ELECTRONIC PUBLISHING, VOL . 4(2), 109–118 (JUNE 1991)

# Using logical objects to control hypertext appearance

P. J. BROWN

*Computing Laboratory*
*The University*
*Canterbury*
*Kent, CT2 7NF, UK*

**SUMMARY**

**It is accepted wisdom that documents should be represented in terms of their logical structure rather than their appearance. Nevertheless most of the popular document processing systems concentrate on appearance rather than structure, mainly because most users opt for a user interface that is interactive, simple and direct.**

**This paper considers issues related to fonts and other appearance attributes within hypertext documents. It first presents the relevant differences between hypertext systems and document preparation systems whose end product is paper. It then goes on to describe a scheme for representing appearance through logical structure. The scheme aims to meet the extra needs of hypertext systems, and yet still to be simple enough to attract wide usage.**

KEY WORDS Hypertext Logical object Font Guide

## INTRODUCTION

This paper is about introducing fonts, colours and other attributes into hyperdocuments. We shall start, however, by reviewing the perceived wisdom for document preparation systems in general, since there is now a considerable experience of their use.

## SOME PERCEIVED WISDOM ABOUT FONTS IN DOCUMENTS

When a user says 'I want to introduce fonts into my document' he is thinking at the wrong level. What he should say is 'I have several different kinds of logical object in my document. I wish to distinguish these. When I display the document I want to use fonts to make the logical objects look different'. The logical objects might be components of the presentation structure of the document, such as section headings, or they could be objects that relate to the topic of the document, e.g. book titles that are referenced in the document, part numbers, or, in a document about programming, variable names.

The best approach is therefore for a document processing system to allow the user to define and use logical objects. These logical objects will have several properties, one of which is the font (or fonts — maybe screen fonts are different from paper fonts) in which it is displayed. The advantages of this approach are twofold:

(1) It is easy to make systematic change, e.g. to change the font in which book titles are to be represented. This can be done even if book titles are currently displayed in the same font as, say, section headings. Overall it is easier to

0894–3982/91/020109–10$05.00
*Received 6 September 1991*
*Revised 13 November 1991*

convert the document between different representations and even between different source formats.

(2) The meaning of the document, rather than just its appearance, is captured. This advantage is somewhat abstract, but can be expressed as a more concrete example: it would be easy to write a tool that extracted all the book titles referenced in a document.

Users should not, therefore, generally use fonts directly. Instead they should work via logical objects, and indeed this is particularly encouraged by the SGML standard. The reality is, however, that the average user of a word processor *does* use fonts directly, because that is the simple way; the user worries later about any problems that this short-sighted approach brings.

Thus most of the popular WYSIWYG document preparation systems are often condemned as 'What You See Is *All* You Get', but such systems are hugely successful in the market. Recently there has been some move towards separate specification of formatting information, typically in the form of *style sheets*. Nevertheless, as the very name reveals, there is still an emphasis on appearance rather than on logical form.

There do exist excellent WYSIWYG systems that are centred on the logical nature of material, notably Grif [1] and Quill [2]. With these, what you see is certainly not all you get, but they are not (yet?) a large force in the market. The design issues are well described in [3].

To summarize, the message from document preparation systems is that, although logical objects are desirable, they must be presented in a very simple way if users are really going to use them.

## HYPERTEXT SYSTEMS

We now focus on hypertext systems. All that has been said so far applies to hypertext systems, but there are important differences that may affect needs. Some of these differences are:

(1) *the on-line difference*: in hypertext the on-line form is the primary form, and any paper form is very much secondary.

(2) *the integration difference*: following on from the above, the hypertext system must work together with other programs. Users want solutions, not hypertext systems *per se*, and the hypertext system is normally part of the solution. Often the hypertext system is running at the same time as other systems, exchanging documents dynamically with them.

(3) *the directed graph difference*: in hypertext the underlying structure is generally a directed graph, not a tree.

(4) *the interlinked corpus difference*: it is more common to have several different documents open at the same time, e.g. a tutorial document that links to a reference document. More generally, document boundaries may be blurred and a massive body of material may be collected together in one interlinked corpus.

We shall discuss the impact of these differences as we proceed through the paper. There are two further differences, both following from hypertext's directed graph structure, whose impact is readily apparent:

(5) *the non-linear difference*: hypertext is not processed linearly. Thus techniques such as *troff's* setting of global variables to represent current point-size, etc., as the scan proceeds through a document, are inappropriate.

(6) *the chunk difference*: hypertext consists of several small chunks linked together, rather than one stand-alone document. In some hypertext systems that use the scroll model, such as Guide, chunks are actually inserted within other chunks when the document is displayed on the screen. Moreover the same chunk may be inserted at several places in the document. It is therefore vital that there be some concept of inheritance, whereby a chunk takes its properties from the chunk that linked to it.

This paper describes an essay in implementing logical objects within a hypertext system, with the dual aims of building on the experience of document processing systems, but catering for these six differences. The hypertext system for which the implementation has been done is the UNIX version of the Guide hypertext system [4], which is a product from the University of Kent at Canterbury. Henceforth reference to Guide should be taken to mean this implementation of Guide, rather than the similar and better-known PC/Mac version of Guide, which is a product of OWL [5]. It is not, however, assumed that the reader is familiar with (either implementation of) Guide.

## GUIDE'S APPROACH

Guide's basic approach to logical objects is a standard one: the user chooses a name for each type of logical object, and supplies a table which attaches properties to this name. Properties include fonts, colours, etc. A property that is not specified takes on a default value. It was a struggle to find a suitable term for these logical objects — the term 'logical object' is not in itself one to set the blood tingling. In the end they were called 'contexts', a term which aptly describes what happens when other material is inserted within them, but nevertheless is far from perfect. The table that specifies the names and properties of contexts is called the *context-table*. Contexts, like most Guide objects, can be nested to arbitrary depth. If a property is not specified for a context that property is inherited, for each instance of the context, from the containing context.

Document preparation systems have structural components such as paragraphs, and hypertext systems have additional structural components concerned with linking. Thus Guide has replacements, buttons, enquiries, etc. Such structural components are often called objects, and one approach is to combine them with the logical objects we have just described into one uniform mechanism. In this case matters concerned with hypertext presentation, such as 'this is a button', would simply be a property of a logical object; a button thus becomes a way of representing a logical object, rather than a logical object itself. (More generally, the aim of most hypertext models is to separate out the components of hypertext into distinct layers, rather than having mechanisms that intermix content, structure and appearance [6, 7].) In spite of the attractions, Guide has not adopted this uniform approach, mainly because unifying rather different types of object does not make for simplicity. Thus in Guide a context is a different animal from a button or a replacement, but nevertheless all structural components must be properly nested. Hence if an instance of a context lies within a replacement it must lie entirely within that replacement; this is a natural rule as otherwise structured editing would be a nightmare to both user and implementer alike.

Following systems such as Quill, properties can be set relative to properties of the containing context. A font property might therefore be:

- the same as the containing font, but italic.
- the same as the containing font, but two points bigger.

We call this *relative inheritance*.

The value of relative inheritance arises because of difference (6) — the chunk difference: for example, assume a chunk of text includes a title that is to be in a bigger font than the surrounding text. This can be represented by a context called *Title* with a property that its font is two points bigger than its containing font. The chunk of text may be inserted into several different places in the hyperdocument. In each case the containing font may be of a different size, but the title will adapt. As a further example, the chunk could be used by two users, one of whom had a small default font, and the other a large: again the size of each *Title* would adapt accordingly.

It is necessary to provide some flexibility with relative inheritance: for example, specified fonts may be unavailable or too big/small to read. To achieve this, authors of context-tables can provide fall-back options, e.g. one point bigger rather than two points, or, as a last fall-back, exactly the same as the containing font.

## GRAMMATICAL RULES FOR CONTEXTS

In [3], Furuta *et al*. discuss the relative merits of the interrelationships between objects being constrained or unconstrained. It is another discipline v. freedom issue. Bearing in mind the reluctance of users to accept discipline, even if it is in their own long-term interest, Guide has opted for freedom. There is therefore no grammar associated with contexts: any context can contain anything. (It often does: a complete Guide hyperdocument can be enclosed in a context.) The only restriction is the rule for proper nesting. An unconstrained approach is, in fact, the usual one for interactive document preparation systems [8].

## READERS AND AUTHORS

A consequence of difference (1) — the on-line difference — is that hypertext systems need to cater for two classes of users:

- readers, who work solely with the document as it appears.
- authors, who need to be aware of the logical structure of the document.

Guide readers are only aware of contexts to the extent that contexts change the appearance of the document. Thus readers are aware that Guide supports fonts (etc.) in some way, but can be totally unaware of the concept of logical objects — probably a good thing because evidence from the use of word processors indicates that if they did need to know about logical objects, they would be put off. (Actually readers probably are subconsciously aware of logical objects that relate to the structure of the document, such as chapters or sections, because of familiar typographical conventions for representing them — assuming the author has followed these conventions.)

Authors *do* need to know about contexts. In Guide, authors see the document as a reader does, but they get an enhanced view of the document, which shows structural

boundaries. These structural boundaries are indicated by inverse-video metacharacters embedded in the document. A sample line containing two instances of the context *Book-title* would be seen by the author as

Jane Austen's {Book-title **Pride and Prejudice** } and {Book-title **Northanger Abbey** } are

Here the 'metacharacters' show where each instance of a *Book-title* starts and ends. (The '}' character indicates the end.) The reader would not see the metacharacters, i.e. the line would appear as

Jane Austen's **Pride and Prejudice** and **Northanger Abbey** are

The menu for creating contexts is controlled by context-tables: thus if *Book-title* is in the context-table it appears in the author's menu.

## WORKING WITH OTHER TOOLS

Following on from difference (2) — the integration difference — most real hypertext applications involve working with other tools. Moreover it is the UNIX philosophy that tools should not be monolithic, but instead that several tools should naturally work together — perhaps in a pipe — on the same source file. Guide source files are therefore textual, to allow the use of most UNIX tools. The ideal would be an accepted official standard format, accepted by all tools that might process documents. Since this does not exist, the next best, in a UNIX environment, is the *troff* style of embedded mark-up, which is something of a *de facto* UNIX standard. Thus Guide uses a *troff* style of mark-up: our above example of a *Book-title* would be stored in a source file as

```
.Co Book-title
Pride and Prejudice
.cO
```

Such a representation makes it easy to write a UNIX script to, say, extract all the *Book-title*s from a Guide document, sort them into order, and delete duplicates.

Guide could equally well have been designed, like Quill, to use an SGML mark-up. However, UNIX tools such as *spell* assume a *troff* mark-up, and this swung the balance.

The main thrust of the integration difference is that a hypertext system is often wrapped intimately together with other software to provide a solution to a user's need. A good example of this is ICL's LOCATOR fault diagnosis system [9], which wraps Guide together with (a) a system that logs fault reports from customers, and (b) a system for controlling the dispatch of engineers to fix faults. Guide extracts information from its hyperdocument and passes it, as each fault is diagnosed, to the dispatch system. This information includes part numbers. Such part numbers are of no particular interest on the hypertext side, though they form part of the text displayed on the screen, but are objects of special interest to the dispatch system. In order to identify them they can be marked as instances of the context *Part-number*. On the hypertext side this context has no properties concerned with appearance: it is therefore displayed in the same way as normal text. The marking of *Part-numbers* only comes into its own when text containing a *Part-number* is passed by Guide to the dispatch system: the dispatch system can readily identify part numbers within the material passed to it, and there is no ambiguity between true part numbers and other strings of characters which might happen to look like part numbers.

This use of contexts — marking objects that are of interest to some other system — is a frequent usage. It is possible to make such objects invisible to readers if they are irrelevant to the hypertext side. However, the existence of such objects is always apparent to authors, so that they are not treated wantonly when a document is edited.

## SAVING

Guide supports two ways of saving material: a structural save of the hyperdocument itself, and a what-you-see save, covering the hyperdocument as currently displayed on the screen (i.e. with the current state of button expansions). Cut-and-paste can be considered as a special case of save and restore, and the same two approaches are available. One of the properties of a context is a specification of what to do on a what-you-see save: the user might want to record some indication that the context was there. Specifically properties can be set to insert strings before and after each instance of a particular context. If, for example, the saved material was to be inserted into a *troff* document, each instance of a saved *Book-title* could be prefixed by the string '\fI' and followed by the string '\fP', thus causing the book title to appear in italics when formatted by *troff*. (This facility is analogous to a facility found in SGML.)

## LIBRARIES AND CONTEXT-TABLES

Sometimes hypertext is used as an aid to writing or thinking, and hyperdocuments are read by the author and no-one else. More often, however, hyperdocuments are prepared for a number of different readers, possibly distributed over different sites, and working in different hardware/software environments. This leads to a requirement that hyperdocuments be self-contained. In our application this requirement says that each hyperdocument should include its own context-table, which contains context-specifications for all the contexts used in the hyperdocument. Each context-specification gives the name and properties of a context.

If a reader wants to change some properties, e.g. to make fonts bigger or to exploit colour more, he can make a copy of a hyperdocument and edit the context-specifications within the context-table.

The above requirement for diversity can be matched by an equally strong requirement for centralization. If all context-tables are combined into a single library then a change in the properties of a context can be effected just by changing its context-specification within the library. If, instead, there are hundreds of documents, each containing a copy of a given context-specification, then making the change becomes next to impossible.

The centralization versus diversity issue arises in many branches of computing (e.g. with SGML DTD libraries, subroutine libraries, graphics libraries, etc.), as it does in business, and there are no absolute answers. There are some possible compromises, like centralization with selective local overriding, and Guide follows this approach.

The details are as follows. Guide has a central library of context-specifications. Ideally this should be the same on all Guide installations, just as, for example, ideally the library of built-in functions for C compilers should be the same on every C installation. Reality is of course otherwise, but the hope is that, as with C, serious problems will be rare. The library will generally contain 'generic' contexts such as

| | |
|---|---|
| *Emphasized* | for emphasized words |
| *Greek* | for a Greek font |
| *Bigger* | for bigger text. (Instances of this may well be nested, to give doubly bigger text.) |

In addition each Guide hyperdocument can contain its own context-table. Each instance of a context within a document has, as we have seen, an associated context-name, and the mark-up of the instance takes the form:

```
.Co Aname
some text
.cO
```

On finding this mark-up within a document, Guide first looks for *Aname* in the context-table (if any) for the current document, and only if this fails is the library searched.

Specialized contexts, such as our *Part-number* example, tend to be placed in local context-tables, as do changes made by individual readers, i.e. the reader can override the properties of a context in the library by re-specifying the context locally within a document.

Contexts can be local to a part of the document that the user sees on the screen: if a file is inserted within a Guide document as the replacement of a button, its context-table (if any) is local to the current replacement. The replacement of a button can be generated by running a UNIX shell script. In this case the shell-script, if it needs to, can generate a context-table as well as some text. Context-tables use the same *troff*-like mark-up as the rest of a document. An example of a line within a context-table is

```
.\" Book-title  F=-*-lucida-medium-r-*-*-14-*-*-*-*-*-*-* RGB=(100,40,250)
```

This gives the context-specification of the *Book-title* context with its font (F=) and colour (RGB=); it is represented as a *troff* comment so that the line is stripped out when a spelling check of the document is made. Such local context-tables are useful when the shell script interfaces with another tool that generates material that is to be viewed in an idiosyncratic way, e.g. using special fonts.

When replacements are nested, a particular instance of a context could lie within the scope of several context-tables: this is a similar situation to nested variable declarations in a block-structured programming language, and similar rules apply, i.e. the most closely encompassing declaration takes precedence.

In general, Guide's scroll model, with replacements within replacements, tries to give the user a hierarchical illusion, though the underlying data structure is not a tree [10]. The scope of a context-table is based on this hierarchy, i.e. the scope is a contiguous area of the user's scroll. This helps surmount difference (3) — the directed graph difference: in particular it overcomes the problem that the directed graph which underlies most hyperdocuments is not a good base for applying such concepts as locality of scope.

As we have said, the Guide author's menu always contains commands for creating instances of each context specified in the current context-table(s). The menu also includes any library contexts used in a current document; there are mechanisms for enhancing this to make the menu encompass *all* contexts in the library. The current context-table (and therefore the menu) may grow as a document is built up. For example, a hyperdocument, like any other document, is sometimes created using cut-and-paste

from other documents. When some material is copied from one Guide hyperdocument to another, Guide checks for any instances of contexts within the material, and may as a result augment the context-table of the receiving hyperdocument. Thus if an instance of a *Part-number* is pasted into a hyperdocument the hyperdocument's context-table is automatically augmented by a specification of *Part-number*.

Difference (4) — the interlinked corpus difference — leads to a problem with the name-space: one author may choose a name for an object, and another author may choose the same name for a completely different type of object. For example two Guide authors may choose the name *Part-number* for completely different contexts. The problem comes to light when the work of the two authors is combined.

The scheme we have described, which has a name-space for contexts that can be (a) local to a replacement; or (b) local to a document, or (c) global to all documents using a particular library, goes some way towards tackling this problem. The scheme has the defect, however, that the name-space and the properties specified in context-tables relate only to Guide. In our *Part-number* example, its properties would not be known by other tools that used *Part-numbers*, except in the unlikely case that such tools made the effort to understand Guide's context-tables.

## COLOUR

One property of a context is the colour used to display material within that context. (In fact there are several colours associated with each context, but we will not go into details here.) Colours can be used as a complement to fonts or even as a substitute (e.g. a user might choose to display the *Emphasized* context in a brighter colour rather than in a different font).

Colours can be defined in a relative way, e.g. 20% redder than the containing colour. If used with considerable restraint this can be a valuable facility: it would probably be much more valuable if Guide used an HSV (also called HSB) colour model rather than its current RGB model.

## CONCLUSIONS

A hypertext system is different from a system for preparing paper documents. We have outlined several differences, but the most important are

- what the reader views is a document made up from several disparate chunks.
- the hypertext system needs to work directly with other tools.

We have described a scheme for introducing logical objects into one hypertext system, Guide. The scheme has many similarities (by accident rather than by design, as it happens) with the Quill system for preparing conventional paper documents. Important features of the scheme are

- a simple embedded mark-up that can easily be used by other tools. The same style of mark-up is used both for instances of the logical objects and for their definitions.
- relative inheritance.
- a mechanism for the scope of a logical object to be local to a document plus those documents it links to.

- a way of allowing readers to focus just on the appearance of logical objects, but authors to be aware of them as structural entities.

ACKNOWLEDGEMENTS

Three referees of this paper made shrewd comments that have led to improvement.

REFERENCES

1. Vincent Quint and Irène Vatton, 'GRIF: an interactive system for structured document manipulation', in *Proceedings of the International Conference on Text Processing and Document Manipulation (EP86)*, ed. J.C. van Vliet, Cambridge University Press, pp. 200–213, April 1986.
2. Y. Wolfsthal, 'Style control in the Quill document editing system', *Software — Practice and Experience*, **21** (6), 625–638 (1991).
3. R. Furuta, V. Quint, and J. André, 'Interactively editing structured documents', *Electronic Publishing — Origination, Dissemination and Design*, **1** (1), 19–44 (1988).
4. P.J. Brown, 'A hypertext system for UNIX', *Computing Systems*, **2** (1), 37–53 (1989).
5. OWL, *GUIDE: the ultimate way to present information*, OWL International, Bellvue, Wa., 1990.
6. F. Halasz and M. Schwartz, 'The Dexter hypertext reference model', in *Proceedings of the Hypertext Standardization Workshop*, ed. J. Moline, D. Benigni and J. Baronas, National Institute of Standards and Technology, pp. 95–133, 1990.
7. R. Furuta and P.D. Stotts, 'A functional meta-structure for hypertext models and systems', *Electronic Publishing — Origination, Dissemination and Design*, **3** (4), 179–206 (1990).
8. R. Furuta, 'An object-based taxonomy for abstract structure in document models', *Computer Journal*, **32** (6), 494–504 (1989).
9. G. Rouse, 'An application of knowledge engineering to ICL's customer service', *ICL Technical Journal*, **7** (3), 546–553 (1991).
10. P.J. Brown, 'High level hypertext facilities: procedures with arguments', *Hypermedia*, **3** (2), 91–100 (1991).