TYPOGRAPHIC STYLE IS MORE THAN COSMETIC

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

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

89-60-5

TYPOGRAPHIC STYLE IS MORE THAN COSMETIC

Paul W. Oman
Computer Science Department
University of Idaho
Moscow, Idaho 83843
(208) 885-7219

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

Abstract

There is disagreement about the role and importance of typographic style (source code formatting and commenting) in program comprehension. Results from experiments and opinions in programming style books are mixed.

This paper presents principles of typographic style consistent and compatible with the results of program comprehension studies. It introduces the book format paradigm, an implementation of these style principles. Results from four experiments demonstrate that the typographic style principles embodied in the book format significantly aid program comprehension and reduce maintenance effort.

The typographic style principles presented in this paper have direct application to code formatters, such as prettyprinters and syntax-directed editors, programming language design, and programming instruction practices.

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

Keywords: Programming style, coding style, code formatting.

INTRODUCTION

Program comprehension plays an important role in many programming tasks. For example, about one-half of a maintenance programmer's time is spent studying the source code and related documentation. Maintenance programmers cite understanding the intent and style of another programmer's code as the major difficulty in making a change [4]. Unfortunately, in many instances the only reliable description of a program is the source code itself.

The contribution of typographic style (source code formatting) to the understandability of the program is not clear. Ledgard and Tauer [7] believe that code "should speak for itself" and that code formatting is "not window dressing, but a visible display of the meaning" of the program. On the other hand, in their classic book on programming style, Kernighan and Plauger [5] state that "if the code is clear and simple to begin with, formatting details are of secondary importance." They illustrate each of their style rules by describing the shortcomings of an example code segment, rewriting the example in a better style, and drawing the general rule from the specific case. It is interesting to note that virtually all of their rewritten versions contain unmentioned changes in typographic style.

Experimental studies of typographic factors have also been inconclusive. For example, most programmers believe that programs with indentation are easier to comprehend than programs without indentation. However, Shneiderman and McKay [15] found no significant differences between subjects who were asked to locate and correct an error in the indented and unindented versions of the same program. Also, Love [9] tested the impact of indentation and control flow complexity on program comprehension and found no significant differences between indented and unindented code. However, a later experiment by Miara, et al, [10] did show significant differences between indentation levels. Subjects were tested with programs containing one of four different methods of indentation: no indentation, two space indentation, four space indentation, and six space indentation. They found that indentation does aid program comprehension and that the optimal level of indentation is between two and four spaces.

In light of these disagreements about the importance of typographic style, Sheil [14] noted that "the existence of both positive and negative results suggests a search for some set of principles which indicate how and when formatting techniques will be effective." In this direction, Baecker [2] has developed a framework for "program visualization" based on principles of effective graphics design. His approach is to enhance the source code through the use of multiple fonts, variable character widths, proportional character spacing, and gray-scale tints; the enhanced source code is output on high resolution bit-mapped displays and laser printers. He found a 25 percent increase in the readability of an enhanced source text of C programs as measured by comprehension quiz scores.

In this paper we present a set of principles for program formatting that are based in program comprehension theory. We then introduce the book format, an implementation of these principles, and show through a series of experiments that the book format significantly improves program comprehension. The major difference between our work and Baecker's is that our principles are based on results from program comprehension studies and his are based on principles of graphic design. He concentrates on improving the appearance of the source code while we concentrate on providing the clues and access mechanisms used by programmers in understanding a program.

TYPOGRAPHIC STYLE PRINCIPLES

We define typographic style as the set of style characteristics concerned with the formatting and commenting of source code. By definition, typographic style does not impact the execution of the program. We have found it convenient to divide typographic style into macro and micro subclasses [12]. Macro-typographic style factors include overall program formatting, global and intermodule commenting, module separation conventions, identifier naming conventions, and conventions for special case, font or type styles. Micro-typographic style factors include statement formatting, indentation and embedded spacing, use of blank lines, and intramodule commenting.

Our typographic style principles for when and how to format and comment source programs are based on results from programmer comprehension studies. All programmer comprehension studies support the existence of:

1. Mental schemata, or plans, that guide the programmer's comprehension of code [1,16]. Programmers acquire and modify these plans through experience; they are an integral part of long term memory.
2. Chunks, or meaningful units of information, that programmers use to organize and remember code [1].
3. Beacons, or highlighted semantic clues, that are used to direct the review and recognition of code [3]. Beacons

are used for searching, chunking and hypothesis checking.

4. Multiple strategies and access paths used by programmers when working with non-trivial programs [8]. Strategies are guided by a variety of plans and conjectures depending upon individual differences, application domains, and the implementation of the code and supporting system.

Using these results from comprehension studies, we identified several principles of macro and micro-typographic style. Example macro-typographic principles:

1. Make the components and organization of the program obvious. This means that code areas for global definitions, the main program, support routines, and included code segments should be easily identifiable. Module separation should also be obvious.

2. Identify the purpose and use of each component.

3. Make the execution control and information flow between components readily apparent. Highlight beacons indicating intermodule control flow and communication.

4. Make the program readable and easy to browse by providing different access paths into the code. That is, clues should be provided to enable non-linear code traces (e.g. top-down, bottom-up, focused, and browsing).

Example micro-typographic principles:

1. Make the sections and organization of the module obvious. This means dividing modules into easily recognizable parts (e.g. constants, data declarations, and code body) by highlighting beacons that delimit sections.

2. Identify the purpose and use of each section.

3. Make the underlying control and information flow within the module obvious. This means control and information constructs should be separated into easily recognizable chunks. Highlight beacons indicating changes in control flow.

4. Make statements readable and easy to scan by providing spatial clues and "white space" to indicate statement grouping and separation.

We emphasize that these are general principles of good typographic style. We do not claim that this is a complete list;

rather, we provide these examples to demonstrate the separation of principle and implementation. Without concern for implementation techniques we have enumerated several principles of good typographic style consistent with all models of program comprehension. Note that all of the above principles may be implemented via numerous typographic factors. For instance, commenting, naming, blank lines, and embedded spacing can all serve to separate modules, sections, chunks, and statements.

In the next section we introduce the "book format" paradigm, a mechanism of implementing the principles outlined above. We then show through a series of experiments that our macro- and micro-typographic style principles, implemented in "book format," aid program comprehension and reduce maintenance effort.

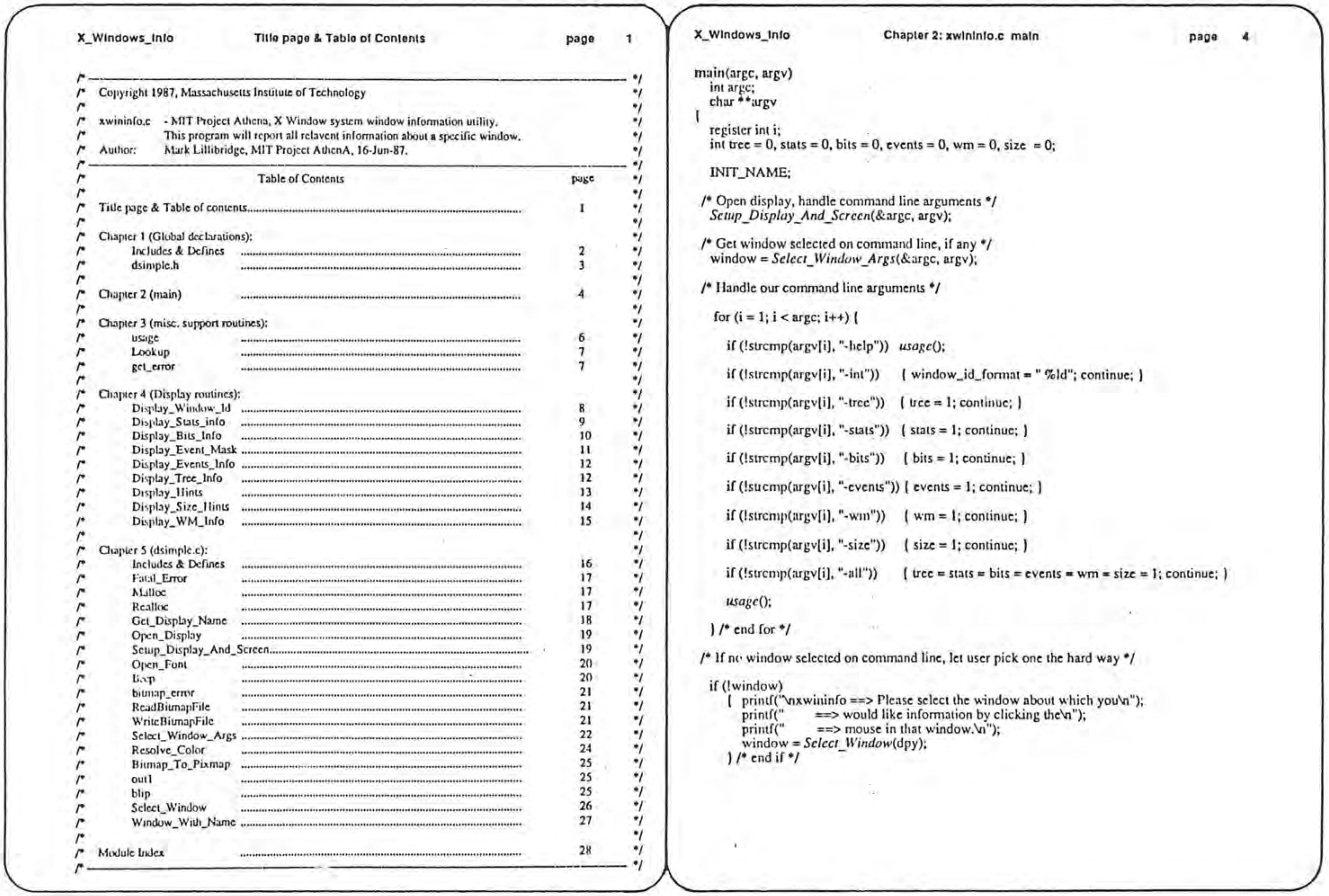
BOOK FORMAT PARADIGM

Programmers use multiple strategies and access paths when working with programs. A book is a collection of information organized to permit easy comprehension and a variety of access methods. The components of a book (preface, table of contents, indices and pagination, chapters, sections, paragraphs, sentences, and punctuation, type style, and character case) are all designed to facilitate rapid information access and transfer. There are obvious parallels between the information contained in a book and that of program source code. The major difference is that the typographic style of a book provides simple and immediate clues to aid the reader in locating and recognizing the parts of a book. Traditional methods of program formatting do not always provide these typographic clues. Hence, the book format is a more appropriate form for representing program source code.

Selected pages from a "book format" program listing are shown in Figure 1. The program is a portion of the X_Windows system, originally from MIT, rewritten into book form. (For a more complete description, see Experiment 4 in the next section.) Implementation techniques for most of our typographic principles can be seen in this example.

The book format paradigm of source code formatting incorporates both macro- and micro-typographic style factors. Macro-typographic factors used in the book format paradigm include creation of a preface, table of contents, chapter divisions, pagination, and indices. The preface is a block of comments identifying author, system, dates, etc. The table of contents is a high-level map to the structure of the program (or system). It can be generated automatically by a cross reference utility that recognizes chapter breakdowns. Chapters can be created for global declarations, the main program module, support routines accompanying the main program, and "included" code. Note that the chapter division accommodates many "styles of programming" as chapters can be defined in object-oriented units, by functional breakdown, by implementation, or by any number of considerations. Indices can also be generated automatically. Indices for module definition and usage, global variables, and virtually all other identifiers, could be created by a simple symbol table management and cross referencing program.

Micro-typographic factors used in the book format paradigm include identification and/or creation of code segments, code paragraphs, sentence structures, and intramodule comments. To do



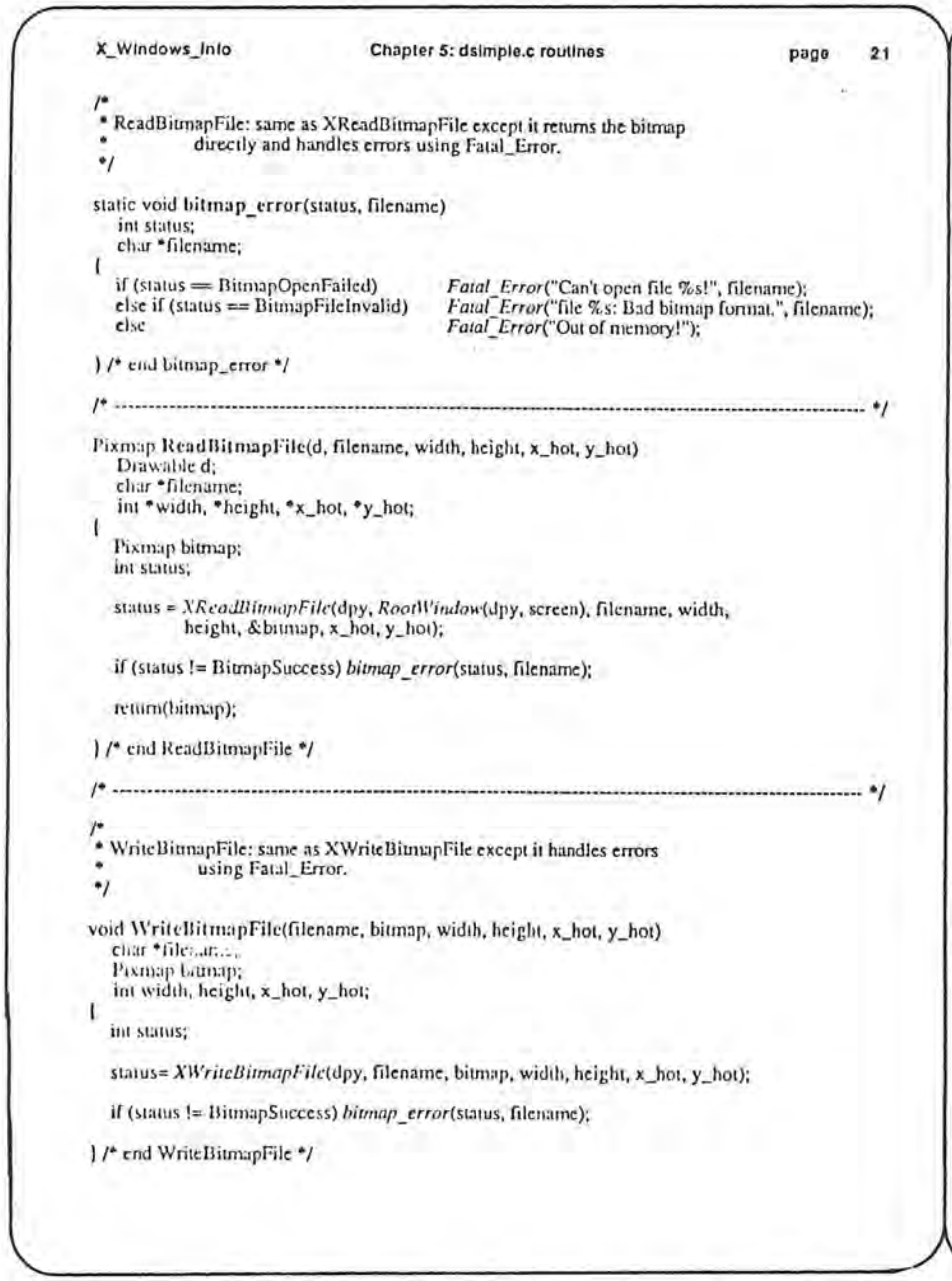
X_Windows_Info   Title page & Table of Contents   page 1

```
/*------------------------------------------------------------------------------*/
/*  Copyright 1987, Massachusetts Institute of Technology                       */
/*                                                                              */
/*  xwininfo.c  - MIT Project Athena, X Window system window information utility. */
/*               This program will report all relavent information about a specific window. */
/*  Author:     Mark Lillibridge, MIT Project AthenA, 16-Jun-87.                */
/*------------------------------------------------------------------------------*/
/*                        Table of Contents                        page         */
/*                                                                              */
/*  Title page & Table of contents.......................................  1    */
/*                                                                              */
/*  Chapter 1 (Global declarations):                                            */
/*        Includes & Defines ............................................  2    */
/*        dsimple.h ......................................................  3    */
/*                                                                              */
/*  Chapter 2 (main) .....................................................  4    */
/*                                                                              */
/*  Chapter 3 (misc. support routines):                                         */
/*        usage ..........................................................  6    */
/*        Lookup .........................................................  7    */
/*        get_error ......................................................  7    */
/*                                                                              */
/*  Chapter 4 (Display routines):                                               */
/*        Display_Window_Id ..............................................  8    */
/*        Display_Stats_info .............................................  9    */
/*        Display_Bits_Info .............................................. 10    */
/*        Display_Event_Mask ............................................. 11    */
/*        Display_Events_Info ............................................ 12    */
/*        Display_Tree_Info .............................................. 12    */
/*        Display_Hints .................................................. 13    */
/*        Display_Size_Hints ............................................. 14    */
/*        Display_WM_Info ................................................ 15    */
/*                                                                              */
/*  Chapter 5 (dsimple.c):                                                      */
/*        Includes & Defines ............................................. 16    */
/*        Fatal_Error .................................................... 17    */
/*        Malloc ......................................................... 17    */
/*        Realloc ........................................................ 17    */
/*        Get_Display_Name ............................................... 18    */
/*        Open_Display ................................................... 19    */
/*        Setup_Display_And_Screen........................................ 19    */
/*        Open_Font ...................................................... 20    */
/*        Beep ........................................................... 20    */
/*        bitmap_error ................................................... 21    */
/*        ReadBitmapFile ................................................. 21    */
/*        WriteBitmapFile ................................................ 21    */
/*        Select_Window_Args ............................................. 22    */
/*        Resolve_Color .................................................. 24    */
/*        Bitmap_To_Pixmap ............................................... 25    */
/*        outl ........................................................... 25    */
/*        blip ........................................................... 25    */
/*        Select_Window .................................................. 26    */
/*        Window_With_Name ............................................... 27    */
/*                                                                              */
/*  Module Index ......................................................... 28    */
/*------------------------------------------------------------------------------*/
```

X_Windows_Info   Chapter 2: xwininfo.c main   page 4

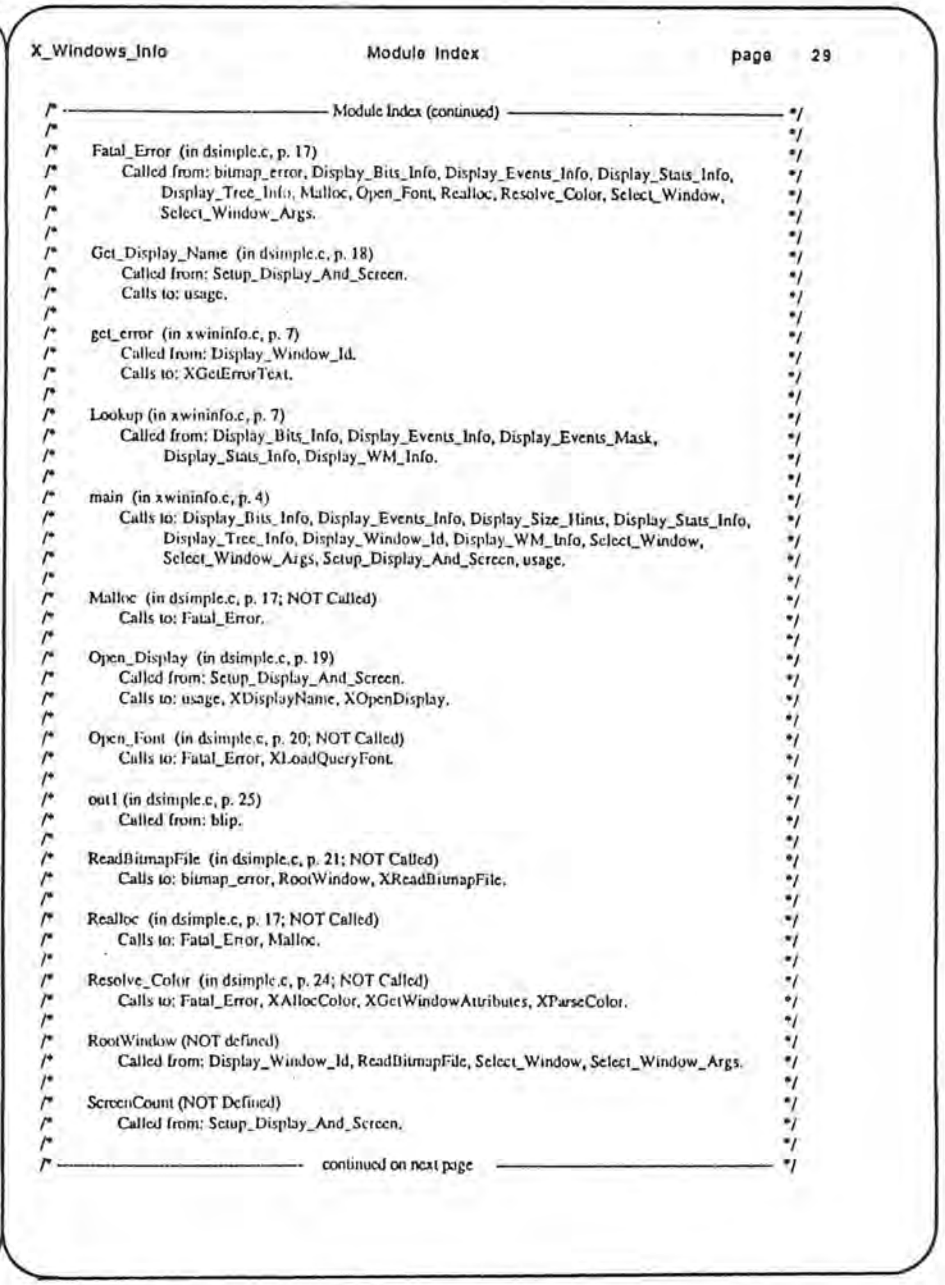
```
main(argc, argv)
   int argc;
   char **argv
{
   register int i;
   int tree = 0, stats = 0, bits = 0, events = 0, wm = 0, size  = 0;

   INIT_NAME;

 /* Open display, handle command line arguments */
   Setup_Display_And_Screen(&argc, argv);

 /* Get window selected on command line, if any */
   window = Select_Window_Args(&argc, argv);

 /* Handle our command line arguments */

    for (i = 1; i < argc; i++) {

       if (!strcmp(argv[i], "-help"))  usage();

       if (!strcmp(argv[i], "-int"))     { window_id_format = " %ld"; continue; }

       if (!strcmp(argv[i], "-tree"))    { tree = 1; continue; }

       if (!strcmp(argv[i], "-stats"))   { stats = 1; continue; }

       if (!strcmp(argv[i], "-bits"))    { bits = 1; continue; }

       if (!strcmp(argv[i], "-events"))  { events = 1; continue; }

       if (!strcmp(argv[i], "-wm"))      { wm = 1; continue; }

       if (!strcmp(argv[i], "-size"))    { size = 1; continue; }

       if (!strcmp(argv[i], "-all"))     { tree = stats = bits = events = wm = size = 1; continue; }

       usage();

    } /* end for */

 /* If no window selected on command line, let user pick one the hard way */

   if (!window)
       { printf("\nxwininfo ==> Please select the window about which you\n");
         printf("         ==> would like information by clicking the\n");
         printf("         ==> mouse in that window.\n");
         window = Select_Window(dpy);
       } /* end if */
```

Figure 1. Pages from a Book Format Listing.



X_Windows_Info   Chapter 5: dsimple.c routines   page 21

```
/*
 * ReadBitmapFile: same as XReadBitmapFile except it returns the bitmap
 *            directly and handles errors using Fatal_Error.
 */

static void bitmap_error(status, filename)
   int status;
   char *filename;
{
   if (status == BitmapOpenFailed)        Fatal_Error("Can't open file %s!", filename);
   else if (status == BitmapFileInvalid)  Fatal_Error("file %s: Bad bitmap format.", filename);
   else                                   Fatal_Error("Out of memory!");

} /* end bitmap_error */

/* ------------------------------------------------------------------------------------------ */

Pixmap ReadBitmapFile(d, filename, width, height, x_hot, y_hot)
   Drawable d;
   char *filename;
   int *width, *height, *x_hot, *y_hot;
{
   Pixmap bitmap;
   int status;

   status = XReadBitmapFile(dpy, RootWindow(dpy, screen), filename, width,
            height, &bitmap, x_hot, y_hot);

   if (status != BitmapSuccess) bitmap_error(status, filename);

   return(bitmap);

} /* end ReadBitmapFile */

/* ------------------------------------------------------------------------------------------ */

/*
 * WriteBitmapFile: same as XWriteBitmapFile except it handles errors
 *            using Fatal_Error.
 */

void WriteBitmapFile(filename, bitmap, width, height, x_hot, y_hot)
   char *filename;
   Pixmap bitmap;
   int width, height, x_hot, y_hot;
{
   int status;

   status= XWriteBitmapFile(dpy, filename, bitmap, width, height, x_hot, y_hot);

   if (status != BitmapSuccess) bitmap_error(status, filename);

} /* end WriteBitmapFile */
```



X_Windows_Info   Module Index   page 29

```
/* ------------------------------ Module Index (continued) ------------------------------ */
/*                                                                                        */
/*    Fatal_Error  (in dsimple.c, p. 17)                                                  */
/*       Called from: bitmap_error, Display_Bits_Info, Display_Events_Info, Display_Stats_Info, */
/*            Display_Tree_Info, Malloc, Open_Font, Realloc, Resolve_Color, Select_Window,  */
/*            Select_Window_Args.                                                         */
/*                                                                                        */
/*    Get_Display_Name  (in dsimple.c, p. 18)                                             */
/*       Called from: Setup_Display_And_Screen.                                           */
/*       Calls to: usage.                                                                 */
/*                                                                                        */
/*    get_error  (in xwininfo.c, p. 7)                                                    */
/*       Called from: Display_Window_Id.                                                  */
/*       Calls to: XGetErrorText.                                                         */
/*                                                                                        */
/*    Lookup (in xwininfo.c, p. 7)                                                        */
/*       Called from: Display_Bits_Info, Display_Events_Info, Display_Events_Mask,        */
/*            Display_Stats_Info, Display_WM_Info.                                        */
/*                                                                                        */
/*    main  (in xwininfo.c, p. 4)                                                         */
/*       Calls to: Display_Bits_Info, Display_Events_Info, Display_Size_Hints, Display_Stats_Info, */
/*            Display_Tree_Info, Display_Window_Id, Display_WM_Info, Select_Window,       */
/*            Select_Window_Args, Setup_Display_And_Screen, usage.                        */
/*                                                                                        */
/*    Malloc  (in dsimple.c, p. 17; NOT Called)                                           */
/*       Calls to: Fatal_Error.                                                           */
/*                                                                                        */
/*    Open_Display  (in dsimple.c, p. 19)                                                 */
/*       Called from: Setup_Display_And_Screen.                                           */
/*       Calls to: usage, XDisplayName, XOpenDisplay.                                     */
/*                                                                                        */
/*    Open_Font  (in dsimple.c, p. 20; NOT Called)                                        */
/*       Calls to: Fatal_Error, XLoadQueryFont.                                           */
/*                                                                                        */
/*    out1 (in dsimple.c, p. 25)                                                          */
/*       Called from: blip.                                                               */
/*                                                                                        */
/*    ReadBitmapFile  (in dsimple.c, p. 21; NOT Called)                                   */
/*       Calls to: bitmap_error, RootWindow, XReadBitmapFile.                             */
/*                                                                                        */
/*    Realloc  (in dsimple.c, p. 17; NOT Called)                                          */
/*       Calls to: Fatal_Error, Malloc.                                                   */
/*                                                                                        */
/*    Resolve_Color  (in dsimple.c, p. 24; NOT Called)                                    */
/*       Calls to: Fatal_Error, XAllocColor, XGetWindowAttributes, XParseColor.           */
/*                                                                                        */
/*    RootWindow (NOT defined)                                                            */
/*       Called from: Display_Window_Id, ReadBitmapFile, Select_Window, Select_Window_Args. */
/*                                                                                        */
/*    ScreenCount (NOT Defined)                                                           */
/*       Called from: Setup_Display_And_Screen.                                           */
/*                                                                                        */
/* ------------------------------ continued on next page ------------------------------- */
```

Figure 1. Pages from a Book Format Listing (continued).

this, micro-typographic factors such as blank lines, embedded spaces, type styles, and in-line comments, are used to achieve our desired principles of good micro-typographic formatting. Code sections can be separated into easily recognizable units by using blanks, beacons, alignment, and in-line comments to show the begin and end of the code sections. For example, the Pascal Const and Var sections could be delimited by placing those reserved words in boldface (or all capitalized letters) on separate lines preceded by a blank line. This is exactly analogous to section headings in a book.

Code paragraphs can be separated into easily recognizable chunks by using the same typographic factors. Blank lines can separate chunks; alignment and embedded spacing (which includes indentation) can provide spatial clues about the content of the chunks. Statements can be written as sentences (by this we are suggesting a preference of horizontal statement formatting, e.g. several statements per line, over vertical statement formatting) and character case and type styles can be used to highlight important constructs within sentences (in some languages).

All of these implementation techniques could be automated. For example, a syntax directed editor could: (1) add in-line comments indicating the end of control structures, (2) bold-face or italicize procedure calls, (3) align conditional structures (e.g. IF's and CASE's) into spatially tabular structures, (4) place blank lines before and after programming constructs that span more than a few lines, (5) highlight well-defined code segments like data declaration areas, and (6) highlight globally defined identifiers.

Organizing program source code into book format provides programmers with: (1) a familiar 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. In the next section we demonstrate, through controlled experiments and empirical studies, the value of our principles of good typographic style as encapsulated into the book format paradigm.

EXPERIMENTS

This section presents four studies that demonstrate the benefits of the book format paradigm of source code formatting. These studies show that: (1) our principles of macro-typographic style reduce maintenance effort and improve programmer performance, (2) our principles of micro-typographic style aid comprehension of both Pascal and C source code, and (3) the book format paradigm is an easy and natural representation of source code that improves programmer comprehension and performance.

Experiment 1: Testing Macro-Typographic Principles

This experiment tested our assertion about macro-typographic principles. We measured students ability to perform a maintenance task using two versions of a large Pascal program - one a traditional listing and the other a book format listing incorporating only macro-typographic features.

Materials: A line-oriented text editor program, written in Pascal, was taken from [13]. The original 1543 line program was modified by removing the Skip_Blanks procedure and the five calls to it. The Skip_Blanks procedure skips over leading blanks when the editor command input line is being parsed. The resulting modified program still worked; it was just incapable of handling free-form command inputs. The program was then reduced to 1011 lines by removing procedures unrelated to the command parsing. This was done to reduce the program to a size that could be managed by student programmers in one hour.

The modified program was then ported to Lightspeed Pascal 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 format 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.

Subjects: Fifty-three senior and graduate level computer science students volunteered to be subjects in the experiment. They were randomly assigned into two treatment groups: subjects in one group received version 1 (28 subjects), while subjects in the other group received version 2 (25 subjects). All subjects received the same instructions. No special instructions or explanations were given to subjects receiving the book format listing. This was deliberately done as a test to see if subjects could "naturally" use the book format listing (i.e. without training).

Task: Subjects were given one of the program versions, asked to read the instructions, study the code, write a Skip_Blanks procedure that would enable free-form command inputs, and indicate where (on the listing) the procedure would be called. Thus, the task was a maintenance exercise to implement free-form command processing without having any knowledge of the original Skip_Blanks procedure. In order to do this, subjects 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 recreate the Skip_Blanks procedure and its calls.

Dependent measures for each subject were: ability to write the Skip_Blanks procedure, ability to identify where it was called (5 locations), and the time required to complete the maintenance task. Subjects were given 55 minutes to complete the maintenance task.

Results: The code writing portion of the maintenance task was scored by tallying subjects' responses into four categories: (1) Skip_Blanks routines similar or identical to the one that was removed, (2) functionally correct but dissimilar Skip_Blanks routines, (3) incorrect Skip_Blanks routines, and (4) those who could not complete the task (i.e. gave up or could not get started). Results from the code writing portion (shown in Table 1) show that the book format listing group outperformed the traditional listing group by approximately two correct answers to one. Further, by adding the first two categories together (exactly correct plus functionally correct) the total correct is 52% for the book format listing versus 25% for the traditional listing. Also note that subjects in the traditional listing group were twice as likely to quit or not even start writing code.

The procedure call portion of the maintenance task was scored only for those subjects who wrote a correct Skip_Blanks procedure. Results are shown in Table 2. For the traditional listing group, the 7 subjects who successfully completed the routine correctly identified an average of 1.71 places where the procedure would be

Table 1.

Experiment 1: Code Writing Ability

	exactly correct	functionally correct	wrong	gave up or not finished
Traditional listing (n = 28)	14 % (n=4)	11 % (n=3)	36 % (n=10)	39 % (n=11)
Book listing (n = 25)	36 % (n=9)	16 % (n=4)	32 % (n=8)	16 % (n=4)

Total correct: Traditional -- 25 %
Book listing -- 52 %

Percent difference between groups..... 27 %

Table 2.

Experiment 1: Ability to Identify Procedure Calls

Dependent measure	Traditional listing	Book listing
Number writing correct procedure	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 %

called. In contrast, the average for the 13 subjects in the book format group was 2.38 correct identifications.

No significant differences in times were observed between the two groups. The average time for the traditional group was 53.5 minutes and 52.2 minutes for the book format group.

Discussion: Results from this experiments show the benefit of using the book format paradigm for macro-typographic style. We emphasize that the only difference between version 1 (traditional listing) and version 2 (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. And, it should also be emphasized that subjects using the book format listing performed better without any explanation, description, or justification of the book format listing.

Experiment 2: Testing Micro-Typographic Principles in Pascal

To demonstrate that our micro-typographic style is better than traditional methods of Pascal code formatting, we conducted an experiment comparing our book format style against traditionally formatted industrial code.

Materials: Two procedures (94 lines of code) were extracted from Borland's Turbo Pascal Toolbox. The original code from the toolbox was formatted in the traditional manner of Lightspeed Pascal. This was version 1. Version 2 was a typographic rearrangement of that code using our book format principles of micro-typographic style to guide the formatting. Specifically, section headings were highlighted, sections and control constructs were separated by blank lines, statements were written as sentences when possible, procedure calls were highlighted, and related clauses were aligned and/or chunked together. Excerpts from the two version are listed in Figure 2. We emphasize that the difference between versions 1 and 2 is entirely micro-typographic arrangement: indentation, embedded spacing, alignment, and the use of character case and type style.

Subjects: Thirty-six intermediate computer science students volunteered to be subjects in the experiment. They were randomly assigned into two treatment groups of 18 subjects; subjects in each group received one of the two code versions (traditional or book format). Both groups received the same instructions.

Task: Subjects were given one of the two code versions and asked to complete a short comprehension test using the code listing. The test consisted of 10 multiple choice and short answer questions. Some of the questions had several parts; consequently, there were 14 answers for the 10 questions. Subjects were asked to complete the test and then subjectively rate the readability of the code. Subjects were given 10 minutes to answer the questions.

Dependent measures for each subject were: score (0 to 14 points), time required to answer the questions (1 to 10 minutes), performance score (number of correct answers per minute), and a subjective readability rating on a 5 point forced-choice scale (1-very poor, 5-very good).

Results: Average scores, times, performance indexes, and ratings for both groups are shown in Table 3 and Figure 3. In support of our micro-typographic principles, all four measures --

```
repeat
  case L of
    1 :
      InputStr(FirstNm, 15, 12, 6, [CtrlZ,Tab,Enter], EndChar);
    2 :
      InputStr(LastNm, 30, 39, 6, [CtrlZ,Enter], EndChar);
  end;
  if (EndChar = Tab) or
    (EndChar = Enter) then
      L := 3 - L;
until (EndChar = CtrlZ) or
      ((EndChar = Enter) and
      (L = 1));
```

2.(a) Traditional Micro-typographic Style

```
Repeat

    Case L of
        1 : InputStr ( FirstNm,  15, 12, 6, [CtrlZ,Tab,Enter], EndChar );
        2 : InputStr ( LastNm,   30, 39, 6, [CtrlZ,Enter],     EndChar );
    end;

    If ( EndChar = Tab ) or ( EndChar = Enter ) then L := 3 - L;

until ( EndChar = CtrlZ ) or ( (EndChar = Enter) and (L = 1) );
```

2.(b) Book Micro-typographic Style

Figure 2. Experiment 2: Pascal Code Excerpts

Table 3.

Experiment 2: Results from Pascal Code Comparison

	Version	
Averages:	Traditional (n=18)	Book form (n=18)
test score	7.39	10.39 *
test time	9.31	8.90
performance(score/time)	0.81	1.23 **
readability rating	2.72	3.28 ***

* significant: (F=10.57, p<.005, d.f.=1,34)
** significant: (F= 8.57, p<.01, d.f.=1,34)
*** significant: (F= 4.45, p<.05, d.f.=1,34)

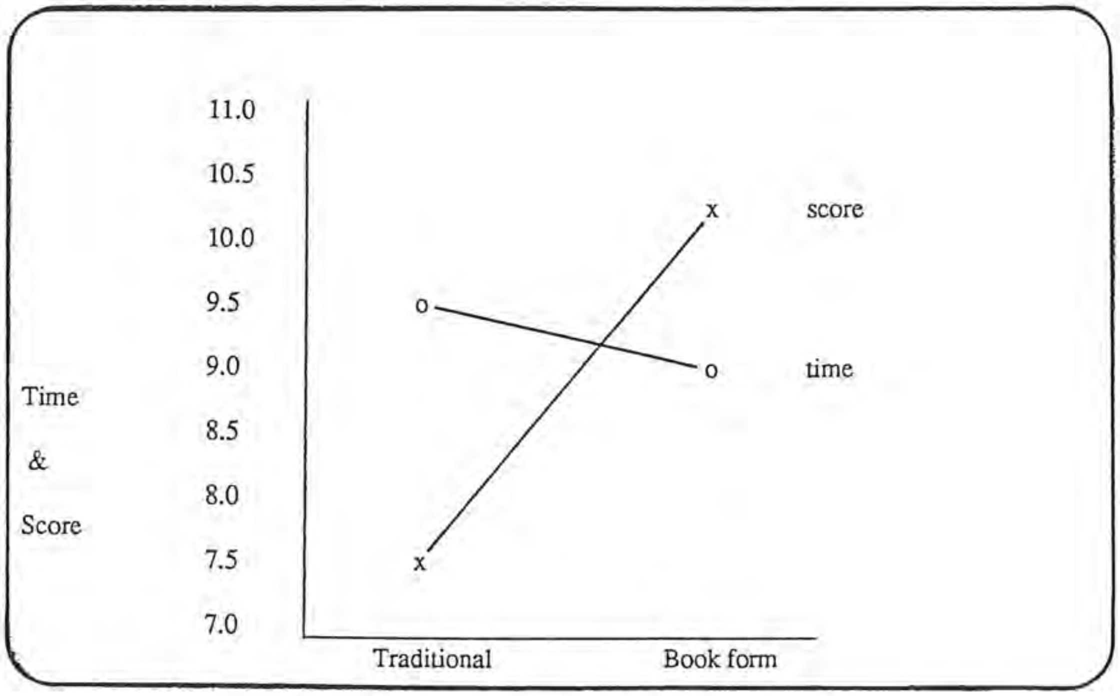


Figure 3. Scores and Times from Experiment 2.

scores, times, correct answers per minute, and ratings -- improve with the book format listing. Univariate analysis of variance showed significant differences for score (F=10.57, p<0.005, d.f.=1,34), performance (F=8.57, p<.01, d.f.=1,34), and readability rating (F=3.28, p<.05, d.f.=1,34). Although the trend is for time to improve with the book format, these differences are not strong enough to report significance.

Group difference can also be seen in score and performance ranges. Scores ranged from 3 to 12 in the traditional listing group and from 6 to 14 in the book format group. Similarly, the average correct answers per minute ranged from 0.3 to 1.40 for the traditional group and from 0.6 to 2.33 for the book format group.

Discussion: Group differences can be seen by calculating the percentage difference between them. An average score of 7.39 for the traditional listing group represents an accuracy rate of 53% while the average score of 10.39 for the book format group represents an accuracy rate of 74%. The increase in accuracy is 21%. Similar ratios for the time measure show the traditional listing group used 93% of the available time, while the book format listing group used 89%; a 4% savings in time even though they were more accurate. These comparisons demonstrate that the group working with the book format listing were more accurate and faster than the group with the traditional listing.

Experiment 3: Testing Micro-Typographic Principles in C

To further test our assertions, and to demonstrate language independence, we repeated the micro-typographic experiment using two different versions of C source code.

Materials: A reverse Polish desk calculator program written in C was taken from Kernighan and Ritchie's book, The C Programming Language [6]. Version 1 was the original code taken from the textbook. It is formatted in the traditional method of writing C source code, which is commonly used by professional programmers and is frequently referred to as the "Kernighan and Ritchie style." Version 2 was a typographic rearrangement of that code using our book format principles of micro-typographic style to guide the formatting. Excerpts from the two versions are listed in Figure 4. Again, we emphasize that the difference between versions 1 and 2 is entirely typographic: indentation, embedded spacing, alignment, and the use of boldface font.

Subjects: Forty-four advanced computer science students volunteered to be subjects in the experiment. They were randomly assigned into the two treatment groups of 22 students; subjects in each group received one of the two code versions (traditional or book format). Both groups received the same instructions.

Task: Except for the materials, the procedures used in this experiment were identical to those used in the previous experiment. The comprehension test contained 9 questions, one of which had a two part answer for a total of 10 answers.

Results: Average scores, times, performance indexes, and ratings for both groups are shown in Table 4 and Figure 5. As expected, scores, times, and number of correct answers per minute are better for the book format listing. Analysis of the data showed significant differences for score (F=9.40, p<0.005, d.f.=1,42) and performance (F=11.41, p<0.005, d.f.=1,42). Again, the trend is for time to improve with the book format, but these

```
switch (type) {

 case NUMBER:
     push(atof(s));
     break;
 case '+':
     push(pop( ) + pop( ));
     break;
 case '*':
     push(pop( ) * pop( ));
     break;
 case '-':
     op2 = pop( );
     push(pop( ) - op2);
     break;
 case '/':
     op2 = pop( );
     if (op2 != 0.0)
         push(pop( ) / op2);
     else
         printf("zero divisor popped\n");
     break;
```

4.(a) Traditional Micro-typographic Style

```
switch ( Type ) {

  case Number:  Push( AtoF(S) );        break;

  case '+':     Push( Pop( )+Pop( ) );  break;

  case '*':     Push( Pop( )*Pop( ) );  break;

  case '-':     Op2 = Pop( );
                Push( Pop( )-Op2 );     break;

  case '/':     Op2 = Pop( );
                if ( Op2 != 0.0 )  Push( Pop( )/Op2 );
                else printf( "zero divisor popped\n" );
                break;
```

4.(b) Book Micro-typographic Style

Figure 4. Experiment 3: C Code Excerpts.

Table 4.

Experiment 3: Results from C Code Comparison

	Version	
Averages:	Traditional (n=22)	Book form (n=22)
test score	6.73	7.89 *
test time	9.52	8.84
performance(score/time)	0.71	0.93 **
readability rating	3.45	3.50

* significant: (F= 9.40, p<.005, d.f.=1,42)
** significant: (F= 11.41, p<.005, d.f.=1,42)

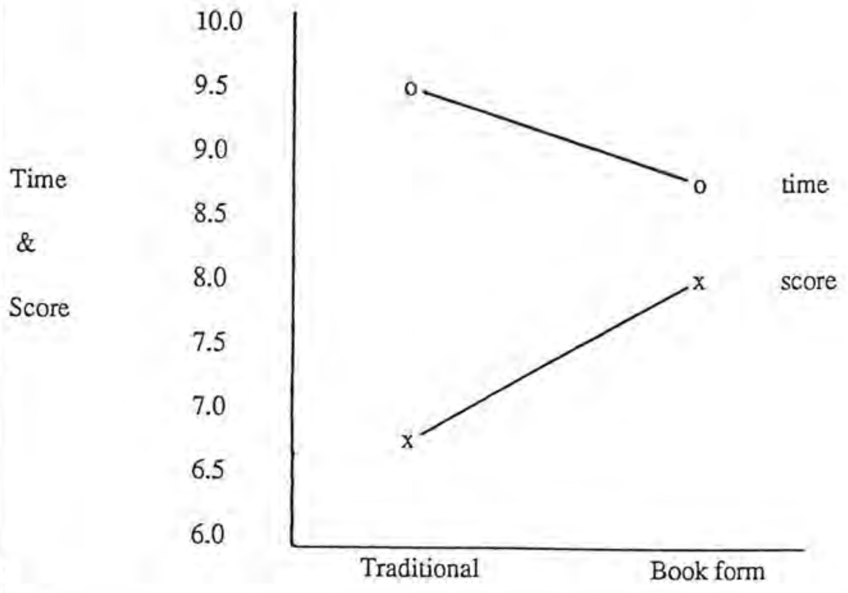


Figure 5. Scores and Times from Experiment 3.

differences are not significance.

Group differences can again be seen in score and performance ranges. Scores ranged from 4 to 9 in the traditional listing group and from 6 to 10 in the book format group. Average correct answers per minute ranged from 0.4 to 1.0 for the traditional group and from 0.6 to 1.54 in the book format group.

The very small difference between readability ratings of the two groups is interesting. We speculate that this is because subjects were informed that the code was taken from Kernighan and Ritchie's book, which is considered the model of good C programming style.

Discussion: The results from the two micro-typographic experiments provide strong evidence that our principles are compatible with programmer comprehension and, in fact, improve comprehension of small code segments. We point out that both traditional listings were formatted in well known and widely accepted styles (Lightspeed Pascal formatting and Kernighan and Ritchie's style of C). We emphasize the book format versions were obtained from simple micro-typographic rearrangements of the original code.

Experiment 4: Complete Book Format Listing

Thus far we have demonstrated the value of our book format paradigm for macro- and micro-typographic style in separate studies. In this experiment we test the complete book format listing (both macro- and micro-typographic principles) with professional programmers working on a large industrial program written in C.

Materials: A portion of the X_Windows package was obtained from an international computer firm. X_Windows is a window and mouse management system originally developed at M.I.T. and now bundled with minicomputer Unix systems. The C code we obtained was the X_Windows Information program which consists of the xwininfo.c main program file and two of its include files, dsimple.h and dsimple.c. There were 1057 lines of commented C code in the three files.

Two printed listings of the X_Windows Information program were created. Version 1 was the original as received from the company except that it was laser printed with pagination for readability. Version 2 was our typographic rearrangement of the code using the book format paradigm for both macro- and micro- typographic style to guide the reformatting. No 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 index (see Figure 1). Although the table of contents and index added 269 lines of comments to the source file, the micro-typographic rearrangement sufficiently compressed the original source code such that the resulting listing was only 41 lines longer than the original code!

Subjects: 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 twelve 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. Subjects were tested one at a time in a closed room with only the experimenter present.

Two of the subjects were deliberately chosen because they were highly-experienced programmers responsible for portions of the X_Windows system. Both were familiar with the xwininfo.c program and had previously studied the dsimple files. These two subjects represent experts already familiar with the code to be studied and were used to establish top-line performance for the dependent measures. None of the other subjects had prior experience with the code to be studied, but they did have varying degrees of Unix systems experience. Background characteristics for the subjects appears in Table 5. The subject pairs are listed in decreasing order of Unix and C experience. The first, labeled X_t and X_b, are the two X_Windows experts.

Task: Subjects were given one of the two code versions and asked to complete a three part comprehension/maintenance exercise consisting of: (1) a 30 minute study period with "Think aloud" protocols, (2) a 7 question (10 point) oral comprehension test, and (3) a pen and paper exercise to complete a call graph for the program. The test took approximately two hours and was recorded on audio tape.

As in our other experiments, the independent variable was the typographic style of the code (traditional listing and book format listing). Dependent measures for each subject were: comprehension test score (0 to 10 points), time required to answer the test questions (1 to 30 minutes), call graph score (0 to 39 points, one for each node and edge), and time to complete the call graph (1 to

Table 5.

Experiment 4: Professional Programmers' Experience

pair #	degree of X_Windows & Unix experience	subject label	yrs. prof. experience	yrs. C experience
1	X_Windows maintenance experts	X_t	9	4
		X_b	7	4
2	Unix development programmers	A_t	7	5
		A_b	8	7
3	Unix & C systems programmers	B_t	8	6
		B_b	7	5
4	Unix & C applications programming	C_t	9	3
		C_b	7	2
5	C applications programming	D_t	12	2
		D_b	10	2
6	C applications programming	E_t	13	2
		E_b	6	2

Note: Subject label subscripts denote listing version.
$_t$ for traditional listing, and $_b$ for book format listing.

30 minutes). The think aloud protocols were used as data gathering device to check for behavior patterns between and within groups.

We emphasize that all subjects received exactly the same instructions; that is, subjects working with the book format listing received no explanation or justification about it. All subjects were given a test booklet containing written instructions (and test questions) for each of the three parts.

Think aloud protocols: Subjects were given one of the program listings and told to imagine a scenario where the person responsible for maintaining this program had just quit the company and the responsibility was transferred to them. It was their task to become familiar with the program within 30 minutes because the person who just quit would be coming around to answer questions. Further, subjects were instructed to think out loud as they studied the program, so their progress could be recorded on tape.

This scenario was first used by Pennington [11] as a means of establishing the motivation to study programs in a non-goal directed manner. That is, the programmers studying the code are not addressing a specific maintenance task; rather they are trying to get a "feel" for the program in a short amount of time.

Subjects then studied the code listing, "thinking out loud" for 30 minutes. The experimenter' role was to remain unobtrusive, so as not to guide or interfere with the subjects study. The experimenter limited his interaction to answering questions about the test procedure, asking for clarification on incomplete or garbled verbalizations, and prompting the subjects about noticeable behavior changes not accompanied by verbalizations.

Protocol Results: The 12 subjects showed a variety of study patterns. All subjects:

1. Initially reported that the code seemed well structured and consistent. Later they pointed out areas where the overall organization was incorrect and the (detailed) code could be improved.

2. Had a browsing phase where they quickly scanned through the entire listing. However, this phase did not always come at the beginning of the study period. There were no recognizable group-specific browsing patterns.

3. Had distinct browsing behavior intermixed with detailed code studying throughout the study period.

4. Directed and "pruned" their analysis of the code on the basis of module name and/or location. All subjects skipped certain modules on the basis of name (e.g. "Get_Error, I can ignore that.") or location (e.g. "Oh, this is the dsimple include file, I don't need to look at these.").

5. Studied the main program and its support routines prior to studying the include files.

6. Scattered or separated the code listing into various piles, stacks, and groups. The most common ordering was distinct piles for the main program body, the support routines, and the include code.

Differences between the version 1 group (traditional listing) and version 2 group (book format listing) were that subjects receiving the book format listing:

1. Noticed and commented on the "documentation" provided by the table of contents and module index. All traditional listing subjects expressed a desire to have a cross reference map, while the book format listing subjects commented that it was a "luxury" to have one included in the code.

2. Made extensive use of the table of contents and module index while browsing and studying detail.

3. Maintained separate piles of code pages with the index in one pile and the table of contents face-up off to the side.
4. Noticed and commented on the use of italics and bold-face to highlight module names (e.g. "Gee, that's a great idea. I wish I could do that."). Interestingly, four of the six programmers receiving the traditional listing went through

and highlighted module names by underlining and/or circling them.

5. Noticed and commented on the use of horizontal statement spacing rather than vertical (e.g. "See here, it's not all strung out the way Kerrigan [sic] and Ritchie do it.").

6. Commented that the code was better written than their own. None of the traditional listing subjects made this claim.

Oral Comprehension Test: For this part of the experiment subjects were given a written instruction page containing seven questions. Three of the questions had two-part answers, so there was a total of 10 points on the test. The questions ranged from high-level general questions to low-level specific questions.

The questions were read out loud with the subjects verbally answering each question before going on to the next. Subjects were allowed to keep the written instruction sheet and could refer to it as often as desired. The experimenter's role was limited to reading the questions and prompting for more detail (if necessary) to achieve either a clearly right answer or a clearly wrong answer. When a definite right or wrong answer was obtained the experimenter responded with "O.K." and moved to the next question. No feedback was given on the correctness of answers. The oral comprehension test was timed from the reading of the first question to the answering of the last. Hence, the independent variable for the task was the listing version and the dependent variables were score (0 to 10 points) and time (1 to 30 minutes).

Comprehension Test Results: The comprehension test results are presented in Figure 6. As can be seen, programmers working with the book format listing scored higher, and did so faster, than the programmers working with the traditional listing. There were no

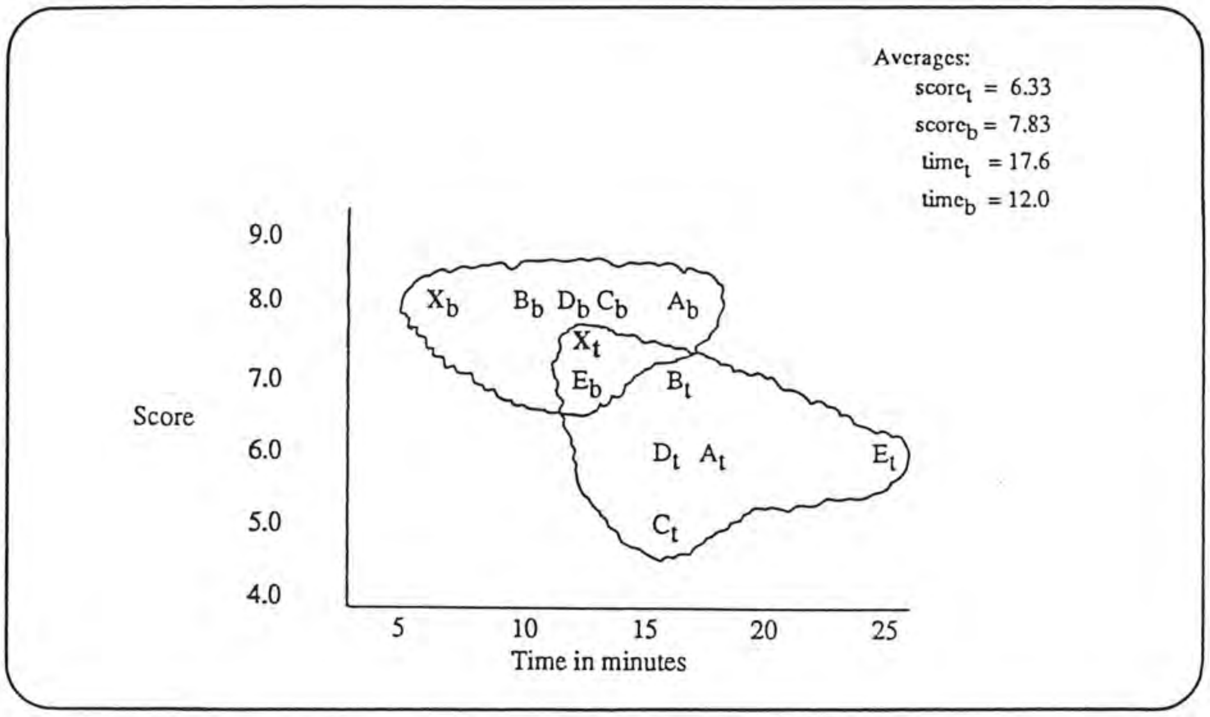


Figure 6. Scores and Times from Experiment 4 Comprehension Test.

noticeable patterns in the ability to answer specific questions.

Note that there is little difference between the two experts and that they represent the top-line performance. Also note that all other subjects working with the book format listing performed nearly as well as the two experts, but none of the subjects working with the traditional listing did. The clear separation between the groups reflects the improved comprehension afforded by the book format listing.

Call Graph Exercise: For the third task, subjects were given a written instruction page and an incomplete call graph and were asked to complete the call graph. Prior to the task the experimenter read the instructions out loud and traced through the code for the main routine, pointing out the relationship between the code and the incomplete call graph. All subjects reported that they understood the exercise and the call graph before beginning the task.

The incomplete call graph consisted of 12 nodes and 11 edges; it represented the top-level calls from main to its support routines. The completed call graph had 23 nodes and 39 edges, so the task was to find and add the missing 11 nodes and 28 edges.

The experimenter's role was limited to reading the instructions, tracing through the incomplete call graph, and indicating when a node was a dead-end because its code was not included in the listing. (The program makes a number of calls to various X_Windows libraries.) The call graph exercise was timed from when the subjects started looking at the code to when they indicated that they were finished. Score for the exercise was the total number of new nodes and edges they successfully added to the call graph. Hence, the independent variable was the listing version and the dependent variables were score (0 to 39) and time (1 to 30 minutes).

Before starting the exercise, five of the six subjects working with the version 2 listing indicated that they could use the index to complete the call graph without looking at the code. They were told it was an exercise in code reading and they were to build the call graph from the code, not the index. They were permitted to use the index and table of contents only to find modules when tracing the execution of the code they were reading.

Call Graph Results: The call graph exercise results are presented in Figure 7. As can be seen, except for the two experts, the programmers working with the book format listing scored higher, and did so faster, than the programmers working with the traditional listing. On the average, subjects working with the traditional listing missed twice as many call graph connections and took one minute longer, that those working with the book format listing. There is little difference between the experts. Again subjects working with the book format listing performed as well or better than the experts and those working with the traditional listing performed noticeably worse.

Discussion: As a group, the programmers working with the book format listing outperformed those working with the traditional listing. Further, all subjects in the book format group performed as well as or better than the two expert programmers already familiar with the code. This is in sharp contrast to the subjects working with the traditional listing who performed noticeably worse that the two experts. In every matched pair the subject working

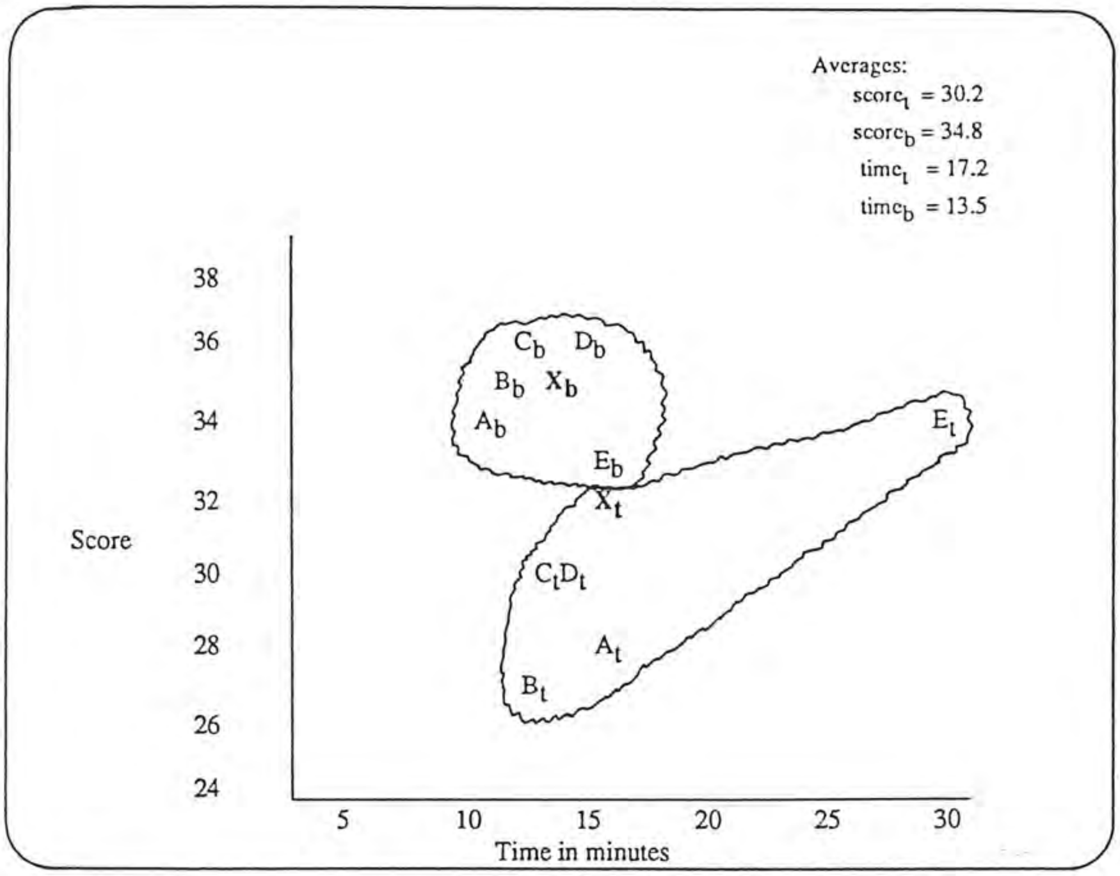


Figure 7. Scores and Times from Experiment 4 Call Graph Exercise

with the book format listing scored better and worked faster than those working with the standard listings.

CONCLUSIONS

This paper identifies principles of typographic style and demonstrates through empirical studies that typographic characteristics significantly impact program comprehension. Typographic style can provide visual clues to the underlying structure of the code and can support a variety of code access strategies.

The book format, an implementation of these typographic style principles, was introduced. Our four experiments with the book format for source code formatting show that: (1) our macro-typographic implementation aids in maintenance tasks on large programs, (2) our micro-typographic implementation aids in the comprehension of Pascal and C code segments, and (3) professional programmers can benefit from the book format model. Furthermore, the book format model seems very natural and convenient to use because it takes advantage of the subjects' familiarity with structure and organization of a book. Also note that subjects given the book format listings were not given any introduction or instructions on how to use it.

Our research has direct application to code formatters such as prettyprinters and syntax-directed editors. These code formatters are designed to improve the readability of source code by formatting it in a consistent manner. However, present-day code formatting tools follow different sets of source code layout rules which are selected by the developer. These rules reflect the subjective judgement of the developer and are rarely (if ever) supported by empirical evidence showing that they aid program comprehension. Our research provides the principles and a foundation for formatting rules that aid program comprehension.

Our research also has application in programming language design and programming teaching practices. In language design, programming language constructs should be compatible with the way programmers view code. (For instance, syntactic constructs can be beacons or unnecessary distractions.) In programming instruction, the teaching of specific style rules may be counterproductive. While there may be no single best way to format a source program, teaching beginning programmers language independent typographic style principles (rather than specific layout rules) will make them aware of the purpose of particular style rules. This will provide them with a solid basis for the source program layout guidelines they choose.

References

1. Adelson, B. Problem solving and the development of abstract categories in programming languages. Memory and Cognition 9, 4 (1981), 422-433.

2. Baecker, R. Enhancing program readability and comprehensibility with tools for program visualization. Proceedings 10th International Conference on Software Engineering, Singapore, 1988, 356-366.

3. Brooks, R.E. Towards a theory of the comprehension of computer programs. Int. J. of Man-Machine Studies, 18 (1983), 543-554.

4. Fjeldstad, R.K. and Hamlen, W.T. Applications program maintenance study: Report to our respondents. Proceedings GUIDE 48, Philadelphia, PA, 1979.

5. Kernighan, B.W. and Plauger, P.J. The Elements of Programming Style. McGraw-Hill, New York, 1974.

6. Kernighan, B.W. and Ritchie, D.M. The C Programming Language. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

7. Ledgard, H. and Tauer, J. Professional Software: Volume II Programming Practice. Addison-Wesley, Reading, MA, 1987.

8. Littman, D., Pinto, J., Letovsky, S., and Soloway, E. Mental models and software maintenance. Empirical Studies of Programmers, (E. Soloway & S. Iyengar, editors), Ablex Publishing, Norwood NJ, 1986, 80-98.

9. Love, T. An experimental investigation of the effect of program structure on program understanding. Proc. ACM Conference on Language Design for Reliable Software. March 1977, 105-113.

10. Miara, R.J., Musselman, J.A., Navarro, J.A. and Shneiderman, B. Program indentation and comprehensibility, Comm. ACM 26, 11 (Nov. 1983), 861-867.

11. Pennington, N. Stimulus structures and mental representations in expert comprehension of computer programs, Cognitive Psychology 19 (1987), 295-341.

12. Oman, P. and Cook, C. R. A programming style taxonomy. Technical Report 88-60-20, Computer Science Department, Oregon State University, Corvallis, Oregon 97331. (Submitted for publication)

13. Schneider, G. and Buell, S. Advanced Programming and Problem Solving With Pascal. John Wiley & Sons, New York, 1981.

14. Sheil, B.A. The psychological study of programming, _Computing Surveys_ 13, 1 (March, 1981), 101-120.

15. Shneiderman, B. and McKay, D. Experimental investigations of computer program debugging and modification. _Proc. 6th International Congress of the International Ergonomics Association_. July 1976, College Park, MD.

16. Soloway, E. and Ehrlich, K. Empirical studies of programming knowledge. _IEEE Transactions on Software Engineering_ SE-10, 5 (Sept., 1984), 595-609.