

OREGON STATE

UNIVERSITY

COMPUTER

SCIENCE

DEPARTMENT

REVERSE ENGINEERING CODE LISTINGS INTO "BOOKS" TO  
IMPROVE MAINTAINABILITY

Paul W. Oman  
Computer Science Department  
College of Engineering  
University of Idaho  
Moscow, Idaho 83843

Curtis R. Cook  
Computer Science Department  
Oregon State University  
Corvallis, Oregon 97331-3902

89-60-1

**REVERSE ENGINEERING CODE LISTINGS INTO "BOOKS" TO  
IMPROVE MAINTAINABILITY**

Paul W. Oman  
Computer Science Department  
College of Engineering  
University of Idaho  
Moscow, Idaho 83843  
(208) 885-6589

Curtis R. Cook  
Computer Science Department  
Oregon State University  
Corvallis, Oregon 97331-3902  
(503) 754-3273

**Abstract**

We have identified a "book paradigm" for source code formatting which improves program comprehension and assists in maintenance work. The book paradigm can be implemented by reverse engineering code listings into a "book" with preface, tables of contents, chapters, sections, indices and pagination. This reverse engineering effectively reorganizes source code listings into a more usable form of information.

Our empirical tests with the book format show that a significant improvement in program comprehension and corresponding reduction in maintenance effort can be achieved by this process. These results have a direct impact on programming standards, automated style analyzers, and code formatting tools like pretty-printers and syntax directed editors.

CR Categories and Subject Descriptors: D.2.3 [Coding], D.2.2 [Tools and Techniques]

KEYWORDS: Programming style, coding style, code formatting, programming tools, coding tools, reverse engineering.

REVERSE ENGINEERING CODE LISTINGS INTO "BOOKS" TO  
IMPROVE MAINTAINABILITY

**Toward better programming style.**

In Edward Yourdon's book, Techniques of Program Structure and Design, he lists seven major problems facing maintenance programmers [Your75]. Number six is:

"A very basic problem is that most people have great difficulty understanding other people's code. Perhaps this is because most programmers seem to evolve their own personal programming style; a larger part of the problem, though, is that many programmers write their code in a relatively disorganized style."

Why do we have this disorganization in programming styles? Mostly because our code formatting tools and techniques are designed to produce "pretty" code, with little thought as to whether it really aids programmer comprehension. What we need is a method of formatting programs consistent with programmer comprehension strategies and maintenance activity.

We have identified a "book paradigm" of program formatting, which we believe is the most appropriate typographic organization of source code documents. The book paradigm incorporates the use of statement sentences, paragraphing, sectional division, chapter division, prefaces, indexing and pagination. By organizing source code in this manner, consistent with other forms of information, programmer comprehension is improved, thus facilitating maintenance activity.

In this paper we review the known characteristics of programmer comprehension and maintenance behavior and then explain how reverse engineering program listings into "books" aids those activities. Formatting source code like a book is an implementation technique that incorporates principles of typographic style compatible with most comprehension strategies and approaches to maintenance. We present two experiments showing the benefits of the book paradigm in maintenance situations.

Our book model is a familiar and easily understood paradigm. It should not be confused with Knuth's style of "literate programming," which calls for a change in the programming process [Bent86]. The only similarity between our book paradigm and Knuth's method is that the end-products both have a table of contents and an index. We are building a book-like document from source code, not changing the whole programming process.

## **When are hard-copy listings used?**

With the advent of large screen, multiwindowed workstations, we were concerned that hard-copy listings may no longer be used in day-to-day maintenance programming situations. We asked several professional programmers when (and if) they used hard-copy source code listings. All respondents indicated that hard-copy listings are still used in situations where on-line code reading was awkward or inadequate. They said hard-copy listings were used:

- o When making a lot of changes and I need to see the big picture.
- o When the construct I'm looking at does not fit on the screen.
- o When I'm trying to understand a large, new program.
- o To have the header files handy for cross referencing.
- o When changes are so interconnected that I need to make notes about them.
- o For really long deeply nested conditionals and while loops.
- o When explaining code to other people; for instance, code walk-throughs.
- o When starting on a program I've never seen before, paper gives me a much bigger context.
- o When I want to study code away from the office.
- o When trying to understand a poorly-designed program.

All programmers indicated hard-copy source listings were necessary for large complicated systems and when scrolling, multiwindow-viewing environments were not available. It's interesting to note that all of our respondents had large-screen, multiwindow workstations in their offices and yet they still resorted to hard-copy listings.

## **Programmer comprehension and program maintenance.**

Exactly how programmers read and understand code listings is not well understood. However, empirical studies with programmers show that "chunks," "plans," and "beacons" play an important role in this process. Studies on how expert programmers remember code show they "chunk" code into meaningful program segments and then mentally organize the chunks based on the functional purpose of the code [Adel81]. Mental "plans" or "schemata" guide the organization and processing of this information [Adel84, Solo84].

"Beacons" are easily recognized code structures (or tokens) that programmers use to locate and isolate meaningful chunks of code while formulating and/or verifying their mental plans [Broo83]. Virtually all research in programmer comprehension supports the existence of chunks, plans, and beacons.

Comprehension plays a crucial role in program maintenance. It's estimated that maintenance programmers spend between 47 and 62 percent of their time trying to comprehend code [Pari83]. Depending upon the task at hand, this effort may require overall program comprehension, focused on-the-spot detailed knowledge, or just browsing behavior. Maintenance is usually broken down into three types: corrective, adaptive, and perfective. Corrective, or repair maintenance, is the best understood of the three, although it accounts for only 20 to 25 percent of the total maintenance effort [Bend87]. Adaptive and perfective maintenance, on the other hand, account for 75 to 80 percent of the total maintenance effort, but little is known about the characteristics of this work.

Empirical studies of programmers indicate that a many strategies and techniques are used in maintenance activity. Some programmers attempt to understand the entire program prior to making changes, while others zero-in on the area needing change and ignore the rest of the code. Transcripts of programmers "thinking out loud" while doing perfective maintenance show they frequently form and test conjectures about the code under study and browse through the code in a variety of ways while formulating and testing their assertions.

When working with non-trivial programs, programmers use multiple strategies and multiple access paths, all guided by a variety of plans and conjectures. Researchers recognizing this variation have proposed documentation generation tools that use reverse engineering. For example, the Parser/Documenter described in [Land88] applies reverse engineering to generate Nassi-Shneiderman charts from Fortran source code; and attempts to generate specifications documents from Cobol source code are described in [Snee88]. These, and other such attempts, are efforts to improve system documentation by reversing the software lifecycle (e.g., producing design specifications from code). We have taken another, simpler approach; we suggest that system documentation can be improved by reformatting source code into a "book."

#### **The book paradigm for program formatting.**

A book is a collection of information organized to permit easy comprehension and a variety of access methods. Its structure permits top-down and bottom-up traversals, overall comprehension strategies, as-needed strategies, and browsing. The components of a book are all designed to facilitate rapid information access and transfer. Notice the parallels between the information contained in a book and a program:



- o Preface -- an introduction to the book, from the author to the reader; similar to introductory header comments in a program.
- o Table of Contents -- a high-level "map" of the book's contents; similar to a structure chart showing the main components of a program.
- o Indices and Pagination -- low-level "maps" of the book's contents; analogous to program cross-reference maps with line numbers.
- o Chapters -- the major high-level divisions of a book; similar to program units, packages, include files, or the separation of the program main body from its support routines.
- o Sections -- divisions within chapters that group related information and provide mid-level organizational structure; analogous to intramodule code sections (e.g., Pascal's Const, Type, Var, and body sections).
- o Paragraphs -- Chunks of information in the form of grouped sentences; similar to nested or related programming statements (e.g., loops, ifs, cases).
- o Sentences -- Statements and queries delimited and defined by punctuation, type style, character case, etc.; analogous to programming statements and declarations.
- o Punctuation, type style, character case -- Mechanisms for delimiting and highlighting the beginning and/or ending of proper names, phrases, sentences, queries, quotes, paragraphs, sections, chapters, etc.; functionally the same as the punctuation, type style, and character case used in programming.

The major difference is that the format of a book provides simple and immediate clues to aid you in locating and recognizing the parts of a book (e.g., it is trivial to distinguish between names, sentences, paragraphs, etc.). Traditional methods of program formatting do not always provide you with these clues.

#### Reverse engineering code into books.

Our book paradigm of source code formatting calls for both macro-typographic (intermodule) reformatting and micro-typographic (intramodule) reformatting. It does not change the control flow or information structure of the program; it is an entirely typographic arrangement of program source code.

Macro-typographic factors used in the book paradigm include creation of a preface, table of contents, chapter divisions, pagination, and indices. The preface is essentially the program

header comments. The table of contents is a high-level map to the structure of the program (or system); it is automatically generated by a cross reference utility. Indices are also generated automatically for module definition and usage. Other indices for global variables and other identifiers could also be created.

Chapters are created for global declarations, the main program module, support routines accompanying the main program, and "included" code. Note that chapter division also accommodates many "styles of programming." That is, chapters can be defined in object-oriented units, by functional breakdown (support routines), by implementation packages, or any number of considerations.

Micro-typographic factors used in the book paradigm include identification and/or creation of code sections, code paragraphs, sentence structures, and intramodule comments. To do this, techniques such as blank lines, embedded spaces, type styles, and in-line comments, are used to achieve our desired form of source code formatting.

Code sections are separated into easily recognizable units by using blanks, beacons, alignment, and in-line comments to show the beginning and ending of the code sections. For example, the Pascal Const and Var sections are delimited by placing those reserved words in boldface (or all capitalized letters) on separate lines preceded and followed by blank lines. This is analogous to section headings in a book. Code paragraphs are separated into easily recognizable chunks by using the same techniques. Blank lines separate chunks, alignment and embedded spacing (note that this includes indentation) provide spatial clues about the content of the chunks.

Other micro-typographic implementation techniques that can be automated include: (1) adding in-line comments indicating the end of control structures, (2) bold-facing or italicizing procedure calls, (3) aligning conditional structures (e.g., IF's and CASE's), (4) placing blank lines before and after programming constructs that span more than a few lines, (5) highlighting well-defined code segments like data declaration areas, and (6) highlighting globally defined identifiers. There are many such micro-typographic factors that could be used by intelligent source code formatting programs to aid program comprehension.

Our prototype "Book-Maker" programs reverse engineer existing source code listings (Pascal or C) into a printed book format. It is not a completely automated process, however, because certain aspects must be intelligently guided (e.g., chapter division). Much of the reformatting can be automated and can be incorporated into a variety of tools. For example, the principles behind the book paradigm can be implemented within host compiling systems, syntax directed editors, intelligent pretty-printers, and version control archiving and librarian systems. In any case, the key to the viability of the index and

table of contents is that they are consistent with the code file that corresponds to the executable object file. We have circumvented this problem by inserting both the table of contents and the index into the source code as comment blocks.

Organizing program source code into a book format gives you: (1) an easily recognized document paradigm, (2) high-level organizational clues about the code, (3) low-level organizational chunks and beacons, and (4) multiple access paths via the table of contents and indices. It's just a typographic rearrangement of the original source code that is a natural form for program listings which places no additional burden on the programmer.

### **Empirical Tests of the Book Paradigm.**

We have tested our principles of typographic formatting in several empirical studies with both student and professional programmers. Our studies show that a 10 to 20 percent improvement in comprehension can be attained by reformatting code according to our typographic principles. Here we present just two studies demonstrating that the book format improves program maintainability. (For a complete description of our studies see [Cook89, Oman88, Oman89]).

**Experiment 1:** In a controlled study we measured programmers ability to perform maintenance tasks using two different versions of a 1000+ line Pascal program -- one a traditional listing, the other our book paradigm listing. The program was a working text editor taken from [Schn81] and modified by removing a small procedure which handled the free-form command inputs to the editor. The five calls to the procedure were also removed. The resulting modified program still worked; it was just incapable of handling free-form inputs.

The modified program was then ported into Lightspeed Pascal (a syntax directed code formatter) and printed with pagination. This listing was version 1; it represents the traditional manner in which Pascal source code is formatted. Version 2 was a macro-typographic rearrangement of version 1 as defined by our book paradigm. That is, the code was separated into chapters and a table of contents and module index were added. There were no other changes made to the code.

Participants in the experiment were 53 Computer Science students enrolled in a senior/graduate level operating systems course at Oregon State University. They were randomly assigned into two groups; roughly half the subjects (28) received the traditional listing while the other half (25) received the book listing. Each subject was given a listing and asked to recreate the missing procedure that would enable free-form command inputs. They were also asked to indicate where the procedure would be called. Hence, the maintenance exercise called for them to enhance the program by adding a module that skipped spaces on the command input line. This is not unlike many real world



maintenance tasks. In order to do the exercise, they first had to understand the command line record structure and then understand the execution flow of the routines that manipulated the command line. Then, and only then, could they begin to write the missing procedure.

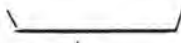
For each subject we measured the time required to perform the task (up to 1 hour), their ability to recreate the missing routine, and their ability to identify where it was called (five locations). No special instructions or explanations were given to subjects receiving the book listing. This was deliberately done as a test to see if subjects would "naturally" use the book listing (i.e., without training).

The code writing portion of the maintenance task was scored by tallying subjects' responses into four categories: (1) routines similar or identical to the one that was removed, (2) functionally correct but dissimilar routines, (3) incorrect routines, and (4) those who could not complete the task (i.e., they gave up or could not even get started). We originally expected that at least 50 percent of each group would be able to complete the task, but results from the code writing portion (shown in Table 1) indicate that the book listing group outperformed the traditional listing group by approximately two correct answers to one! A Chi-square analysis of the results, assuming an equal .25 probability across all four categories, indicates that differences between the traditional listing and book listing are significant ( $X=10.45$ ,  $p<.025$ ,  $d.f.=3$ ).

Group differences can also be seen by collapsing the two correct categories together (exactly correct plus functionally correct) and collapsing the two incorrect categories together (wrong plus not finished). The total correct is 52 percent for the book listing versus 25 percent for the traditional listing. That is, 27 percent more got it right when working with the book format! A Chi-square test of independence on the resulting 2 by 2 design (using Pearson's computed expectancy values) shows a significant difference between the two versions ( $X=3.73$ ,  $p<=.06$ ,  $d.f.=1$ ). Furthermore, subjects in the traditional listing group were twice as likely to quit or not even be able to start writing code.

The procedure call portion of the maintenance task was scored only for those subjects that wrote a correct routine. Results for the call identification task are shown in Table 2. For the traditional listing group the 7 subjects that successfully completed the routine, correctly identified a total of 12 places where Skip\_Blanks needed to be called. This was an average of 1.71 correct identifications per person; an overall accuracy rate of only 34 percent. On the other hand, the 13 subjects that correctly wrote the Skip\_Blanks routine using the book listing correctly identified a total of 31 Skip\_Blanks calls; an average of 2.38 correct identifications per person, an overall accuracy rate of 48 percent.

**Table 1. Code Writing Ability.**

|   | exactly<br>correct  | functionally<br>correct | wrong | gave up or<br>not finished |
|---|---|-------------------------|-------|----------------------------|
| Traditional listing<br>(n = 28)             | 14 %  | 11 %                    | 36 %  | 39 %                       |
| Book listing<br>(n = 25)                    | 36 %  | 16 %                    | 32 %  | 16 %                       |
|   |  |                         |       |                            |
| Total correct:                              | Traditional   | -- 25 %                 |       |                            |
|   | Book listing  | -- 52 %                 |       |                            |
| Percent difference between groups..... 27 % |   |                         |       |                            |

**Table 2. Call Identification Ability.**

| Dependent measure                            | Traditional<br>listing | Book<br>listing |
|--|------------------------|-----------------|
| Number correct                               | 7                      | 13              |
| Total correct identifications                | 12                     | 31              |
| Average identifications per person           | 1.71                   | 2.38            |
| Percentage accuracy for the group            | 34.2 %                 | 47.6 %          |
| Percent difference between groups.... 13.4 % |                        |                 |

Results from this experiment show the benefit of using the book paradigm for macro-typographic style. We emphasize the the only difference between version 1, the traditional listing, and version 2, the book format listing, was that the code was divided into chapters and indexed by a table of contents and a module index. There were no micro-typographic differences between the two versions.

**Experiment 2:** To demonstrate that our book paradigm is useful to professional programmers working with large programs, and to test the feasibility of the book paradigm for large programs, we conducted an empirical study of real programmers working with a large industrial program written in C.

A portion of the X\_Windows package was obtained from a large international computer corporation. X\_Windows is a window and mouse management system originally developed at M.I.T. and now bundled with various Unix systems. The C code we obtained consisted of a main program file and two of its include files. There were 1057 lines of commented C code in the three files.

Two printed listings of the X\_Windows program were created. As in our previous experiment, version 1 was the original listing as received from the corporation, except that it was laser printed with pagination for readability. Version 2 was our book formatted version of the code. All changes were simple typographic alterations; no module rearrangement and identifier renaming was used, and no comments were added other than the table of contents and the module index. The resulting listing consisted of 1098 lines of commented C code including the table of contents and the index. Although these two components added 269 lines of comments to the source file, the micro-typographic statement reformatting sufficiently compressed the original source code such that the end result was only 41 lines longer than the original code!

Twelve professional programmers, each with at least two years of C programming experience, volunteered to serve as subjects. Each subject was paid \$40.00. The 12 programmers were paired by experience and job function so each member of a pair had approximately the same experience with Unix, C, and X\_Windows. For each of the six pairs, one member was assigned to work with version 1 while the other worked with version 2. The version assignment was determined by a coin flip for each pair. Subjects were tested one at a time in a closed room. The test sessions took about 2 hours.

Two of the subjects were deliberately chosen because they were corporate maintenance programmers responsible for portions of the X\_Windows system. Both were familiar with the test program and had previously studied the include files. They were experts already familiar with the code to be studied. (None of the other subjects had prior experience with the code to be studied.) Background characteristics for the subjects appears in Table 3. The subject pairs are listed in decreasing order of

Table 3. Professional Programmers' Experience.

| pair # | degree of X_Windows & Unix experience | subject label  | yrs. prof. experience | yrs. C experience |
|--------|---------------------------------------|----------------|-----------------------|-------------------|
| 1      | X_Windows maintenance experts         | X <sub>t</sub> | 9                     | 4                 |
|        |                                       | X <sub>b</sub> | 7                     | 4                 |
| 2      | Unix development programmers          | A <sub>t</sub> | 7                     | 5                 |
|        |                                       | A <sub>b</sub> | 8                     | 7                 |
| 3      | Unix & C systems programmers          | B <sub>t</sub> | 8                     | 6                 |
|        |                                       | B <sub>b</sub> | 7                     | 5                 |
| 4      | Unix & C applications programming     | C <sub>t</sub> | 9                     | 3                 |
|        |                                       | C <sub>b</sub> | 7                     | 2                 |
| 5      | C applications programming            | D <sub>t</sub> | 12                    | 2                 |
|        |                                       | D <sub>b</sub> | 10                    | 2                 |
| 6      | C applications programming            | E <sub>t</sub> | 13                    | 2                 |
|        |                                       | E <sub>b</sub> | 6                     | 2                 |

Note: Subject label subscripts denote listing version.  
<sub>t</sub> for traditional listing, and <sub>b</sub> for book format listing.



Unix and C experience. The first pair, labeled  $X_t$  and  $X_b$ , are the two X\_Windows experts.

Subjects were given one of the two code versions and asked to complete a comprehension/ maintenance exercise consisting of: (1) a 30 minute study period with "Think aloud" protocols, (2) a 7 question (10 points) oral comprehension test, (3) a pen and paper exercise to create a call graph for the program, and (4) some open-ended questions about the way they work with large programs. The test session took approximately 2 hours and was recorded on audio-tape.

For each programmer we measured their scores and time for the comprehension test and call graph exercise. The think aloud protocols and open-ended questions were just used as a data gathering device to check for behavior patterns between and within groups. All subjects received exactly the same instructions; that is, subjects working with the book listing received no explanation or justification about the book listing.

Scores and times for the comprehension test are shown in Table 4 and Figure 1. As can be seen, programmers working with the book listing scored better, and did so faster, than the programmers working with the traditional listing. A comparison between the two groups can best be seen in Figure 1, which plots time and score for each subject. Note that there is little difference between the two experts; hence, they represent the top-line performance for the task. Also note that all other subjects working with the book format listing performed as well as the two experts, but none of the subjects working with the traditional listing did! We emphasize that the two experts were already familiar with the code. The clear separation between the subjects working with the traditional listing and those working with the book format listing (excluding experts) reflects the improved comprehension afforded by the book listing.

The call graph exercise was a measure of their ability to work with the program listing. In a call graph, each node represents a function (module) and each edge represents the call to that function. An incomplete call graph, consisting of the 12 top-level nodes (main and the 11 functions it calls) and their 11 edges, was given to the subjects with instructions to complete the call graph. The completed call graph contains 23 nodes and 39 edges, so the task was to find and add the missing 11 nodes and 28 edges. The score for the exercise was the total number of correct nodes and edges on their completed call graph.

Scores and times for the call graph exercise are shown in Table 5 and Figure 2. As can be seen, programmers working with the book listing scored better, and did so faster, than the programmers working with the traditional listing. Group differences can best be seen in Figure 2, which plots time and score for each subject. Note the major differences between groups; on the average, subjects working with the traditional listing missed twice as many call graph connections and took one

Table 4. Comprehension Test Results.

| subject                    | test questions |   |   |    |    |   |    |    |    |    | total score | total time |
|----------------------------|----------------|---|---|----|----|---|----|----|----|----|-------------|------------|
|                            | 1              | 2 | 3 | 4a | 4b | 5 | 6a | 6b | 7a | 7b |             |            |
| X <sub>t</sub> :           | c              | x | x | c  | c  | c | c  | c  | c  | c  | 8           | 13         |
| A <sub>t</sub> :           | x              | c | c | c  | c  | x | c  | x  | c  | x  | 6           | 18         |
| B <sub>t</sub> :           | c              | c | x | c  | c  | c | c  | x  | c  | x  | 7           | 17         |
| C <sub>t</sub> :           | x              | x | c | c  | c  | x | c  | x  | c  | x  | 5           | 16         |
| D <sub>t</sub> :           | c              | x | x | c  | c  | c | c  | x  | c  | x  | 6           | 16         |
| E <sub>t</sub> :           | c              | x | c | c  | x  | x | c  | x  | c  | c  | 6           | 26         |
| traditional list averages: |                |   |   |    |    |   |    |    |    |    | 6.33        | 17.6       |
| X <sub>b</sub> :           | c              | c | x | c  | c  | c | c  | x  | c  | c  | 8           | 7          |
| A <sub>b</sub> :           | c              | c | c | c  | c  | x | c  | x  | c  | c  | 8           | 17         |
| B <sub>b</sub> :           | c              | c | x | c  | c  | c | c  | c  | c  | x  | 8           | 10         |
| C <sub>b</sub> :           | c              | c | x | c  | c  | c | c  | x  | c  | c  | 8           | 13         |
| D <sub>b</sub> :           | c              | c | x | c  | c  | c | c  | x  | c  | c  | 8           | 12         |
| E <sub>b</sub> :           | c              | x | c | c  | c  | x | c  | x  | c  | c  | 7           | 13         |
| book list averages:        |                |   |   |    |    |   |    |    |    |    | 7.83        | 12.0       |

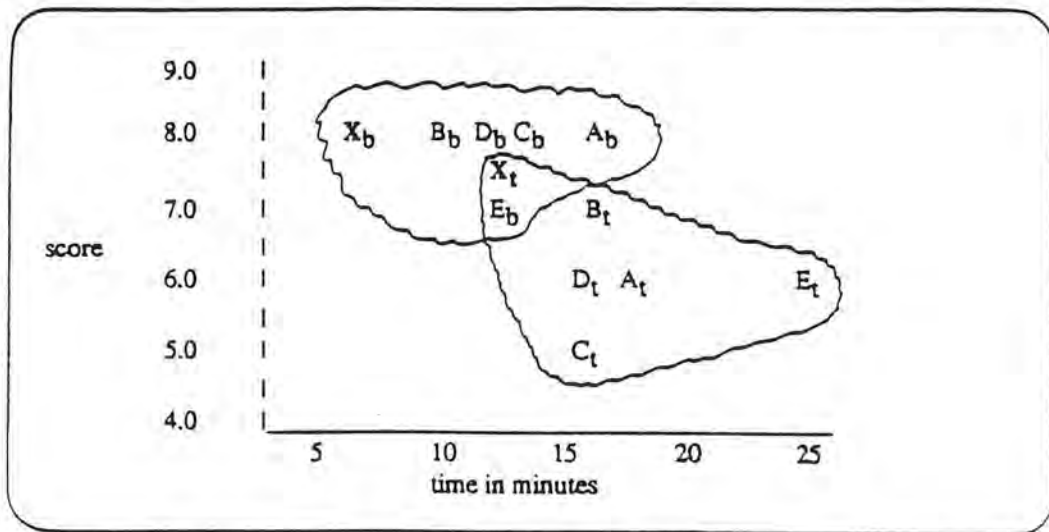


Figure 1. Scores and Times from Comprehension Test.

Table 5. Call Graph Exercise Results.

| <u>traditional listings</u> |      |           | subject        | <u>book format listings</u> |      |           |
|-----------------------------|------|-----------|----------------|-----------------------------|------|-----------|
| score                       | time | traversal |                | score                       | time | traversal |
| 32                          | 16   | 2         | X              | 35                          | 14   | 1 & 3     |
| 28                          | 16   | 1 & 3     | A              | 34                          | 11   | 1         |
| 27                          | 13   | 3         | B              | 35                          | 12   | 3         |
| 30                          | 14   | 5         | C              | 36                          | 13   | 2         |
| 30                          | 14   | 1         | D              | 36                          | 15   | 2         |
| 34                          | 30   | 2         | E              | 33                          | 16   | 4         |
| 30.2                        | 17.2 |           | <- averages -> | 34.8                        | 13.5 |           |

-----

traversals: 1. linear trace through code listing.  
 2. entirely top-down, depth first execution order.  
 3. heuristically guided depth first execution order.  
 4. entirely top-down, breadth first execution order.  
 5. heuristically guided breadth first execution order.

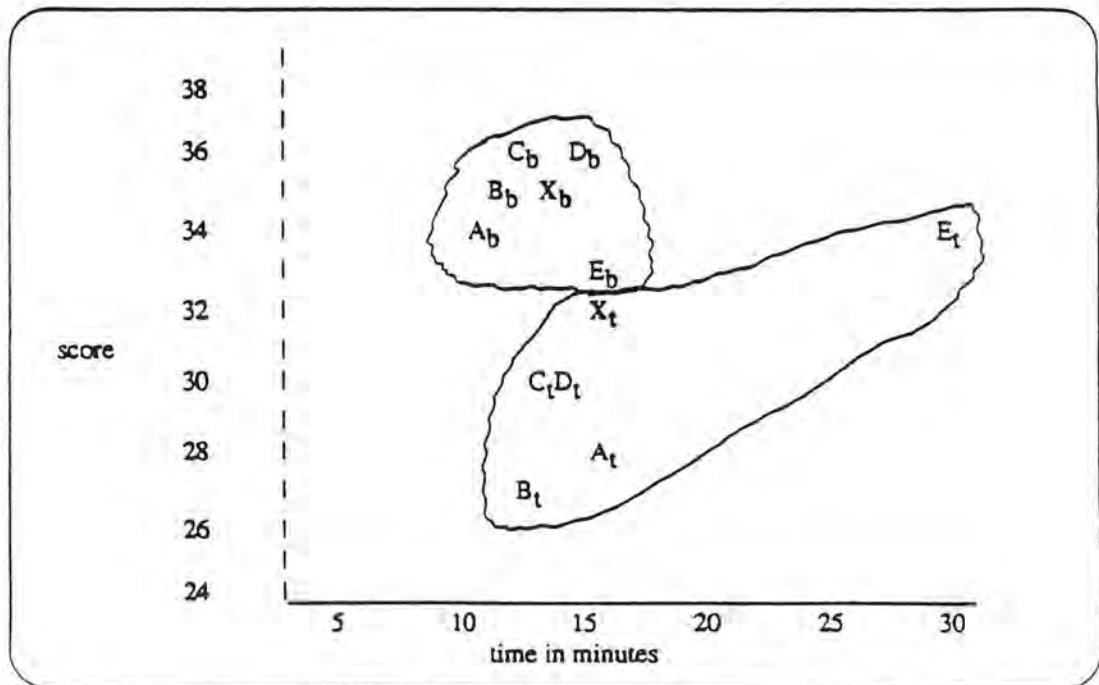


Figure 2. Scores and Times from Call Graph Exercise.

minute longer, than those working with the book format listing. Once again, subjects working with the book format listing performed as well or better than the experts; those working with the traditional listing performed noticeably worse.

Results from this experiment show that the book paradigm is a natural form for formatting source code that is better than traditional methods. In every matched pair the subject working with the book listing scored better, worked faster, and expressed more feelings of comfort and capability than those working with the standard listings. All programmers working with the book listing agreed it was a better way of formatting code than they had seen before. Further, the programmers working with the book listing performed as well or better than the two expert programmers already familiar with the code. This is a sharp contrast to the programmers working with the traditional listing who performed noticeably worse than the two experts.

We emphasize that in both experiments subjects used the book format listing without any explanation, description, or justification; and they did so better than their counterparts working with traditional listings.

#### **Conclusions and discussion.**

Programmers use many strategies and approaches when working with source code. Usually, the only reliable documentation you have for a program is the source code listing (or file). But, unless it's a trivial program, that listing is just a linear ordering of a non-linear collection of functions. You need multiple avenues or access paths to get "into" the code. Transforming code into books creates the organizational structure and clues that permit a variety of access paths.

Good typographic formatting reflects the underlying structure of the code by providing visual clues and a variety of ways to view the code. The book model is just one mechanism for implementing those objectives. It uses reverse engineering to convert existing source code listings into book-like documents that have macro- and micro-typographic clues to assist in program comprehension.

Our controlled experiments and empirical tests of the book paradigm show that it aids in maintenance tasks on large programs. Further, we have shown that professional programmers can benefit from the book model because it's a "natural" format for source code listings.

This work has several implications on code formatting tools:

1. Useful code formatting tools must be more sophisticated and compatible with the way programmers view and work with code. Today's simplistic pretty-printers and syntax



directed editors are inadequate and, in fact, decrease maintainability by obscuring comprehension clues.

2. Language directed editors could be designed to incorporate "intelligent" code formatting principles. This could be implemented in varying degrees, from simply highlighting beacons while the code is being displayed, to arranging code into a book format while it is being edited.
3. Hypertext code maintenance tools could be designed to allow programmers to have simultaneous views into the code being studied. Current hypertext code viewing systems access and display information outside the source code listing; this creates a version control problem. The power of the book paradigm is that the cross referencing information is incorporated into or extracted from the source code.

## References

- [Adel81] B. Adelson, "Problem Solving and the Development of Abstract Categories in Programming Languages," Memory and Cognition, vol. 9(4), 1981, pp. 422-433.
- [Adel84] B. Adelson, "When Novices Surpass Experts: The Difficulty of a Task May Increase With Expertise," Journal of Experimental Psychology, vol. 10(3), 1984, pp. 483-495.
- [Bend87] S. Bendifallah & W. Scacchi, "Understanding Software Maintenance Work," IEEE Transactions on Software Engineering, SE-13(3), Mar. 1987, pp. 311-323.
- [Bent86] J. Bentley & D. Knuth, "Literate Programming," Communications of the ACM, vol. 29(5), May 1986, pp. 364-369.
- [Broo83] R. E. Brooks, "Towards a Theory of the Comprehension of Computer Programs," International Journal of Man-Machine Studies, vol. 18, 1983, pp. 543-554.
- [Cook89] C. Cook & P. Oman, "Typographic Style is More Than Cosmetic," O.S.U. Computer Science Technical Report, submitted to Communications of the ACM, under review.
- [Land88] L. Landis, P. Hyland, A. Gilbert, & A. Fine, "Documentation in a Software Maintenance Environment," Conference on Software Maintenance 1988 Proceedings, IEEE Computer Society Press, 1988, pp. 66-73.
- [Oman89] P. Oman & C. Cook, "A Programming Style Taxonomy," O.S.U. Computer Science Technical Report, submitted to Journal of Structured Programming, under review.
- [Oman88] P. Oman & C. Cook, "A Paradigm for Programming Style Research," ACM SIGPLAN Notices, vol. 23(12), Dec. 1988, pp. 69-78.
- [Pari83] G. Parikh & N. Zvegintzov, "The World of Software Maintenance," Tutorial on Software Maintenance, (G. Parikh & N. Zvegintzov, editors), IEEE Computer Society Press, Los Angeles CA, 1983, pp. 1-3.
- [Schn81] G. Schneider & S. Buell, Advanced Programming and Problem Solving With Pascal, John Wiley & Sons, New York NY, 1981.
- [Snee88] H. Sneed & G. Jandrasics, "Inverse Transformation of Software from Code to Specification," Conference on Software Maintenance 1988 Proceedings, IEEE Computer Society Press, 1988, pp. 102-109.

- [Solo84] E. Soloway, & K. Ehrlich, "Empirical Studies of Programming Knowledge," IEEE Transactions on Software Engineering, SE-10(5), Sept. 1984, pp. 595-609.
- [Your75] E. Yourdon, Techniques of Program Structure and Design, Prentice-Hall, Englewood Cliffs NJ, 1975.