The Role of Modularity in Document Authoring Systems

Janet H. Walker
Cambridge Research Lab
Digital Equipment Corporation

Abstract

Modularity is a fundamental concept in documents and document development as well as in programming. I hypothesized that the effectiveness of writing professionals could be increased by providing a working environment to support explicit modularity in documents and document development. This paper briefly describes an environment designed to enable testing this hypothesis and evaluates it by means of discussions with professional writers who used it for large, real-world, development projects.

Introduction

Building technical documentation is a form of engineering. In taking this perspective, people about to engage in designing software to support technical writers will find that they share both problems and solutions with designers of software engineering environments. The project described in this paper abstracted techniques for managing complex system development from local software development practice and applied them to the problem of document development.

Modularity and Abstraction in Software

In software development, modularity and its companion, abstraction, are now widely recognized as fundamental to managing program complexity. Modularity is the result of designing a program as a number of small pieces, the modules, from which the software artifact is constructed. Abstraction is the act of naming something and using it for what it does (in a problem sense) instead of for how it works (in an implementation sense [3]).

Effective modularity is not attained by breaking a program into arbitrary pieces. Rather the pieces must be designed to serve some specific, clear purpose in the overall system. The essence of modularity lies in designing the pieces so that each does one thing well and can be maintained independently of the others [5]. To achieve that clarity, the pieces must minimize their relationships with other pieces and make any remaining relationships explicit. Dijkstra [3] points out that clarity has a quantitative aspect as well; something long is not likely to be clear. These considerations suggest that the pieces should be small and that their contents should reflect some single, simple purpose.

When you plan to build something from pieces, the pieces have to be designed to fit together. In architecture, furniture, and even children's block sets, modular parts are often essentially identical or physically interchangeable. In the engineering of intangibles, however, the concept of a building block is less constrained; physical interchangeability, for example, is not an issue. The goal in software is to define pieces that are small, self-contained, mutually independent, general, and hence easy to maintain.

Effective use of abstraction depends heavily on choosing appropriate names for the pieces. A successful name represents the module so that it can be used by name to represent its functionality without requiring that users become familiar with the procedural details of its internals. Hence, modules are named, not with a machine-generated ID or some equally meaningless label, but with some user-assigned name that reflects the essential nature or intended usage of the piece. Good naming is essential to successful use of modularity.

The work on which this paper is based was performed completely while the author was employed by Symbolics, Inc.

There are no automatic ways to decompose a problem. Good modular design depends on the skill of the human beings involved; building a software system, even modularly, remains a lot of work. Easing that work is a primary goal of much current effort in constructing software development support environments. Fortunately, it is easier to build support environments for languages that encourage modular design than for monolithic ones.

Programming environments and modularity

Much research in programming environments was actually carried out in the context of artificial intelligence research [7]. By its nature, AI research deals with large, complex problems; hence, the programming environments that evolved to support AI programming are particularly suited to developing any large software system.

An AI-style programming environment derives much of its power from knowledge about the identity and interrelationships of the modules that constitute the software [12]. For example, the environment itself (rather than batch tools that the user invokes manually) maintains information about the names of all the modules and about the relationships between them.

This base of knowledge can enable searches for module names containing particular words, provide location information for source code, and support incremental compilation and dynamic linking of changed modules. For example, the environment software can continuously maintain knowledge about which software modules call which other software modules. As a result, a programmer can determine easily from the environment which other modules are affected by a proposed design change.

The better the modular decomposition of the system, the more assistance such a programming environment can provide.

Modularity and Abstraction in Documents

The essential job of a document is to create a state change in the mind of a reader, ideally from some confused or uninformed state to a state of full comprehension. The writer is the person responsible for providing the material that enables the reader's state change. For complex technical material, this state change is a highly complex, non-deterministic process, which writers sometimes despair of even influencing, let alone enabling.

The best the writer can do is to make assumptions about the initial states of various kinds of readers, identify necessary component state changes and the material needed to enable each of them, and then compose a document which contains all this material organized according to a well-designed communications plan. The rest is, and must be, up to the readers.

It is conventional in the field of document processing to describe the *logical structure* of documents as a tree, usually consisting of chapter, section, subsection, and other components, arranged according to some document grammar [2, 4, 6]. The semantics of this structure is then expressed in the formatting of the document. In this kind of model, a correct document is one that conforms to the particular tree structure defined for a document of that type.

From a writer's point of view, however, a correct document is one that, independent of its logical structure or appearance, enables the desired state changes in the minds of its readers. Hence the conventional view of document logical structure does not address the essential issue for writers, which is that of enabling the transfer of meaning.

From this alternate point of view, a document is composed of meaning elements or modules, each designed to contribute to desired state changes in the minds of the readers. Thus, a module in a document is a semantic building block rather than an arbitrary physical block of content (for example, a page or a screen). A module is a component that says something self-contained and comprehensible. Well-designed modules would have few connections to other modules; the ones that they do need (to prerequisite and subsequent modules) would be explicit.

Titles of modules in a document reflect the material that the modules contain. Thus, abstraction for a writer consists of being able to use the name of a module to stand for what it is supposed to communicate, rather than for the exact sentences therein.

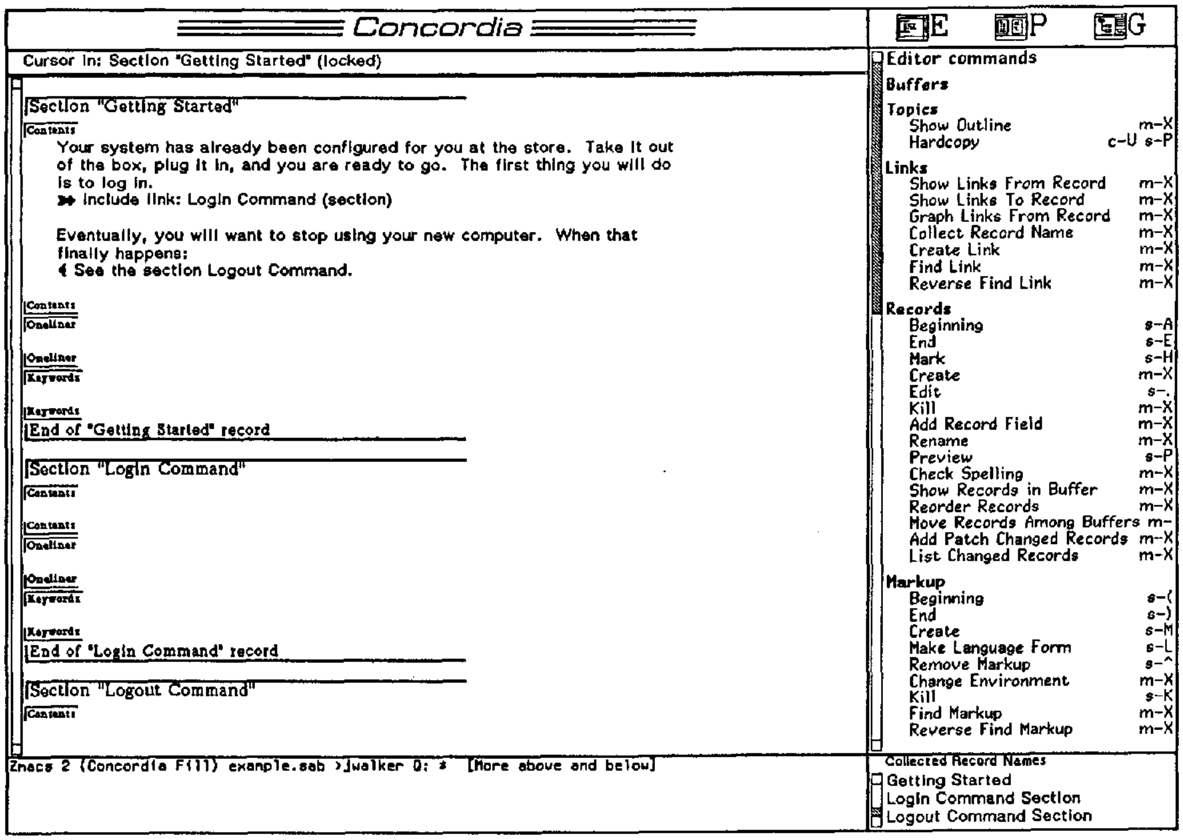
Design and Implementation of the Authoring Environment

In undertaking this design, my hypothesis was that writers need to manage the meaning elements of a document independently from and in addition to its physical appearance specification. This perspective led in two major directions: to documents based on modularity and to a support environment for managing the modules.

As described earlier, the major power of a good software development environment comes from its knowledge of the components of the software system. We built a support environment for documents based on analogous knowledge of document components and the ways in which they can be connected. The support environment is known as the Concordia environment[1]; its features are described in more detail elsewhere [10]. The brief description of it in this paper concentrates on the aspects of it that are relevant to the discussion of modularity.

The Concordia editor is a form of structure editor that maintains modules, connections, and formatting markup as object-oriented data structures instead of text [11]. Writers edit the textual parts with standard, unconstrained text-editing com-

[1]Concordia is a trademark of Symbolics, Inc.



```
Cursor In: Section "Getting Started" (locked)

Section "Getting Started"
Contents
    Your system has already been configured for you at the store.  Take it out
    of the box, plug it in, and you are ready to go.  The first thing you will do
    is to log in.
    ➧ Include link: Login Command (section)

    Eventually, you will want to stop using your new computer.  When that
    finally happens:
    ◀ See the section Logout Command.

Contents
Oneliner

Oneliner
Keywords

Keywords
End of "Getting Started" record

Section "Login Command"
Contents

Contents
Oneliner

Oneliner
Keywords

Keywords
End of "Login Command" record

Section "Logout Command"
Contents

Zmacs 2 (Concordia Fill) example.sab >jwalker 0: *  [More above and below]
```

Figure 1. A screen display from the Concordia editing environment, showing both completed modules and a template for a newly defined module.

mands; they edit the structural framework with special-purpose structural commands. Since the material being edited consists of modules for a document, rather than the document itself, the goals of WYSIWYG editing do not apply. The editor instead presents an interface that combines elements of both WYSIWYG and generic markup, along with the structure editing (Figure 1).

Modules

Each document module has two identifiers, an internal, machine-generated unique ID and an external, writer-generated "name." From the software's point of view, the external names need not be unique. External names consist of a title and a type. The title reflects the content; the module types (section, variable, function, and so on) reflect the writer's semantic classification of the module rather than any characteristics related to layout.

The modules are created independently, with as many modules per file as writers find convenient. The placement and ordering of modules in files is not at all related to where or whether they actually appear in documents.

Modules contain information besides that appearing normally as the "subject matter" of the module. Information for any topic can be categorized in several ways to include material for end-users, keywords for indexing, and fields for notes or status information. Figure 1 shows several modules in the editing environment, with the different classes of information visible.

Connections and documents

Documents are composed from modules by means of connections (or links) from within the contents of a module to another module. Two operations, inclusion and reference, constitute the two construction operations for creating a document from its building blocks. In the example in Figure 1, the writer of a conventionally structured technical document would want to either *include* the description of the login command within the section entitled "Getting Started" or else *refer* to it from there.

Inclusion means that one module can specify that some of the contents of another be included within it. The writer uses a *symbolic* link to make the connection to the module that is to be included. References serve the function of conventional cross-references but are created and manipulated symbolically instead of being typed in simply as text. Figure 1 contains several examples of inclusion and reference operations.

The lines containing the connections are not editable as text; they are presentations of internal data structures rather than arrays of characters. The writer uses commands to change the connections. For example, a link can be changed from inclusion to reference type using a command to edit the link.

Authoring environment "feel"

In working on Concordia documents, writers are not editing running text; they are working with independent modules of the document, which vary in size and in nature. A high-level module in a document might be a section that contained a short introductory paragraph and a set of connections to included modules. A low-level module might be a description of a command option that contained several paragraphs of description, possibly with a few references to modules describing similar options.

The differences that make one module a chapter and another a low-level reference item are simply in how the writer supplies their contents and connections. Something can be published as a stand-alone book or can appear as a chapter within a larger collection, with no change to the material. The process of following the connections at display or publication time determines the level (chapter, section, and so on) of a module within a document.

All modules are treated alike by the software and all commands apply equally to any module. For example, the writer can run the formatter interactively on any module in order to see that module on the screen as it would appear within a particular finished document.

The Concordia environment maintains all modules and connections as data structures that are simply being presented conveniently, mostly as text, for editing purposes. It maintains complete knowledge about the location of each module within files and within documents. It knows how each module is connected with others. This knowledge is the basis for the commands that enable writers to work as conveniently with these structures as with text. For example, Figure 2 shows the results of a command that displays the connections to and from a specified module. All module names in the graphic are active and could be clicked on to invoke various editing commands that take modules as arguments.

Results

This modular methodology, with several different generations of user interface, has been in use at Symbolics, Inc. for producing technical documentation since late 1983[2].

At the beginning of the project, the then-existing document set of around 2,500 pages was decomposed into modules in Concordia internal file format. The standard document set, when printed in early 1988, was about 7,000 pages. It was then composed of roughly 12,000 modules with 23,000 inclusion and reference connections. Modules ranged in size from several hundred characters to over 30,000 characters, with the average size being about 1,000 characters.

The module size provides at best a rough upper estimate of the size of the processed text content, since the modules are stored in binary form with a significant amount of structural information stored in each module. In addition, the raw module size gives no indication of the displayed size when other modules are to be included, because the included modules are represented only by symbolic commands. The raw module size does, however, give a good indication of the visual size of the units that the writers have chosen to work with.

Assessing the writers' experience

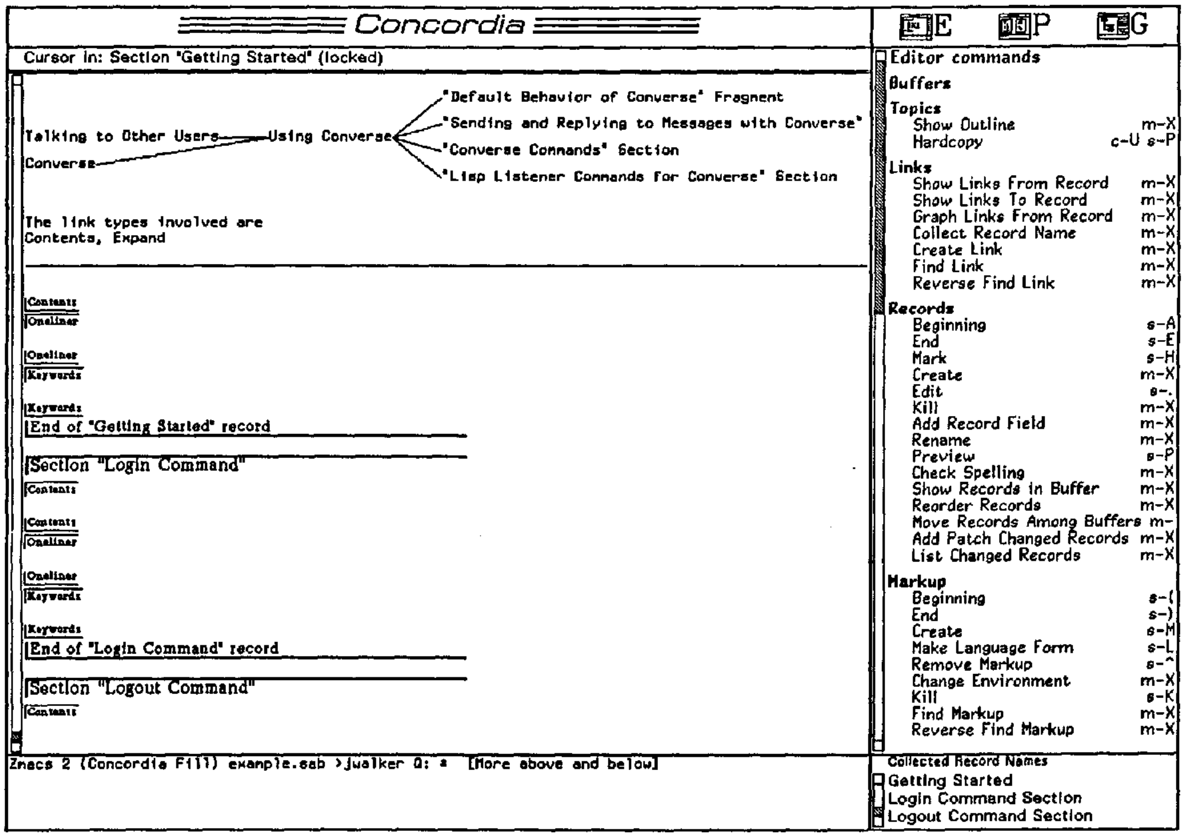
The hypothesis that a document engineering environment built around this kind of modular document can support writers effectively has been tested daily for over five years by the Symbolics, Inc. documentation department. The experiences and preferences of the writers were used to refine the overall design and to design the final user interface.

For the purposes of this paper, I interviewed a group of four writers. These writers averaged three and a half years of experience with this methodology (minimum three years) and 14 years experience as professional writers (minimum seven years). The rest of this section provides a digest of their comments about modularity and the support for it in the engineering environment.

Modularity

What criteria did people use for breaking their subject matter into modules? Writers mentioned a number of issues. In many cases, the most fundamental decomposition was dictated by the inherent nature of the artifact being documented because, in technical material, many of the topics in documents mirror the structure of the software or hardware components they describe. The writers also thought about their readers and created modules for topics that they believed readers would be trying to find.

[2]The most recent interface was released in mid-1988 by Symbolics, Inc. as the Concordia document engineering environment.



```
Concordia
Cursor in: Section "Getting Started" (locked)

                                                  "Default Behavior of Converse" Fragment
Talking to Other Users——Using Converse            "Sending and Replying to Messages with Converse"
Converse                                          "Converse Commands" Section
                                                  "Lisp Listener Commands For Converse" Section

The link types involved are
Contents, Expand

Contents
Oneliner

Oneliner
Keywords

Keywords
End of "Getting Started" record

Section "Login Command"
Contents

Contents
Oneliner

Oneliner
Keywords

Keywords
End of "Login Command" record

Section "Logout Command"
Contents

Zmacs 2 (Concordia Fill) example.sab >jwalker Q: *  [More above and below]
```

```
E   P   G
Editor commands
Buffers
Topics
   Show Outline                   m-X
   Hardcopy                   c-U s-P
Links
   Show Links From Record         m-X
   Show Links To Record           m-X
   Graph Links From Record        m-X
   Collect Record Name            m-X
   Create Link                    m-X
   Find Link                      m-X
   Reverse Find Link              m-X
Records
   Beginning                      s-A
   End                            s-E
   Mark                           s-H
   Create                         m-X
   Edit                           s-.
   Kill                           m-X
   Add Record Field               m-X
   Rename                         m-X
   Preview                        s-P
   Check Spelling                 m-X
   Show Records in Buffer         m-X
   Reorder Records                m-X
   Move Records Among Buffers m-
   Add Patch Changed Records      m-X
   List Changed Records           m-X
Markup
   Beginning                      s-(
   End                            s-)
   Create                         s-M
   Make Language Form             s-L
   Remove Markup                  s-^
   Change Environment             m-X
   Kill                           s-K
   Find Markup                    m-X
   Reverse Find Markup            m-X
Collected Record Names
Getting Started
Login Command Section
Logout Command Section
```

Figure 2. Asking to see how a particular module is connected finds everything that refers to it or includes it and everything that it refers to or includes.

When a particular piece of information was needed in several places, they would create a separate module for it. Not necessarily self-contained, these modules were used for redundant information, like standard sentences describing some ubiquitous command option. Finally, writers subdivided sections that were getting "too large" to be manageable and felt that they had developed a sense for the appropriate size of modules.

They agreed that writing modularly, that is, putting together material that belongs together, had always been a goal for them in technical writing. Part of teaching new technical writers consists of helping them discover how to determine the logical structure of complex material and turn that into a document that readers can understand. The writers mentioned that conventional writing tools make them do all that work in their heads, providing no support for the cognitive, logical decomposition process that was taking place. The modular writing methodology provided by this environment "reinforces the need to think things through logically" while at the same time not forcing any particular work style.

Re-usability

As the project progressed, writers identified more opportunities for re-using modules. It took experience and practice to recognize the opportunities, but they felt comfortable with writing material to be shared among different writers and documents. Several experimental documents were built by selecting and arranging existing material. One writer commented, "Writers who think modularly find creating new documents from already written pieces, with audience appropriate introductions, to be delightfully easy."

Module size

The writers agreed that with experience they developed an intuitive sense for optimum module size. When they found themselves adding extra keyword phrases (that weren't just synonyms) to a module, it was a warning that the section should be further decomposed. The names of modules reflected the contents well enough that extra manual indexing was rarer than we had anticipated.

Upon finding it confusing to work on a particular module, they would look for ways to break it down further. At the lowest level, they worked primarily with modules that contained one to several paragraphs.

One writer had the opportunity to work on a textbook as well as technical manuals. Originally, she anticipated that modules in the book would be larger than those in a technical manual because readers would be working through the book linearly and because she would want much tighter writing style and passage flow. In the course of working on the book, she found she had difficulty with what she was writing because the modules were too large and had too much material in them. After she broke things up further, it was easier to manage the organization of the material and to have better access to it while writing. She felt that this modularity helped her significantly in doing the writing and did not in any way compromise her overall style goals for the book.

Naming

The documents produced by the writers were either printed in hardcopy or displayed online with a specialized reading interface [9]. For usage by online readers, the software initially required that modules have unique external names (within their particular type). Originally all writers viewed this as a serious impediment because they were accustomed to using certain standard names often, like "Introduction." After several years of experience, they grew accustomed to using highly specific names and found that these helped them in clarifying their writing, as well as helping readers identify topics likely to be of interest.

A module has to be named at the time it is created. Some writers had been uneasy about giving a name for something before writing its contents, in case the act of writing changed the intent of the section. This turned out not to be a problem since they could either work out the content separately and then create a module containing it or else use a command to rename modules (which automatically updated all the connections to them).

Abstraction

The writers found that the amount of effort required for working on a complex document was significantly reduced. Separating content, organization, and formatting made it easier to concentrate on one at a time.

They could work on the structure of high-level section modules (that is, the set of symbolic connections to other modules) without regard to the actual contents of the included modules. For low-level description modules, they could work on the content without regard to where it fit into one or more different documents. In effect, the modular organization enabled writers to focus on the activity most appropriate to the module's level of abstraction within the document.

Document organization

In a hierarchical document, the organization is modified through reordering or moving the symbolic connections to other modules. In some respects, this kind of capability is present in outliners or in other document processing systems (for example, Grif [6]).

The difference between the current system and others lies in what happens when a writer decides to move a section to a different level within a document. In conventional systems, each portion of the text is identified by its level within the document so that moving it requires changing its syntactic identity. That is, in a standard hierarchical document, writers have to not only cut and paste the right span of material but also change the designations for all of the headings. For example, to move a chapter "down a level" in a document, they would have to change the chapter into a section and then all the previous sections into subsections and so on, usually a tedious and error-prone process. In the current system, the position of a module in a document hierarchy is determined dynamically by the inclusion connections encountered during display or printing, not by some predetermined designation that says "this is a chapter."

The writers made organizational changes heavily, particularly during early stages of document design, and commented on the ease of making and evaluating these experiments with structure.

Environment

Perhaps the largest number of comments were related to the usability of the structure editor component of the system. In the early days of the system, the writers used the same modular methodology, but a different user interface that was implemented by an embedded command language for marking the structural boundaries, connections, and formatting directives [8]. The change from embedded, textual commands to a true structure editor made the editor more usable by eliminating all the preventable errors in specifying structure and markup. The writers commented that simulating structure with embedded commands makes them do more work because the structure exists only in their minds, without technological support from the environment. Essentially, it was not using a markup language itself that caused early difficulties, but rather the lack of high-level knowledge and debugging support in the environment.

Document style

Early in the project, there was significant concern among both readers and writers that the modular methodology would cause a decline in writing quality. The issues most often mentioned were flow and mixing of styles from different writers. These early concerns have been allayed.

The writers recognized that technical material is not designed for beginning-to-end reading and that people use the manuals idiosyncratically, by starting anywhere and reading only as much as seems relevant [1]. As a result, they felt that seamless, flowing prose was most important within a module and that trying to connect modules tightly and seamlessly was not a good idea anyhow; the reader has to have some textual hints about when to stop. Despite their early fears, the writers now feel that the documents produced with this style of writing development are consistent with normal standards for running prose.

Early in the project, the writers became what they described as "cross-reference happy." (Those cross-references were not typed in as text sentences but were created by the display software as a result of processing a reference link.) The writers found these kinds of automatically generated cross-reference sentences intrusive. They proposed changes in the system to enable integrating symbolic references into text sentences, which enabled better running prose. Although the prose is smoother, the current design does not support sophisticated presentation of cross-references based on the reader's context. This is an open research issue.

Readers and writers hearing about this technology for the first time comment on the potential for readers being distracted by a mixture of voices in a document hierarchy that contains pieces written by many writers. The writers interviewed felt that the concern is real but that it had not been a problem in practice. This group, like most professional documentation groups, has a "house style" that mandates basic things about passive voice, imperatives, tense, terminology, and so on, which takes care of basic coordination between writers. The further problems of several writers writing modules to fit several purposes were handled by the editors and writers involved, who cooperated in finding a solution that all could be satisfied with.

Group writing

The current system was designed to support a group of writers, working together on a document set, as opposed to individuals, all working independently on their own book(s).

Most of these writers had experienced situations where the software engineer associated with a project handed them either "bottom-level" modules or complete draft documents. The writers then either organized, reorganized, or reconceptualized the material. They all found this to be a very satisfying way to work because they could concentrate on "adding value" to basic material.

The writers mentioned that the modular organization of documents and the facilities in the editor for locating modules made it easier to maintain existing books. In addition, all the writers had experience taking over a partially completed book from another writer. In these cases, the tools for examining the structure of a document made it easier to understand the original writer's design. Also, they tended to make prototype books more complete, with many empty modules in place just to help them keep the overall structure in mind.

Quality

Writers felt that they produced documents of higher quality using this system. This perception is an indirect result of treating document development as an overall engineering process like software development. The integration with the engineering process aided writers sociologically in that their work methods were thereby understood (and respected) by engineers.

Because the document database was installed and maintained incrementally as a integrated part of the software environment, documents were in use by the development community from very early in the prototype stage. As a result, they had been through an unusual amount of informal usability testing long before any formal reviews took place.

Work style

One of the writers worked primarily top down, creating an overall organization of empty modules before she wrote much content. Others worked primarily bottom up, creating a module for each fact that they uncovered before or while working on the overall document organization. Another used both styles at different times. According to the writers, the system accommodated both styles of working equally well.

One writer commented that the system helped her avoid writer's block because there was never any danger of sitting looking at an empty screen, wondering how to start; you always knew *something* and could start by creating a module for it.

Conclusions

Experienced software technical writers have few problems mastering the modularity discipline, as it simply reflects how they think. The writers interviewed for this paper expressed strong preference for this environment, which supports how they approach their task, that is, modularly, with a need to separate issues of content, organization, and appearance.

The Concordia project is an example of how conventional document sets can be produced using a modular methodology and a supportive, interactive development environment. Successful use of this system at Symbolics, Inc. and elsewhere suggests that major improvements in document production can be realized by importing concepts from other branches of engineering. Whether this methodology offers unique advantages over other approaches to structured documents, like that of Grif [6], is a question for further research into the needs and practices of technical writers.

Acknowledgements

Thanks to all the document engineers at Symbolics, Inc., who helped me investigate the strengths and weaknesses of modularity in technical documentation, with special thanks to Ellen Golden, Tom Parmenter, Sonya Keene, and Jackie Covo. The Concordia structure editor was designed by Rick Bryan and implemented with assistance from Bill York and Dennis Doughty. Management commitment from Ilene Lang and Tom Diaz made the project possible. Discussions with Brian Reid, Mary-Claire van Leunen, and Craig Schaffert helped shape this paper. Dick Beane and Andrew Black provided helpful editorial comments.

References

1. Carroll, J. M., Smith-Kerker, P. L., Ford, J. R., & Mazur-Rimetz, S. A. "The Minimal Manual". *Human-Computer Interaction 3*, 2 (1987-1988), 123-153.

2. Chamberlin, D. D., Hasselmeier, H. F., Luniewski, A. W., Paris, D. P., Wade, B. W., & Zolliker, M. L. Quill: An Extensible System for Editing Documents of Mixed Type. Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, Vol II, The Computer Society of the IEEE, 1988.

3. Dijkstra, E. W.. *Notes on Structured Programming*. Academic Press, London, 1972.

4. Furuta, R. An Integrated, but not Exact-Representation, Editor/Formatter. In J. C. van Vliet, Ed., *Text processing and document manipulation*, Cambridge University Press, Cambridge, England, 1986.

5. Kernighan, B. W. & Plauger, P. J.. *The Elements of Programming Style*. McGraw-Hill, New York, 1978.

6. Quint, V., Vatton, I. Grif: An Interactive System for Structured Document Manipulation. In J. C. van Vliet, Ed., *Text processing and document manipulation*, Cambridge University Press, Cambridge, England, 1986.

7. Sandewall, E. "Programming in an Interactive Environment: The LISP Experience". *ACM Computing Surveys 10*, 1 (1978), 35-71.

8. Walker, J. H. Symbolics Sage: A Documentation Support System. Intellectual Leverage: The Driving Technologies, IEEE Spring Compcon84, San Francisco, 1984, pp. 478-483.

9. Walker, J. H. Document Examiner: Delivery Interface for Hypertext Documents. Proceedings of the Hypertext '87 Workshop, Chapel Hill, N. C., November 1987.

10. Walker, J. H. "Supporting Document Development with Concordia". *IEEE Computer 21*, 1 (1988), 48-59.

11. Walker, J. H. & Bryan, R. L. An Editor for Structured Technical Documents. In J. J. H. Miller, Ed., *Protext IV: Proceedings of the 4th International Conference on Text Processing Systems*, Boole Press Limited, Dublin, Ireland, 1987, pp. 145-150.

12. Walker, J. H., Moon, D. A., Weinreb, D. L. & McMahon, M. "Symbolics Genera Programming Environment". *IEEE Software 4*, 6 (1987), 36-45.