Reading Source Code

Darrell R. Raymond

Abstract

Source code is, among other things, a text to be read. In this paper I argue that reading source code is a key activity in software maintenance, and that we can profitably apply experiences and reading systems from text databases to the problem of reading source code. Three prototype systems are presented, and the main features of their design are discussed.

1. Introduction.

Most programmers and project managers have at one time or another maintained software that they did not write. In the classic scenario, one suddenly acquires responsibility for a program consisting of thousands of lines of code. The program serves a wide user community, who complain bitterly if the code is not kept running in a variety of hardware and software environments. The program is highly complex, somewhat flaky, has little or no documentation, and its authors have vanished or otherwise disclaimed responsibility. The user community tries to help by generating a constant stream of bug reports and suggestions for new features and options. While many people claim to know what the program is supposed to do, very few seem to know how the program does it. Other responsibilities cannot be ignored, so rewriting the code from scratch is not an option.

In order to maintain code, one must first understand it. Despite nearly universal acceptance of the importance of design documents, the source code is often the only statement of the program's design. In time-honoured fashion, one might simply dump the code to the line printer and struggle through the print-out for several days. This method has the virtues that the printout can be spread around the room and annotated, but otherwise is not very appealing. Numerous enhancements to this basic approach have been suggested; for instance, the appearance of the printout can improved with pretty-printers[6, 14] or typesetting filters.[13, 18] Workstations might be used for online reading and searching with pattern matching tools such as *grep*. Finally, one might be lucky enough to have access to software that automatically constructs indexes to the code.[16]

Whatever approach is used, it is clear that a central activity in software maintenance is *reading*. In maintenance, the main role of source code is not as a compilable entity, but as a human-readable statement of the intent and mechanism of the program. The source describes the program's algorithms and data structures, naturally, but it also contains many clues about the evolution of the program. The choice of variables, the use of comments, the indentation style, and other uncompilable aspects of the code are hints about the software engineering process.[33] A lack of comments and tangled control flow, for instance, suggests that the program is poorly structured and hence potentially error-prone. Mysterious constants suggest hard limits on dynamic structures and problems for users who exercise the code strenuously. Lack of preprocessor statements may indicate inadequate planning for portability. Important clues can also be extracted from design documents, if they exist, since they are themselves a kind of declarative source code.

IBM contact for this paper is Arthur Ryman, Centre for Advanced Studies, IBM Canada Ltd., Dept. 81/894, 895 Don Mills Road, North York, Ontario M3C 1W3.
This paper is available as IBM Canada Laboratory Technical Report TR 74.070.

Two barriers to effective reading are size and complexity, both of which tend to hide the structure of a program.[10] Size means that the code cannot be held in one's mind as a single entity; one must find a way to break it into smaller, more comprehensible pieces. If the program is properly modularized this task is not difficult, but if the program is a single monolithic routine, then one must develop a conceptual modularization that takes the place of the missing decomposition. Complexity implies interconnectedness, and hence effective comprehension of one fragment typically necessitates reading several other fragments. It is often important to know what *not* to read; in other words, one must choose which parts of the program will remain a black box. Borenstein commented on the problems of maintaining an electronic billboard program:

> The author [Borenstein] maintained the TOPS-20 system for about two years, ending in late 1982. I still have no idea how a large chunk of that program works, nor does the previous maintainer. A few individuals in the community know more but are generally reluctant to admit it for fear they'll have to help fix something.[7]

Size and complexity problems are also encountered when reading large text databases. The *Oxford English Dictionary*, for instance, is some 600 Mbytes in size, and has a dense and sometimes irregular structure.[19, 29] People who want to find information in the *OED* can afford to read only a tiny fraction of the whole text; yet its complexity means that relevant information is often dispersed in various places about the text. The users of the dictionary, like programmers, are faced with the problem of deciding which parts of the document to read. The *OED*'s lexicographers also face problems analogous to those involved in software maintenance; for example, they must check to see if all the citations for a given work are consistent, and ensure that all cross references have a valid target.[31]

Even small texts can be surprisingly difficult to read if time is limited. In an experiment involving online searching of Arthur Conan Doyle's *Hound of the Baskervilles*, for example, we found that subjects had difficulty with problems such as *who murdered whom?* and *see how much you can find out about Mr. Stapleton's physical features*. An interesting result of the experiment was that the subjects who solved the problems most effectively were those who read more of the text, not those who used the query mechanism more extensively.[20]

Our experiences with reading large and complex online texts are part of a general research program on advanced text database systems. One goal of the program is to develop a set of tools for searching, browsing, and displaying both text and the structures imposed on that text. Frank Tompa and I have been investigating the application of these tools to the problem of reading source code. If they prove useful, the prototypes that are described in the next section may become part of 4Thought, a software engineering project at the Centre for Advanced Studies at the IBM Canada Laboratory.[25]

2. Three prototypes.

The prototypes described in this section are a first attempt to apply text database expertise to the problem of reading code and documentation. The capabilities we wish to provide are present only in a rudimentary fashion, and we expect substantial refinement as development continues.

The first prototype demonstrates the advantages of effective typesetting and multiple views for reading source code. Documenters frequently suggest that documentation is very similar to programs, and hence should benefit from software tools;[30] here we explore the other side of this analogy, arguing that source code can benefit from document formatting and typesetting technology. Figure 1 shows several views of two files derived from IBM's ImagEdit ® product.[26] The *Contents* window shows the source in a style known as *C Grind*, which uses a variety of fonts and other typesetting capabilities to improve readability. The other windows display alternative styles which selectively emphasize various aspects of the code. The *Nesting (Reduced)* style, for example, uses typography more aggressively, displaying assignment statements and other non-control flow statements in a very tiny *nil* font. The result is that the control flow statements, which generally define the main blocks of the code, are more prominent. The use of *nil* also allows the *Nesting (Reduced)* format to display more code in a given area; here

the *Nesting (Reduced)* window takes up less than half the space of the *Contents* window, but covers three times the span of code. The other windows show the function definitions, the function calls, the include files, and the names of the files. Each of these views is generated dynamically from the source code files at display time.

The second prototype demonstrates the utility of navigating a document's explicit underlying structure. This sort of browsing is sometimes called *hypertext*, because it involves a non-linear traversal of the parts of the document via means of explicit links.

The system shown in Figure 2 is used to browse the ImagEdit design document. This document describes the data structures, procedures, and the various modules of ImagEdit in such a way that a programming team can develop the source code for the ImagEdit product; it also serves an important purpose in software maintenance, since it is the official statement of the program's proper design. In addition to the overt text, the online version of this document also contains embedded Prolog rules which express a variety of relationships between fragments of the document, such as necessary header files and callable or called functions.

In the figure, embedded rules are displayed shown in double square brackets. The prototype application allows the programmer to find and display the target of any given rule simply by pointing at it. The window at the top left is a table of contents used to jump to a section of the document. The programmer can page back and forth in the table of contents window, and then jump directly to the desired part of the text simply by pointing at the corresponding entry in the table of contents. This part of the document will be shown in the bottom lefthand window. The programmer can read the document in this window, or can choose to follow one of the Prolog links. This is done by pointing to the link; the software then consults the database to find the target of the link, and brings up the relevant part of the document in the bottom righthand window. A history of the document fragments that have been viewed is shown in the slender window in the upper right. Figure 2 shows the state which results after following the links *AppResizeChildren* to *ImgResizeWindow* to *ToolBoxDestroyWindow* to *ViewGetScreenResolution* to *MapProduct*.

The third prototype explores the advantages of general full text searching for navigating by means of implicit or dynamically determined structure. Here the ‘‘links’’ between parts of the document are determined implicitly, by the existence of an appropriate text string.

Figure 3 shows a browser for the ImagEdit design document that permits full text searching. The leftmost window shows the list of packages; by pointing into this view the programmer can jump to a given package, which is shown in the *Full Text* window. While reading the text of the document, the programmer may encounter a new concept, function, data structure, or other object of interest. The string corresponding to this object can be selected by using the mouse; this action generates a database query for the string. The set of hits to the query is displayed in the *Search Results* window, one per line, showing a small amount of the surrounding context. The programmer chooses to view the whole context of any hit by pointing at it, which brings up the relevant part of document in the *Target* window. In the figure, the programmer has come across the term ‘‘image pipe’’, and instituted a full text search for occurrences of that term in order to find its definition. A likely instance of the term was selected from the *Search Results* window and the resulting package shows up in the *Target* window. The filename of that package is displayed in the title bar of the *Target* window.

In each prototype, all windows permit paging through the document or code, as well as jumping to the start or end of the text. All windows can be resized and repositioned according to the programmer's preferences, and can display either multiple or single columns of text.

3. Enhancing readability.

How do the prototypes improve our ability to read source code? The most obvious contribution is the use of enhanced typography. Despite work on typographical presentation of programs,[2, 3] programmers traditionally eschew complex typography, partly because of a lack of tools for viewing typeset programs, but probably also because of an ingrained belief that typesetting is an unnecessary frill. Nevertheless, most programmers are quite adamant about their preferences for layout-related issues, such as indenting policies and the appropriate use of variable names.[11, 17]

Reading is both a physical and a mental activity. The physical effort of reading is alleviated by choosing fonts and spacing that reduce the eye's scanning effort. Substantial work has been done on font design and layout of alphabetic characters, even to consideration of the frequency of the wave corresponding to the vertical bars of the letters.[24] Our prototypes do not operate at this level of refinement. As a first step towards more readable text, however, the use of proportional fonts is generally considered to be a significant improvement on constant width fonts.[4] Proportional fonts also permit more code to be displayed in a given area, on average, because the thinner letters take up less space.

A second aspect of the physical effort of reading is the cost of mechanical motions needed to get to the next fragment of text. Each page of text that has to be read is one more page that has to be manually called up by means of a mouse or key operation, and unless the physical effort is minimized, this quickly becomes a tedious and error-prone activity. The prototypes address the manual effort of reading in several ways. Paging back and forth in the windows requires only a single keystroke or button press, without the precise positioning often needed to operate scroll bar gadgets. The use of proportional fonts increases the amount of text that is displayed and hence helps to reduce the frequency of paging. Paging is also reduced by effective text suppression, which enables large distances to be covered with a single manual action. The ability to navigate implicit and explicit links simply by pointing at them also significantly reduces the effort of finding related fragments, compared to manual scanning of the text. Finally, flexible resizable windows permit the volume of displayed text to be adjusted as desired.

A third aspect of the physical effort of reading is the time delay involved in presenting text. Most desktop publishing systems and batch typesetters can display text very well, but few do it very quickly. When one is merely previewing a document that will be printed offline, a small delay is acceptable because it is still shorter than the time for hardcopy generation. When source code is being read, however, speed is of the essence, since the programmer is attempting to grasp a large amount of detail, and presentation delays will exacerbate short term memory loss. There is another aspect to psychology of speed: programmers are impatient creatures, and will quickly revert to the interface that offers the highest speed, taking on themselves the extra burden of poor display. In order to encourage programmers to use typesetting systems to view code, the speed of the system must be no less than that of their favorite editor. The update of the text display in our prototypes is virtually instantaneous on the IBM RS/6000, even when suppressing large parts of the text.

The mental effort of reading is naturally dominated by the difficulty of the concepts involved in the text. A tangled piece of spaghetti code is not made easier by presenting it with proportional fonts and consistent indenting. Where source code can benefit from techniques to reduce mental effort, however, we should use them. Typography, for instance, can help to convey the structure of the code: typefaces and sizes can be used to make function headers stand out, or, as in the case of the first prototype, to emphasize the control flow statements. Text suppression not only reduces the effort involved in paging through the code, but also the mental effort of connecting up related sections. It also reduces the mental effort required in reading unnecessary code. Unlike physical effort, however, techniques for reducing mental effort are not well understood, because there is little solid theory on how layout should be employed to express semantics.[22] This is a key area for further research.

Generally speaking, efficient reading of source code results from adhering to a principle of locality. By keeping relevant sections of the code close together, both physically and conceptually, we make it easy for the reader to comprehend the text. The prototypes exhibit three basic techniques for increasing locality. Code suppression clusters related information that is separated by long extents of unrelated code. Effective use of typographic techniques such as font changes or indenting allows one to emphasize relationships by means of the weight or size of the characters or the relative positions on the screen. The use of multiple windows and searching tools for following implicit and explicit links permits several widely-spaced pages of code to be placed in close proximity.

In the next sections we discuss the two basic design elements of the prototypes: descriptive markup, and multiple process structuring.

4. Descriptive markup.

Online text, like most data, must be structured to be of use. Text structure is normally represented in one of two ways: either the text conforms to a grammar and hence a parse tree can be derived, or else the structure is explicitly embedded by the use of markup codes or tags. Source code typically conforms to a grammar, and software engineering tools usually exploit this grammar for editing and display. Textual databases, on the other hand, often deal with texts which do not conform to a grammar, and hence rely on markup. Our prototypes are based on the use of explicit markup.

Markup is divided into two classes, procedural and descriptive. Procedural markup employs tags that map directly to some typesetting operation; the *troff* tag `\fI`, for example, invokes a procedure that switches to italic font, and the tag `\fB` invokes a procedure that switches to bold font. Descriptive markup employs tags which map to a data model or other abstract structure imposed on the text; thus a descriptive system might use the tag `<emphasis>` to indicate that a text fragment is to be emphasized; the actual typographical effect is not defined by the markup itself, but done elsewhere. Standard Generalized Markup Language or SGML is the most publicized approach to descriptive markup. [1]

Systems based on descriptive markup typically support the use of *style sheets*, which are symbol tables that map instances of tags to typesetting operations. A collection of such mappings constitutes a style or a single view of a text. The views shown in Figures 1-3 are not obtained from separate files, but instead are derived from a single code and documentation file; different styles employ different mappings between tags and typographic effect.

Descriptive markup has two main advantages. By decoupling the identification of text structures from the operations that are associated with those structures, we can change those operations in a consistent manner, without the need to edit the text. Emphasized text, for example, can be underlined, highlighted, shown in bold, or whatever is desired, simply by changing the style sheet. More importantly, descriptive markup tends to avoid the kinds of ambiguity common in procedural systems; where *troff* has no way to distinguish between italicized fragments, a descriptive markup system will tend to tag structures according to a criterion of use or meaning, thus distinguishing between the mathematical, bibliographic, and purely emphatic uses of italics.

Descriptive markup is gaining favor among documenters, and SGML may soon be a required standard for documentation. Source code, on the other hand, will probably never be marked up descriptively. In order to gain the advantages of descriptive markup for source code, we must preprocess it to add descriptive tags. A transduction preprocessor was built for the code shown in Figure 1; it marked up line breaks, reserved words, braces, comments, strings, and other syntactically simple features of the code with descriptive tags, which can then be converted to typesetting operations by our formatter.

The use of descriptive markup for reading source code highlights an important design tradeoff — the separation of the software system from the syntax of the programming language. Most tools for dealing with source code take advantage of the syntax of the language; they are based on the parse tree model of the text. This couples the tool to the programming language. By using embedded markup, we introduce a level of indirection between the structures in the code and the recognition of those structures, a tactic which has several advantages. First, we gain all the capabilities of multiple views and the use of views as indexes for our software tools with no extra effort. Second, we can more easily use our software to handle multiple languages; we need merely build a preprocessor that installs the markup, and the remainder of the system will operate as before. The expense of constructing style sheets and applications can thus be amortized across several languages. Third, embedded markup need not be restricted to the syntax tree of the language; we can embed whatever structure we desire, including arbitrary semantic structures (though in this case we will require more complicated preprocessing of the code). Finally, since we are not confined to a syntax tree, it is easier to deal with erroneous and partial code fragments. This is in contrast to some syntax-directed systems, which are not able to edit syntactically invalid program fragments.

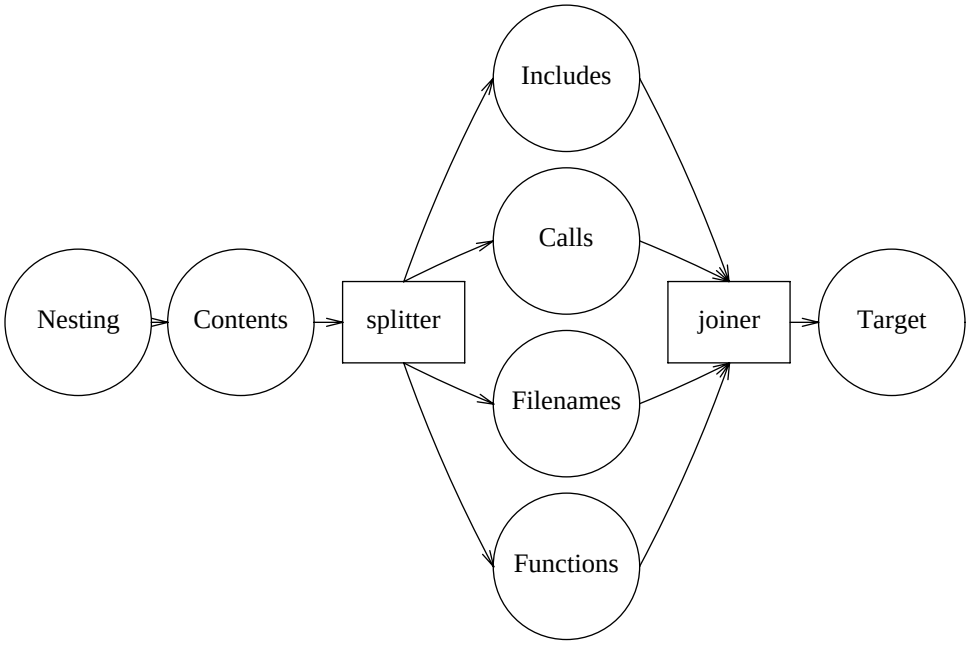

Figure 4. Process structure of code browser.

5. Process structuring.

Many text database and CASE systems are built as strongly coupled, monolithic applications. Such an approach makes their systems difficult to reconfigure in ways the developer did not anticipate. The monolithic approach also involves questionable psychology, given that programmers are notorious for wanting to customize and change their computing environment.

We have avoided strong coupling by decomposing our applications into multiple processes that exchange messages. Multiple processes are not an essential technique for systems for reading source code, but they are a form of decomposition that has many advantages. Processes share only the messages that they exchange, making it simple to swap modules during development and testing. Loose coupling makes it possible to do some run-time configuring of applications, enabling end-users to modify existing applications to their own tastes. Another advantage of process decomposition is that it focuses attention on interfaces rather than on programs. If a collection of generally useful processes is available, programmers will tend to build their applications to fit the interfaces rather than to incorporate the mechanism in their own code. Unix ® filters and pipelines are the best example of this type of reusability.[15] Process-based applications are also easily distributed: each process can be placed on a separate processor without the need to change or recompile an application. Finally, a process-based application is useful when a project is being developed by more than one group of programmers, because the natural unit of decomposition is a standalone program. A recent example of the value of loose coupling was the task of connecting our applications to the GraphLog project at the University of Toronto.[27] Despite the fact that the two systems are compiled in different programming environments (SmallTalk and C) and are currently running on different hardware (Unix workstation, Apple Macintoshes, and industry standard PCs), interconnection was not difficult.

A process-based methodology requires a sufficiently large set of of processes. As we gain more experience with building process-based applications, our toolkit is growing. The first and to date most important tool is our text formatter, which is used to display all the text shown in Figures 1-3. [21] This tool accepts two types of input, text and offsets.

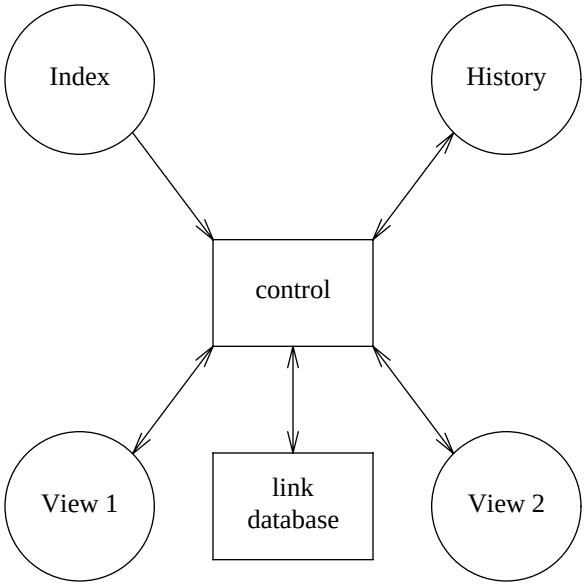

Figure 5. Process structure of document browser.

If a text is received, it replaces whatever text is currently displayed; if an offset is received, the display is updated to that position in the current text. The formatter can also produce text or an offset as output, based on a user's selection from the window. Since the formatter accepts its own output as input, it easy to string together multiple invocations of the formatter.

A second component of the process toolkit is a shell for constructing non-linear process structures. Unix shells such as *sh* and *csh* are primarily intended to support process pipelines, and many non-sequential configurations are difficult or impossible to construct with these shells. As a result, several researchers have constructed shells that support non-linear process structuring, often with some sort of visual editor for setting up the process graph.[5, 8, 12, 23] Our shell is a more modest effort which takes its input from a specification file. The specification describes a set of processes, their command line options, and a list of interconnections; the shell then sets up pipes for the interconnections and spawns the processes, exiting when the process structure has been created. Our shell does little error checking, is not particularly robust, and does not permit shell programming; we expect to address all these problems shortly.

Our process toolkit also involves databases of two types. The first is a full text database used for string-based queries. This database employs a text searching program that facilitates searching for strings of any length. A preconstructed index is used that supports boolean queries, proximity searches, and searches within restricted regions of a text. The second type of database is a simple flat file database that manages n-ary relationships, including the embedded Prolog facts, the information stored in the traversal history list (Figure 2) and the search result list (Figure 3). Currently we use one-shot applications to store this data, but a small and efficient relational system is clearly a more general solution.

Complex process structures often involve making copies of an output stream or merging multiple input streams.[23] We have generalized these activities in processes to split and join standard Unix I/O. Each process is a simple buffer manager that duplicates its input to its output stream, either duplicating to multiple output streams or merging multiple input streams. It may seem exorbitant to allocate a process for stream buffering, but the cost is typically nominal. Moreover, managing buffers and pipes is somewhat tricky, so it is desirable to centralize this activity. The splitter and joiner processes are used in the prototype shown in Figure 1.

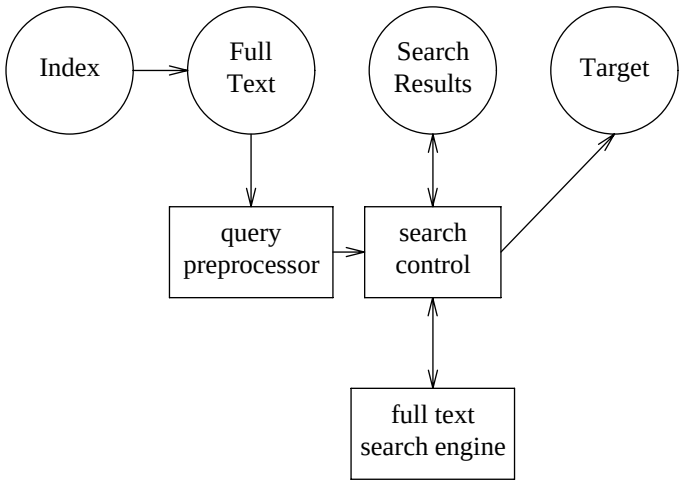

Figure 6. Process structure of string search browser.

In addition to the processes in the toolkit, it is usually necessary to construct one or more one-shot processes that massage the output of one program to fit the input of another, or that buffer information between processes and attach some functionality to that information. These processes are usually quite small, on the order of a few hundred lines of code; often they are simply invocations of existing Unix filters such as *awk* or *sed*.

Figures 4 through 6 show the process structure diagrams of the prototype applications presented in Figures 1-3, respectively. Circles in the process structure diagrams correspond to visible text formatter windows; rectangles in the process structure diagram correspond to invisible filters, control programs, and databases. Arcs in the process structure diagrams indicate flow of data.

Figure 4 is the process structure diagram of the code viewer prototype presented in Figure 1. The *Nesting (Reduced)* and *Contents* view are connected, so that a selection in the *Nesting (Reduced)* view results in updating the *Contents* view. Selections in the *Contents* view are sent in parallel to the four views *Includes*, *Calls*, *Filenames*, and *Functions*. Each of these views is updated when a selection is made in the *Contents* view. Selections in any of the four subsidiary views are then used to update the *Target* view. This structure demonstrates how splitter and joiner processes can be used to manage multiple streams.

Figure 5 is the process structure diagram of the document browser presented in Figure 2. Here a single (invisible) control process handles most of the work. The *Index* view is used to navigate to a particular part of the text; when the programmer selects a line in the view, the corresponding offset is sent to the control process. The control process begins a new history at this point, clearing the *History* view, and sends *View 1* the chosen offset. Selections of Prolog links made in *View 1* or *View 2* are sent to the control process, which queries the link database for the target. The current context is then sent to the *History* view, and the target is sent to the alternate *View*. Selections from the *History* view cause *View 1* to be positioned to that section and *View 2* to be cleared.

Figure 6 shows the process structure for the document browser presented in Figure 3. The *Packages* view sends an offset to the *Full Text* view. The programmer can submit a full text query from the *Full Text* view by selecting a fragment of text. This fragment is passed on to the query preprocessor, which surrounds the text with the necessary query syntax and then submits it to the main search process. The search routine notes the query and then passes it on to the full text search engine. The results of the query are a set of offsets and short text samples, one for each instance of a solution to the query. The offsets and samples are buffered by the main search process, which tags the text samples and passes them to the *Search Results* window for viewing.

When the programmer selects one of the text samples in the *Search Results* window, the offset of the sample is passed back to the search process, which determines the corresponding text offset and forwards this to the *Target* view.

6. Discussion.

The application of text database technology to the problem of reading source code is still in its early stages, and several directions for further work are being considered.

One important direction is empirical studies of programmers engaged in software maintenance. Without observing programmers actually attempting to use these tools, we will be able to conclude little about their effectiveness. Such studies will no doubt identify many aspects of the prototypes that need improvement. It is just as important, however, that we investigate the process of reading source code (and its relationship to the software maintenance problem) as a phenomenon independent of our specific systems. Learning more about the reading activities of programmers could very likely suggest different approaches than the ones described here.

An important limitation of the prototypes is their inability to handle dynamically changing situations. Updates to the source code itself, for example, are not reflected in appropriately updated views. Nor can the views be altered to display dynamically selected objects, as for instance might be useful in highlighting all occurrences of a dynamically chosen variable name. These limitations are partly a result of the history of development of our tools; the *OED* and other texts we have investigated are largely static, unlike source code. A more fundamental difficulty, however, is that markup (descriptive or otherwise) by its very nature involves the implicit assumption that all structures are identified by explicit tags in the text, and thus to add structure means to insert tags. This constraint seems impractical for large texts.

If descriptive markup is inadequate, for dynamic texts, should we reconsider the use of parse tree (that is, internal, implicit) structures? This seems necessary, but in so doing, we must avoid some of the errors made in developing syntax and structure-driven environments. These systems have been extensively studied, but have not made a significant impact in programming workplaces. Part of the reason is that such editors are often irritatingly restrictive, permitting only syntactically valid programs and program fragments. Another problem is that a syntactic parse tree approach is not the best way to edit all objects. Arithmetic expressions, for instance, are almost always entered as infix expressions.[28] Furthermore, syntactic structure is not the only (or even the most important) structure in a program, and programmers generally have more difficulty getting the semantic structures correct than the syntactic ones.[32] Syntax-directed approaches tend to strongly couple the parsing, display, editing and compilation of programs through the parse tree, while the descriptive markup approach has led us towards a loosely coupled multiple process structure approach. Some researchers have concluded that syntax-driven editors are best reserved for application-specific languages, which are typically used infrequently, and so are prone to suffer the kinds of syntax errors that syntax-driven environments can address.[9]

Clearly, a rapprochment between the two approaches is in order. The use of descriptive markup has helped us to discover certain useful techniques and design principles, but it is not itself an essential part of those principles. It is possible to retain a loosely coupled architecture, to emphasize the identification of document structure rather than typography, and to maintain a clean separation between language syntax and system functionality, without the explicit use of descriptive markup at every level of the software. Developing systems that fulfill this potential is the next step in our agenda.

7. Acknowledgements.

Frank Wm. Tompa was extensively involved in developing the ideas and software described in this report. My thanks to Arthur Ryman and the IBM Canada Laboratory for the opportunity to work on the 4Thought project, as well as for financial support under grant DC01357. Financial support was also received from the Natural Sciences and Engineering Research Council of Canada under grant OGP0009292.

8. References.

1. ISO, ‘‘Information processing — text and office systems — Standard Generalized Markup Language (SGML),’’ ISO 8879-1986, International Organization for Standardization (1986).
2. Ronald Baecker and Aaron Marcus, ‘‘Design Principles for the Enhanced Presentation of Computer Program Source Text,’’ *Proceedings of the CHI ’86 Conference on Human Factors in Computing Systems*, pp. 51-58 (April 13-17, 1986).
3. Ronald M. Baecker and Aaron Marcus, *Human Factors and Typography for More Readable Programs*, Addison-Wesley, Reading, Massachusetts (1990).
4. Ion P. Beldie, Siegmund Pastoor, and Elmar Schwarz, ‘‘Fixed Versus Variable Letter Width for Televised Text,’’ *Human Factors*, **25**(3) pp. 273-277 The Human Factors Society, Inc., (1983).
5. Mitali Bhattacharyya, David Cohrs, and Barton Miller, ‘‘A Visual Process Connector for Unix,’’ *IEEE Software*, pp. 43-50 (July 1988).
6. G. Blaschek and J. Sametinger, ‘‘User-Adaptable Prettyprinting,’’ *Software — Practice and Experience*, **19**(7) pp. 687-702 (July 1989).
7. Nathaniel S. Borenstein, ‘‘A House of Cards: A History of the Inorganic Evolution of the CMU Bboard System Software,’’ CMU-CS-85-152, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania (1985).
8. Kjell Borg, ‘‘Ishell: A Visual Unix Shell,’’ *Proceedings of the CHI ’90 Conference on Human Factors in Computing Systems*, pp. 201-207 (April 1-5, 1990).
9. B.A. Bottos and C.M.R. Kintala, ‘‘Generation of Syntax-Directed Editors With Text-Oriented Features,’’ *The Bell System Technical Journal*, **62**(10) pp. 3205-3224 (December 1983).
10. Fredrick P. Brooks, Jr., ‘‘No Silver Bullet: Essence and Accidents of Software Engineering,’’ *IEEE Computer*, pp. 10-19 (April 1987).
11. L.W. Cannon, R.A. Elliott, L.W. Kirchhoff, J.H. Miller, J.M. Milner, R.W. Mitze, E.P. Schan, N.O. Whittington, Henry Spencer, David Keppel, and Mark Brader, ‘‘Recommended C Style and Coding Standards: Revision 6.0,’’ Specialized Systems Consultants, Inc., Seattle, Washington (June 25, 1990).
12. Roger Davis, ‘‘Iconic Interface Processing in a Scientific Environment,’’ *SunExpert Magazine*, pp. 80-86 (June 1990).
13. Van Jacobson, ‘‘tgrind(1),’’ *Unix User’s Manual*, (June 1989).
14. Matti O. Jokinen, ‘‘A Language-Independent Prettyprinter,’’ *Software — Practice and Experience*, **19**(9) pp. 839-856 (September 1989).
15. Brian W. Kernighan, ‘‘The Unix System and Software Reusability,’’ *Proceedings of the Workshop on Reusability in Programming*, pp. 235-239 (September 7-9, 1983).
16. Paul W. Oman and Curtis R. Cook, ‘‘The Book Paradigm for Improved Maintenance,’’ *IEEE Software*, pp. 39-45 (January 1990).
17. Rob Pike, ‘‘Notes on Programming in C,’’ unpublished technical report, AT & T Bell Laboratories (1990).
18. Dave Presotto and William Joy, ‘‘vgrind(1),’’ *Unix User’s Manual*, (August 3, 1983).
19. Darrell R. Raymond and Frank Wm. Tompa, ‘‘Hypertext and the Oxford English Dictionary,’’ *Communications of the ACM*, **31**(7) pp. 871-879 (July 1988).
20. Darrell R. Raymond and Heather J. Fawcett, ‘‘Playing Detective with Full Text Searching Software,’’ *SIGDOC ’90*, **14**(4) pp. 157-166 (October 31-November 2, 1990).
21. Darrell R. Raymond, ‘‘*lector* — An Interactive Formatter for Tagged Text,’’ OED-90-02, Centre for the New Oxford English Dictionary, University of Waterloo, Waterloo, Ontario (1990).
22. Darrell R. Raymond, ‘‘Characterizing Visual Languages,’’ *IEEE Workshop on Visual Languages*, (October 8-11, 1991).

23. Lawrence A. Rowe, Peter Danzig, and Wilson Choi, ‘‘A Visual Shell Interface to a Database,’’ UCB/ERL M87/2, Electronics Research Laboratory, University of California, Berkeley, Berkeley, California (January 13, 1987).

24. Richard Rubinstein, *Digital Typography: An Introduction to Type and Composition for Computer System Design*, Addison-Wesley, Reading, Massachusetts (1988).

25. Arthur Ryman, ‘‘The Theory-Model Paradigm in Software Design,’’ TR 74.048, IBM Canada Laboratory, Toronto, Ontario (October 11, 1989).

26. Arthur Ryman, *ImagEdit V2.0 Design Notebook, Revision P.1*, IBM Canada Ltd., Markham, Ontario (July 1990).

27. Arthur Ryman, Alberto Mendelzon, and Mariano Consens, ‘‘Software Engineering Queries Using GraphLog,’’ IBM TR 74.053, Centre for Advanced Studies, IBM Canada Laboratory, North York, Ontario (June 28, 1991).

28. Uri Shani, ‘‘Should Program Editors Not Abandon Text Oriented Commands?,’’ *SIGPLAN Notices*, **18**(1) pp. 35-41 (January 1983).

29. Frank Wm. Tompa and Darrell R. Raymond, ‘‘Database Design for a Dynamic Dictionary,’’ in *Research in Humanities Computing I, Papers from the 1989 ACH-ALLC Conference*, ed. Ian Lancashire, Oxford University Press (September 1991).

30. Janet H. Walker, ‘‘Supporting Document Development with Concordia,’’ *IEEE Computer*, **21**(1) pp. 48-59 (January 1988).

31. Yvonne L. Warburton and Darrell R. Raymond, ‘‘Resolving Cross-References,’’ unpublished technical report, Centre for the New Oxford English Dictionary, University of Waterloo (March 1989).

32. Richard C. Waters, ‘‘Program Editors Should Not Abandon Text Oriented Commands,’’ *SIGPLAN Notices*, **17**(7) pp. 39-46 (July 1982).

33. Mark Weiser, ‘‘Source Code,’’ *IEEE Computer*, **20**(11) pp. 66-73 (November 1987).

About the author

Darrell R. Raymond is a doctoral student in the Department of Computer Science at the University of Waterloo. He holds an M.Math from the University of Waterloo, where he spent six years as a researcher working on videotex systems and the project to computerize the Oxford English Dictionary. His current research area is data modelling for non-traditional database systems.

Raymond can be reached electronically at drraymond@daisy.uwaterloo.ca.

Figure 1. Code browser.

Figure 2. Document browser.

Figure 3. String search browser.