Step 1, Step 2, ..., and other lists numbered Rhubarb 1, Rhubarb 2, ... How do we code these?

Each variation requires the ability to override the default behavior of the element in question, or to create a new element. This is not difficult to accomplish ad hoc. The design programmer can implement prefix commands modifying the default behavior of a subsequent command or environment, add additional optional arguments, or create new commands and environments. But these solutions demand immediate intervention by the design programmer and also require that the user learn how to handle the special cases.

A markup scheme in which optional attributes accompany elements provides a simple, consistent, and extensible mechanism to handle this type of production difficulty. Instead of the standard \LaTeX environment markup
\begin{theorem}[OOPS, A Theorem]
...
\end{theorem}
we write
\open\theorem{
\title{OOPS, A Theorem}}
\close\theorem

Each attribute consists of a key-value pair, where the key is a single control sequence and the value is a group of tokens. The pair resemble a token register assignment or a simple definition.

The \open macro parses the attributes and makes them available to the procedures that actually typeset the element. Thus any element instantiated with \open...\close allows attributes. Furthermore, any such element may ignore (or simply complain about) attributes it doesn't understand. For example, if an exercise item coded as an element requires both application and difficulty attributes, it can be coded like this:
\open\item{
\application{diving}}
The same formatter can be used with a different face, programmer can modify the procedures that actually typeset the element to make use of the new attribute. In the case of the rhubarb list, we can use an attribute of the list element to modify the name of the items:

\open\steplist{
  \name{Rhubarb}
}

...  
\close\steplist

Basic capabilities. Complex designs require macro packages far more capable than those of standard \LaTeX\ or \LaTeX\X. The designs require color separation, large numbers of typefaces, letter-spacing, complicated page layouts, backdrop screens and changebars, interaction of neighboring elements, and other "interesting" aspects of design. A production system must address all of these aspects of design in order to remain viable in a commercial setting. And it must do so in a cost effective manner.

In their desire to reduce the total and initial costs of a formatting package, production houses ask:

1. How do we use these macros with another manuscript that employs different markup, say, SGML?

2. When will we be able to write our own macro packages?

   The first question has a relatively simple answer. We define a generic markup scheme (very similar to \open...\close). Design programmers implement their formatting procedures assuming that all documents use this markup. A separate layer of parsing macros translates the markup that actually appears in the document into the generic markup. We call the generic markup scheme the OOPS markup. A design implemented behind the OOPS markup is a formatter. The markup scheme that actually appears in the document is the document markup. A set of macros that translate from a particular document markup into OOPS markup is a face.

   The OOPS markup constitutes an interface between formatter and face, while the face bridges the gap between document markup and OOPS markup. The same formatter can be used with a different face, and the same face with a different formatter. This newfound ability to reuse a formatter code makes \TeX\ or \LaTeX\X requires too much skill for most production houses to maintain. The macros described in the remainder of this paper address this deficiency through the application of object-oriented programming techniques to the problem of design implementation.

The OOPS approach

Before delving too far into the actual workings of the system we deliver the propaganda.

   The object-oriented programming paradigm fits the needs of document production extremely well. A document element is an object, and its type is a class. Thus a theorem element is an object of class \texttt{theorem}. Deriving one element type from another and overriding some behavior of the new element is a subclassing operation. For example, a lettered list class may be derived from a numbered list class. Attributes correspond to instance variables. The use of the \texttt{title} attribute in the theorem example above demonstrates this. The \LaTeX\X notion of a document style resembles a class library.

   After defining the OOPS markup, the remainder of this section describes a generic class library. A design programmer implements a particular document design using this standard set of element classes, possibly adding new classes as needed. The reader should consider a class library as an alternative to a \LaTeX\X's document style or document class.

OOPS markup. The OOPS markup for this system works somewhat like the \open and \close markup scheme presented above. The \texttt{@initialize} command creates an element of a particular class. It takes two arguments: the name of the class (or parent class) and the list of instance attributes. The command \texttt{@annihilate} destroys an element. It takes the element class to destroy as its single argument. So, reiterating the theorem example from above, the design programmer assumes the following style of markup:

\begin{verbatim}
@initialize\theorem{
  \title{OOPS, A Theorem}
  \number{2.1'}
}

...  
@annihilate\theorem
\end{verbatim}

This sample code instantiates an element of class \texttt{theorem}, overriding the \texttt{title} and \texttt{number} attributes. After some other processing it then destroys the \texttt{theorem} instance.

We pause here to provide some clues to the reader about how \TeX\ sees commands, elements, classes, and attributes. A \texttt{command} is a control sequence destined for execution by \TeX. In contrast, a \texttt{class}, \texttt{element}, or \texttt{attribute} is a string conveniently represented as a control sequence name. The OOPS
system neither defines nor executes these control sequence names. Instead, it derives a command name from an \element\attribute or class\attribute pair. One can think of attribute as a shorthand for command derived from the class\attribute or element\attribute pair. Thus the phrase execute an attribute means execute the command derived from an attribute:element or attribute: class pair.

Returning to the example, the \@ instantiate command creates an element of class theorem. It copies each attribute of the theorem class for the exclusive use of this particular element and overrides the meaning of the title and number attributes. After instantiation the hidden control sequence corresponding to the element: title attribute expands to "OOPS, A Theorem".

Adopting the OOPS markup as the interface between formatting procedures and document parsing routines is not terribly significant. But combining it with object-oriented programming techniques results in a powerful and flexible system for creating new element classes. Inheritance plays the key role. If one element class functions with the standard markup then new elements derived from it inherit this ability.

Subclassing. The \@element class is the common ancestor of all classes giving rise to document elements. This class supports the OOPS markup described above. It also supports subclassing operations, allowing the design programmer to derive from it new element classes that support OOPS markup.

The \@class command derives one element class from another. In the following example we derive the rhubarb list from the steplist.

\begin{verbatim}
@class rhubarbl ist stepl ist{
  \name{Rhubarb}
}
\end{verbatim}

The arguments to \@class are the new class name (or child class), the name of the parent class, and the list of attributes that override or supplement those of the parent. The \@class command parses its arguments, stores them in standard places, and then executes the \@class attribute from the parent class. The pseudo-code definition of the attribute \@class in the \@element class is:

\begin{verbatim}
@element:\class=
  <execute parent:\preclass>
  <create new class>
  <execute child:\subclass>
\end{verbatim}

The \@new command carries out the low-level processing for subclassing. It takes the new class name, parent class name, and attribute list parsed by \@class out of storage and constructs the new class from them. Every \@class attribute must execute \@new at some point, but can carry out other processing before and after executing \@new. The \@preclass and \@subclass attributes are subclassing hooks used, for example, to set up a default numbering scheme or allocate a class counter.

In the example above, \@class first clones the steplist class as rhubarblist. If the name attribute exists in the rhubarblist class then it is altered, otherwise it is added.

The \@class and \@new commands actually allow multiple inheritance. The parent argument to \@class may consist of one or more class names. The \@class attribute is always executed from the head parent, the first parent in the list. With multiple inheritance two parents may contain conflicting definitions of the same attribute. Attributes are passed from parent to child on a first-come-first-served basis. The child inherits the meaning of an attribute from the first parent class containing that attribute.

The definition of the \@class attribute in most classes is identical to that of \@element because it is inherited without overriding. But this system permits overriding the \@class attribute just like any other.

The design programmer uses \@class to create a class for each element. During document processing these classes are instantiated as document elements.

Instantiation. To instantiate an element the command \@ instantiate parses the OOPS markup and squirrels away its arguments in special locations. It then executes the \@ instantiate attribute from the class (from theorem in the above example). A pseudo-code definition of the \@ instantiate attribute in the \@element class is:

\begin{verbatim}
@element:\ instantiate=
  <execute parent:\pre instantiate>
  <create new element instance>
  <execute child:\ initialize>
  <execute child:\ startgroup>
  <execute child:\ start>
\end{verbatim}

The \@ pre instantiate attribute is analogous to the \@ pre class attribute. It carries out processing that must precede the creation of the new element. The \@ initialize attribute performs processing required by the newly-created element. The \@ start group attribute determines whether the element is subject to \TeX{}'s grouping mechanism. It usually expands to \verb|\begingroup|, but may also be left empty, resulting in an ungrouped element (like the \verb|\text| document environment). The \@ start attribute performs start processing for the particular element, in rough correspondence to the second required argument to the \TeX{} command \verb|\newenvironment|.

As with subclassing, the \@new command performs the low-level instantiation function. It constructs the new element from the material stored...
away by \@instantiate. This brings to light one basic implementation decision: a class is simply an object created with \@class, while an element is an object instantiated with \@instantiate.

The \@annihi1ate command parses class name and stores it in a standard location. It then executes the \@annihi1ate attribute from the most recent instantiation of that element. A pseudo-code definition of \@annihi1ate in the \@element class is:
\begin{verbatim}
@element:annihi1ate =
  <execute element:@end>
  <execute element:@endgroup>
  <destroy element instance>
\end{verbatim}

The \@end attribute corresponds to the third required argument to the \LaTeX \texttt{\newenvironment} command and carries out end processing. The \@endgroup attribute matches \@startgroup, and usually expands to \endgroup. Like \@startgroup it can also be given an empty expansion to eliminate grouping.

The \@free command provides the low-level mechanism for destroying an element instance, complementing \@new. It operates on the element name parsed by \@annihi1ate, cleaning up the remains of the now-defunct object. At some point \@annihi1ate must execute \@free, but other processing may precede or follow such execution.

In conclusion, the \@element class contains the following attributes: \@class, \@preclass, \@subclass, \@instantiate, \@preinstantiate, \@initialize, \@startgroup, \@start, \@annihi1ate, \@endgroup, and \@end. The \@class command derives new element classes from old. The \@instantiate command creates a new instance of a class as a document element, while \@annihi1ate destroys an element. Both \@class and \@instantiate share an underlying mechanism.

The class library. What good is the ability to derive one element class from another? It forms the basis for a class library that can replace a \LaTeX environment.

This class library is founded on the \@element class described above. All classes are ultimately derived from \@element using \@class to add new attributes as needed. We describe block elements, paragraphs, counting elements, listing elements, section elements, and independent elements.

A block element contributes to the vertical construction of the page. Examples include sections, theorems, tables, figures, display equations, and paragraphs. We interpret the common features of block elements as attributes in the class \@block.

Vertical space separates each block from its surroundings. During book production, final page makeup inevitably requires manual adjustment of the space around certain blocks. Strictly speaking, this violates the principle of purely descriptive markup, but the need is inescapable. The difficulty of properly adjusting the vertical space around \LaTeX environments routinely provokes severe consternation in production managers who care about quality.

The \@block class unifies the various mechanisms for inserting space above and below any block element with four attributes: \above, \below, \abovebelow, and \belowbelow. The first two are for the design programmer, and constitute the default space above and below the element. The latter two are deviations added to the first two in the obvious fashion. They are for the convenience of the final page makeup artist.

Each block element class uses these attributes in the appropriate manner. An ordinary block element, such as a paragraph, may insert space above and below using \addvspace. But a display math element would instead use \abovedisplayskip and \belowdisplayskip. In either case the user is always presented with the same mechanism for adjusting this space: the attributes \above, \below, \abovebelow, and \belowbelow. Furthermore, a class library can include the code to handle the most common block elements. Therefore the design programmer can work exclusively with the \above and \below attributes in the majority of cases.

A block element may also insert penalties into a vertical list above and below its content. It carries attributes \abovepenalty, \belowpenalty, \abovebelowpenalty, and \belowbelowpenalty that serve as penalty analogues to the above and below space attributes.

Block elements often require different margins than their surroundings. For example, a design may call for theorems, lemmas, and proofs to indent at each side. Furthermore, the justification may change from block to block. Block elements carry the attributes \leftindent, \rightindent, \leftjust, and \rightjust. The left and right indentation settings measure the relative offset from the prevailing margins, whereas the justification is an absolute setting. Again, each block element makes appropriate use of these attributes. Typically these attributes contribute to the values of \leftskip and \rightskip, with the fixed portion of the glue coming from the "indent" attribute and the stretch and shrink coming from the "justify" attribute. In this way they effectively decompose these \leftskip and \rightskip, demonstrating that we are not tied directly to the model that \TeX provides.

Paragraphs are block elements. In the author’s class library they are ungrouped in order to avoid placing unnecessary burdens on the underlying \TeX system, and the save stack in particular. This class library also uses explicit paragraph instantiation as the \p element, in its most radical departure from \LaTeX. With this approach we can disable \TeX’s automatic insertion of \par at every blank line. This
Explicit paragraph markup also eliminates subtleties that plague \LaTeX\ neophytes everywhere, for example, the significance of a blank line following an environment.

Many elements count their own occurrences in a document. Sections, theorems, figures, and equations are common examples. We call these counting elements. These elements carry attributes that allow them to maintain and present the count. The author's class library accomplishes this by allocating count registers for such elements as part of the @subclass attribute processing, assigning the count register to the @count attribute in the new class. This type of register allocation is appropriate for elements, such as document sections, whose context is not restricted (unlike an item element, which always appears within a list). Each counting element also carries a number attribute to present the properly formattted element count in the document.

A listing element forms the context for item elements, and an item never appears outside of a list. The item functions somewhat like a counting element. It employs number and @count attributes to present its count in the document, but the count register for the item element is allocated when the containing list is instantiated. The count register is deallocated when the list element is destroyed. This dynamic allocation mechanism avoids placing any constraint (other than the number of available count registers) on the nesting depth of lists.

A section element is a counting block element. We leave it ungrouped, in deference to \TeX\'s save stack. A generic section element could be created as follows:

```
@class@section{@block@counting}{
 @startgroup{}
 @endgroup{}
 <attribute overriding and addition>
}
```

The new attributes would include a @head attribute for typesetting the section head. This could immediately typeset a stand-alone head, or defer a run-in head to the subsequent paragraph.

The format of certain elements is independent of their immediate context. Footnotes are the classic example, but in \LaTeX\ there are others: floating environments like figure and table, parboxes, minipage, and paragraph cells within the tabular environment.

We call these independent elements. How does an independent element separate itself from its context?

Multiple hierarchies. The @parboxrestore command constitutes the \LaTeX\ mechanism for separating an element from its context in the document. It establishes "ground state" values for a set of \LaTeX\ parameters, including \leftskip and \rightskip, \par, \everypar, and so on, from which further changes are made. The author's class library modifies this approach, offering the ability to created named "blockstates" consisting of settings for all block element parameters. Such a state may be established at any time, such as at the beginning of an independent element. A paragraph cell within a table would likely use one such blockstate and a floating figure another.

The problem of \everypar is more interesting. It typically carries out processing that one element defers to another. The information must be passed globally since the element deferring it may be grouped. The current \LaTeX\ implementation subjects \everypar to horrible abuse, dramatically reducing its utility. The author's class library rectifies this with "parstates", analogous to blockstates but global in nature. These consist of parameters that hold information deferred to \everypar, including any pending run-in heads, flags indicating special indentation or suppression of the space above the next paragraph block, and the like. A parstate is an object whose attributes contain the parameter values. A corresponding stack tracks parstate nesting.

The definition of \everypar never changes during document processing. It always executes the attribute of its own name from the parstate object on top of the parstate stack. This attribute refers to its sibling attributes for any required parstate information. Push and pop operations on the stack effect "grouping" for parstates.

Perhaps this section should have been marked with a dangerous bend, for it opens a Pandora's box. The author has concluded that \TeX\'s save stack provides insufficient flexibility. The \everypar treatment just described amounts to a separate "save stack" for parstates. We can replace sole reliance on \TeX\'s save stack with different grouping mechanisms for different sets of parameters.

A hierarchy consists of objects and a stack to track their nesting apart from other hierarchies. Each object, either class or instance, is part of a single hierarchy. The @element hierarchy contains all element classes and instances, and the @element class forms its root, from which all other elements are ultimately derived. The author has implemented four different hierarchies for the class library: @element, @blockstate, @parstate, @rowstate. The last pertains to rows in tabular material. Rowstates make it easy to specify different default behavior for table...
column heads and table body entries within the same column.

An independent element can push and pop the parstate at its own boundaries to insure that it does not pick up any stray material from its "parstate context". Or an element can, as part of its end processing, push the parstate to defer special treatment of the subsequent paragraph. After performing the necessary processing to start the paragraph, that parstate can pop itself from the parstate stack. This requires no cooperation from the other parstate objects on the stack, and therefore does not limit what one can accomplish within the \everypar processing.

Associating objects with hierarchies has another important function. Two or more instances of a particular element class can appear simultaneously within a single document, for example in a nested enumerated list. The user would see something like this:

\open\enumerate{}
\open\enumerate{}
\open\item{}
...  
\close\item
\close\enumerate
\close\enumerate

In this example \enumerate and \item are the parent class names. What objects should be created? Where should the attribute values be hidden? The \@new derives a name for the new object from its hierarchy and stores the new attribute values under this name. To match this, \@free annihilates the most-recently-created object in the given hierarchy. For example, with the definitions
\def\open{\@instantiate}
\def\close{\@annihilate}
the first invocation of \open will create an object named \@element0 and the next will create \@element1. Any invocation of \close will annihilate the last such object. This convention suffices because objects within each hierarchy are well nested.

**Implementing a Class Library**

No single class library can anticipate the multifarious requirements of book publishing. When the inevitable occurs, and a new design moves beyond the capabilities of the available class libraries, the class library writer must extend the system for the design programmer. This section briefly describes the OOPS facilities for writing a class library.

We call one command that defines another a defining word. The commands \newcommand and \newenvironment are \LaTeX\ defining words. The \@class and \@instantiate commands from above are defining words in the author's class library. A set of low-level defining words in the OOPS package form the basis for class library creation. Other macros assist in accessing the information stored in the attributes of classes and objects. We now present these programming facilities.

The construction of a new class library begins with the creation of a new hierarchy. The defining word \@hierarchy takes a hierarchy name and a list of attributes as its two arguments. It creates the hierarchy and a class of the same name, endowing the class with the attributes in the second argument. For example, the \@element hierarchy described in the last section was created as follows:

\@hierarchy \@element{
 \@class{%
 \expandafter\@inherit
 \next@headparent\@preclass
 \global\let\@this@element\@next@object
 \@new
 \expandafter\@inherit
 \@this@element\@subclass}
 \@preclass{}
 \@subclass{}
 \@instantiate{%
 \expandafter\@inherit
 \next@headparent\@preinstantiate
 \@new
 \@current\@element\@initialize
 \@current\@element\@startgroup
 \@current\@element\@start}
 \@preinstantiate{}
 \@initialize{}
 \@startgroup{\begingroup}
 \@start{}
 \@annihilate{
 \@current\@element\@end
 \@current\@element\@endgroup
 \@free}
 \@end{}
 \@endgroup{\endgroup}
}

This creates the \@element hierarchy and the base class \@element within that hierarchy.

Any hierarchy is assumed to support the three basic object manipulation commands \@class, \@instantiate, and \@annihilate. To this end, the \@hierarchy defining word provides functional default values for the corresponding attributes, thus insuring that they are always defined.

The definition of the \@element:\@class exposes the control sequences \next@headparent and \next@object as storage places for material parsed by \@class for use by \@new. The list of storage locations is: \@next@object for the new object.
Conclusions

Compatibility with Plain \TeX{} and \LaTeX{}. This entire OOPS package and the markup it employs is fully, but trivially, compatible with the current plain and \LaTeX{} macro packages. It does make use of a small number of low-level \LaTeX{} macros, but these can easily be provided separately. Because the element instantiation\[\open\text{theorem}{\text{OOPS, There It Is}}\]

\[\end{theorem}\]

executes neither \text{theorem} nor \text{endtheorem} it can safely coexist with the standard \LaTeX{} invocation \[\begin{theorem} \text{OOPS, There It Is} \end{theorem}\]

Shortcomings. This system's Achilles heel is its execution speed. Comparisons with ordinary \LaTeX{} mean little, however, since the functionality is so divergent. Perhaps the ongoing PC price wars will provide inexpensive relief. A more blasphemous solution consists of adding OOPS support primitives to \TeX{}. The author achieved a twenty percent speed increase through the addition of a single primitive to \TeX{}.

People familiar with standard \LaTeX{} do not always easily accept the advantage of a different markup scheme. Document authoring tools that enforce complete markup while hiding the details behind a convenient user interface promise to remove this obstacle. Design programmers who work with \LaTeX{} only reluctantly change their approach to programming. Questions like "Will this be compatible with \LaTeX{}3" cannot yet be answered.

Experiences in production. This new system has seen use in book production with encouraging results. The author has used the OOPS approach to typeset tabular material with a colored background screen spanning the column heads, to handle bizarre
numbering schemes for theorems described above, and also to handle cases of neighboring elements interacting (successive definitions sharing a single backdrop screen). The verbose markup caused some initial concern to users, but automatic formatting of the element was ultimately considered more important.

**Futures.** The combination of authoring tools, design editors, and object-oriented macros constitutes a complete, powerful, and cost-effective document production system. Authoring tools can hide the verbose OOPS markup, making it simple for users to work with. Rich markup in turn allows the automation of most design element features. Design editing tools can manipulate classes derived from a particular class library, or even create new class libraries, thus reducing the time and skill presently required to implement a design.

In a commercial book production setting, where tens of thousands of pages are processed in a single year, the prospect of a twenty percent increase in OOPS code execution speed compels the addition of a single primitive. Extensions supporting complex design features are also appropriate, as are more capable class libraries. An author may not require an extended \TeX{} or a sophisticated class library to produce a magnum opus. Adherence to systematic descriptive markup, supported by the OOPS package, allows a production bureau to apply their more capable system to final production without destroying an author's ability to work with the same source files.

Future \TeX{}-based typesetting systems and authoring tools can benefit from the techniques presented here. The advantages the OOPS approach offers are too great to ignore. Everybody should enjoy them.