AN ABSTRACT OF THE THESIS OF

Paul W. Oman, Jr. for the degree of Doctor of Philosophy in Computer Science presented on December 12, 1988.

Title: A Taxonomic Analysis of Typographic Programming Style

Abstract approved: ___Redacted for Privacy___
Curtis R. Cook

Program comprehension is important in program testing, debugging, and maintenance. Programming style impacts program understanding. However, there has not been any systematic identification of individual style factors and their contribution to program comprehension.

In this thesis we present a programming style taxonomy composed of three classes: typographic (program layout and commenting), control structures, and information structures. Each class is further subdivided into macro (whole program or system) and micro (module or statement) subclasses. The taxonomy reveals many conflicting and unsubstantiated rules in collections of style rules from various publications on programming style. Further, it provides plausible explanations for some of the inconsistent results in programming style research.

This thesis concentrates on the isolation of typographic style factors and analysis of their affects on programmer comprehension. General principles of good macro- and micro-typographic style are identified and the "book paradigm," a mechanism for implementing the principles, is presented. Four experiments, involving both student and professional programmers, demonstrate that the macro- and micro-typographic principles incorporated into the book paradigm significantly improved program comprehension and maintenance. These results have direct application to programming language design and programming tools such as pretty-printers, language directed editors, style analyzers, and source code control systems.

A Taxonomic Analysis of Typographic Programming Style

by

Paul W. Oman, Jr.

A THESIS
submitted to
Oregon State University

in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Completed December 1988
Commencement June 1989

APPROVED:

Redacted for Privacy

Professor of Computer Science in Charge of Major

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented December 12, 1988

Typed by Paul Oman

ACKNOWLEDGEMENTS

I would like to express my appreciation to each member of my committee for their time and patience. To Curt Cook, my major advisor, thanks for sticking with me when I got off track. Your insistence on excellence is responsible for the quality and thoroughness of this thesis. To Ted Lewis, Editor-in-Chief of *Software,* your criticism of my proposal is responsible for the (re)organization of this thesis. Also, thank you for giving me the opportunity to get involved with *Software.* To Toshi Minoura, your probing questions are responsible for most of Chapter 3, which is the glue that holds this thesis together, thank you. To Walter Loveland, my graduate representative, thank you for your positive support and assistance. And last, to Donna Cruse, who has guided me without knowing it since 1970, thank you for giving me a strong foundation in science. Your influence can be seen throughout this thesis.

I would also like to express my appreciation to the people who have aided and supported this research. From the University of Idaho I have received much support from William Saul, Dean of the College of Engineering, Richard Jacobsen, Associate Dean, and John Dickinson, Chair of the Computer Science Department . At O.S.U., thanks are due to Lyle Calvin, Dean of the Graduate School, and Walter Rudd, Chair of the Computer Science Department. Thank you all.

Finally, I would also like to express my thanks to my family for their patience and support. You have made it a joy to "come home." My success is founded in your love and encouragement. Thank you.

Table of Contents

Appendices

List of Figures

Figures Page

List of Tables

Table Page

A Taxonomic Analysis of Typographic Programming Style

Chapter 1

Introduction

1.1. What is "programming style"?

Programming style is an intuitive and elusive concept. It is highly individualistic and easily recognized, yet difficult to define or quantify. The goal of programming style is to make a program clear, easy to understand and thereby easy to work with. The following illustrate the diversity in attempts to define programming style.

> "The 'elegance' or 'style' of a program is sometimes considered a nebulous attribute that is somehow unquantifiable; a programmer has an instinctive feel for a 'good' or 'bad' program ..." -- [Berr85,p.80]

> Programming style ... "refers to the entire set of conventions, guidelines, aids and rules that make computer programs easier for people to read, work with and understand." -- [Schn81,p.52]

> "Programming style brings to mind the ways that a creative programmer brings clarity, maintainability, testability, reliability, and efficiency to the coding of a module." -- [Arth85,p.173]

These definitions reflect different views of programming style and, although diverse, they are not at odds with dictionary definitions of style:

> "1. a way of speaking or writing, one characteristic of an individual, period, school, or nation; 2. the custom or plan followed in spelling, capitalization, punctuation, and typographic arrangement and display; 3. the manner or method of acting or performing as sanctioned by some standard; 4. a distinctive or characteristic manner." -- [Merr74]

Programming style shares many similarities with literary writing style. For example, in his book on technical writing Chael Markel lists word choice, sentence construction, and paragraph structure as elements of literary style [Mark84,p.89]. However, effective writing is more than observing established conventions for spelling, grammar, or sentence structure. It is the "perception and judgment" a writer exercises in selecting "from equally correct expressions, the one best suited to his material, audience, and intention" [Birk59].

Several programming style books have rules based on English literature style rules. For example, the form and approach used in Kernighan and Plauger's *The Elements of Programming Style*, [Kern74] is based on Strunk and White's *Elements of Style* [Stru59].

In this thesis we define "programming style" as analogous to "writing style" and, as such, are concerned with the product of programming rather than the process of programming (i.e., we are not considering "styles of programming"). The taxonomy presented in this paper is an operational definition for programming style; it includes definitions and principles of programming style.

1.2. The problems in programming style research.

Programming style rules are indefinite, inconsistent, and contradictory. Differing and contradictory rules governing the use of style factors such as indentation, alignment, commenting, and control structure can be found in several books and articles [Bate81, Crid78, Gilm84, Grog79, Hans85, Kern74, Marc81, Pete77]. Correspondingly, source code analyzers use different sets of style characteristics and weights [Berg84, Berr85, Meek83, Redi86, Rees82], and automated code formatters use different style conventions [Alsy86, Bate81, Crid78, Hans85, Logi87, Thin84, Thin86, Wint88]. None of these style books, style analyzers, and code formatters have a strong theoretic or empirical basis for their rules and methods of code formatting.

This lack of consensus can be traced to the lack of definition of exactly what programming style is, and what style factors contribute to successful programming style. Every author on programming style gives an idiosyncratic definition of style; examples can be found in virtually all introductory programming texts as well as specific programming style guides [Arth85, Berr85, Coop85, Fair85, Kern74, Ledg78, Ledg87, Schn81, Tass78]. While all agree that the major purpose of good style is to improve code readability, none provide evidence that their particular style improves readability. They give style rules and supporting examples for specific implementations, but rarely discuss general principles governing the application, or benefits of their rules over other sets of rules.

While everyone seems to recognize the importance of programming style, little is known about the affects of programming style on programmer comprehension and program maintenance. Many results from existing studies are contradictory and/or indefinite. For example, contradictory evidence on the value of indentation and mnemonic variable names (two relatively simple style factors) can be found in [Gilm84, Shei81, Wei74a, Wei74b]. Some studies report no differences in comprehension of indented versus non-indented code, while others report no comprehension differences between meaningful and nonsense variable names. These results have led some researchers to question the value of code formatting and naming conventions (see [Shei81, p.109]). Such problems in program comprehension research can be traced (in part) to improper experimental methodology [Broo80, Shei81]. For example, there are numerous studies where style factors -- that were not the focus of the study -- were implicitly changed in experimental materials. These uncontrolled changes may account for the discrepancies in their results (for examples see [Gilm84, McKe81, Norc78, Norc82, Shep79, Wei74a, Wei74b]).

1.3. Rationale for a taxonomic analysis of style.

We suggest that this disorganization in programming style stems from the subjectivity introduced in style guidelines, style analyzers, and code reformatters; and from the conflicting results obtained through inadequately controlled studies in style and program comprehension. We recommend that research in programming style needs to follow a strict and orderly paradigm in order to reduce, if not eliminate, the disparity now found in programming style literature.

Since the purpose of employing programming style standards is to promote clarity and simplify structure, stylistic factors should be investigated with a view toward determining how specific style factors affect comprehension and complexity. Hence, there are five stages of programming style analysis:

1. Identifying and categorizing the factors comprising programming style.

2. Determining the affects of these stylistic factors on program complexity and programmer comprehension.

3. Establishing principles of good style and subsequent style standards (relative to local constraints) based on the results from 2.

4. Gauging the adequacy and completeness of the programming style standards.

5. Measuring the degree of conformity between source code and style standards.

When existing programming style research is viewed in light of this five stage paradigm it is obvious why there is much disarray in the field. Style guidelines attempt to address stage 3 without proper attention to stages 1 and 2; style comprehension studies try to measure effects of style factors without carefully defining and categorizing style factors. Note that only the last stage suggests the need for automated style analyzers, and yet this is the area where much work has been concentrated. Also note that syntax-directed editors and pretty-printers operate on formatting rules for arranging code, although there are no established standards for doing so.

It appears that style researchers are attempting to measure style and assess its impact prematurely. Without a systematic effort to define and isolate style factors it is difficult (if not impossible) to adequately control comprehension studies. Establishing standards without understanding the effects on comprehension and complexity is counter-productive. Assessing style without tangible standards is a subjective judgement open to debate.

1.4. Scope of this thesis.

This thesis is an implementation of the first three stages of our programming style research paradigm: (1) a taxonomy identifying and categorizing programming style factors, (2) an analysis of the affects of typographic style characteristics on program comprehension, and (3) formulation and testing of some principles of typographic programming style.

Establishing an entire set of principles for good programming style is too large an undertaking for a single study. Our taxonomy partitions programming style into three orthogonal dimensions: typographic style, control structure style, and information structure style. Each of these dimensions is further divided into macro- and micro-characteristics. (A complete definition and justification of this taxonomy is given in Chapter 3.) In this thesis we concentrate on typographic programming style. The typographic style dimension was selected because we wanted to (1) disprove published claims that typographic factors were "merely cosmetic considerations," [Berr85]

(2) demonstrate that typographic style significantly affects programmer comprehension and impacts program maintenance, and (3) show that existing code formatting tools are inadequate and can actually interfere with program comprehension.

Typographic style factors are concerned with the typographic layout of source code and its associated in-code documentation (comments). This includes such factors as indentation, the use of blank lines, naming conventions, commenting conventions, etc. By definition, these factors affect only the physical layout or format of the source code and do not, in any way, affect the performance of the code. Macro-typographic style pertains to intermodular layout, naming, and commenting; micro-typographic style pertains only to intramodular layout, naming, and commenting. We demonstrate that these factors are more than cosmetic and can significantly affect programmer comprehension. From our analysis of typographic factors we developed a set of principles guiding their use. These principles are consistent with theories of programmer comprehension and could be incorporated into code formatting tools to improve program readability and maintainability.

1.5. Organization of this thesis.

Chapter 2 contains an overview of the four major areas in programming style research: style guidelines, style analyzers, code formatters, and comprehension studies. We critically examine the literature in each area and point out instances of subjectivity, inconsistency, and conflicting results commonplace to this type of research.

In Chapter 3, we introduce our style taxonomy. The taxonomy was formulated by systematically analyzing existing style publications and grouping functionally equivalent objectives and factors into an organized hierarchy. The completeness and robustness of the taxonomy was tested by fitting style rules from seven different publications into the taxonomy. We then demonstrate the utility of the taxonomy by identifying unsubstantiated and conflicting rules.

Chapter 4 contains the results of our preliminary experiments demonstrating the value of analyzing style in light of the taxonomy. Using the knowledge gained from organizing the taxonomy, we were able to: (1) define and implement measures of typographic style useful in style analysis, (2) identify and explain the comprehension differences caused by different typographic arrangements of IF-THEN-ELSE's, and (3) demonstrate the inadequacy of existing code formatters.

In Chapter 5, we then show why existing typographic formatting is incompatible with mechanisms of programmer comprehension and maintenance activity. After briefly reviewing the relevant literature on comprehension and maintenance we suggest that programmers need a better method of code presentation. We formulate a minimal set of rules for typographic style that are consistent and compatible with theories on programmer comprehension and maintenance.

In Chapter 6, we implement the principles of good typographic style outlined in Chapter 5. We introduce our "book paradigm" for source code formatting. Modeling source code after a book permits programmers to view the code through a variety of access paths and provides them with typographic clues enabling easier and faster comprehension. Thus, it facilitates a variety of comprehension strategies and maintenance activities.

Chapter 7 contains the results of our experiments testing the book paradigm for typographic formatting. In a controlled maintenance exercise involving 63 subjects we show that our book model of macro-typographic formatting results in a 13 to 27 percent improvement in programmers' ability to perform a routine maintenance task. In two controlled studies using 80 subjects we show our micro-typographic implementation results in a 12 to 21 percent improvement in comprehension with an associated 4 to 7 percent reduction in time. Further, in an experiment involving 12 professional programmers, we demonstrate that even highly experienced programmers can benefit from the book paradigm. In a comparative study of six matched pairs of programmers, we show that programmers working with the book paradigm outperform comparable professionals working with traditional listings.

In the conclusions and discussions of the last chapter we (1) review the contributions of this research, (2) point out how our results can influence language and tool development, and (3) list topics for further work in programming style.

Chapter 2

An Overview of Programming Style Research

2.1. Studies on programming style.

The literature on programming style can be divided into four areas: (1) the publication of programming style guidelines, (2) the development of automated program style analyzers, (3) the development of code formatters such as pretty-printers and syntax directed editors, and (4) investigations into the affects of program style on program comprehension.

In this chapter we critically review each area to point out weaknesses, inconsistencies, and contradictions found in the literature. The common thread, evident throughout the four areas, is the lack of definition and objectivity in the analysis of style.

2.2. Style Guidelines.

Publications on good programming style usually present a set of style rules and guidelines [Kern74, Ledg78, Ledg87, Marc81, Tass78]. These style rules and guidelines are based on analogous writing style guidelines for English; Kernighan and Plauger's *The Elements of Programming Style* is the best known of these books. Some of these style books are for programming in general, while others are oriented toward specific programming languages. Many introductory programming texts also contain short sections on programming style.

Developers of guidelines on programming style often have difficulty defining style, but they traditionally have not had any problem in identifying style factors that influence readability and understandability. Among the many listed are: indentation, line length, commenting, blank lines, embedded spacing, modularity, usage of programming constructs, meaningfulness of identifiers, execution flow, module length, code reusability, economy and simplicity of code, use of constants and literals, methods of type and data declarations, use of library functions, level of nesting, control flow, information flow, operator and operand counts, and other measures of program complexity.

Many of these style factors are easily quantifiable (e.g., average line length), while others are simply stated as common sense rules (e.g., avoid tricky code). Usually, widely accepted rules of good programming style are goal directed; one rule may stress code efficiency while another addresses code readability. Often these rules are at odds with each other; but, for the most part, the underlying purpose is to produce code that is clear and easily understood without sacrificing performance.

On those rules where there is some consensus on the general principle behind the rule, there is often much disagreement about how the principle should be applied. For instance, all authors advocate using indentation to highlight control structures, but few agree on how to do it. As such, many of these style factors are difficult to assess except by means of an operational approximation. For example, Kernighan and Plauger list the following rules of good program style: (i) Choose variable names that won't be confused, and (ii) Use variable names that mean something.

While no one will dispute the value behind the intent of these rules, it is clear that compliance is a matter of value judgement. For example, consider a program that calculates an arithmetic mean; the identifier used to store that value could be MEAN, AVERAGE, XBAR, or any possible combination of letters and digits with permutations of upper and lower case and an underscore, depending on language syntax. The "meaningfulness" of the identifier is dependent upon application domain, programmer expertise, and non-enumerable contextual dependencies.

Many of the published style rules are actually suggestions for good programming practice. For example, rule 35 from [Kern74] suggests, "Don't stop at one bug," while rule 43 advises the reader to "Check some answers by hand." This failure to distinguish between style rules and programming practices can be found in virtually every published discussion on programming style.

Furthermore, authors invariably choose stylistic factors commonly thought to influence style without supporting evidence demonstrating the importance or effect of that characteristic on program comprehension and maintainability. The only supporting evidence offered are small example code segments illustrating "improvements" resulting from application of their style rules. How these improvements aid comprehension is usually not discussed and often not apparent. Typically their rules are general, and occasionally contradictory, with no guidelines on how to resolve conflicts between rules.

2.3. Automated style analyzers.

Most of the research in program style analysis has concentrated on the development of automated code analyzer programs. Automated style analysis programs for Pascal [Rees82], C [Berr85], and FORTRAN [Redi86] (among others) have been developed. These "style grading" programs collect counts of various program characteristics thought to represent programming style and compute a style score, usually between 0 and 100, as a weighted sum of the factors. The counts used to compute the sum are either fixed or specified by the user. These measures of style characteristics are usually crude metrics for widely accepted style rules. For example, average identifier length is often used as a measure of variable "meaningfulness," on the assumption that longer identifiers are more meaningful than shorter ones.

One of the first automated style analyzers was proposed by Rees [Rees82] who described a Pascal source code style grader based on ten factors: average line length, comments, indentation, blank lines, embedded spaces, modularity, variety of reserved words, identifier length, variety of identifier names, and the use of labels and Gotos. Each factor was operationally approximated (by simple counts and averages) and assigned a positive or negative weight. The weighted factors were then summed into a percentage score. The weights and trigger-points (maximum and minimum values that trigger the assignment of weighted scores for each factor) were established by adjusting the parameters until the automated style analyzer awarded 'A' grades to programs considered to be good.

Rees' style analysis methodology was implemented on a Unix system by Rosenthal, who distributed his source code through the *ACM SIGPLAN Notices* correspondence column [Rose83]. His list of style factors is the same as Rees', except for subtle differences in how the measures are calculated and the omission of the "variety of identifiers" metric. Meekings later published a modified version of the Pascal style checker that used the same style factors [Meek83]. Berry and Meekings then adapted this style analyzer to work on C source code [Berr85]. The Berry and Meekings style analyzer calculates measurements on module length, identifier length, comments, indentation, blank lines, line length, embedded spaces, constant definitions, reserved words, included files, and goto statements as factors contributing to their style measure. These are the same factors used by Rees except the "variety of identifiers" measure was replaced by counts of included files and the "percentage of constant definitions" measure has been added. Other minor changes were made to the manner in which the metrics were calculated, but for the most part, the Berry and Meekings style analyzer perpetuates the same style assessment methodology established by Rees.

In their implementation of a FORTRAN source code style analyzer, Redish and Smyth [Redi86] used 33 factors for the automated evaluation of students programs. The 33 measures can be grouped as follows: commenting (4), indentation (1), block sizes (2), statement labels and formats (7), counts of names and statements (6), array declarations (2), control flow and nesting measures (7), blank lines (1), operator count (1), operand count (1), and parameterization (1). Their AUTOMARK program, similar in purpose to those described by Rees and Berry and Meekings, calculates a score and percentage for each of thirty factors that are summed into a final score. Additionally, their ASSESS program is used to obtain nonnumeric evaluation of ten style factors plus comments on indentation, commenting, and label usage.

Although the list of factors used by ASSESS and AUTOMARK represents the most complete assessment of style used to date, the scoring system is primarily an adaptation of the Rees maximum and minimum trigger-point methodology with factor weights assigned by the instructor. Again, each factor is operationally approximated through simple counts, averages, and percentages.

The choice of factors and corresponding weights in automated style analyzers is entirely subjective. For each factor there is an arbitrarily established range (not necessarily continuous) designed to reflect the "goodness" of a measure's value. Furthermore, program characteristics that are not easily quantified are operationally defined and approximated. This collection of measures is then tallied into a single style score. All the existing code analyzers combine measures of layout characteristics (e.g. indentation, blank lines) with those of structure (e.g. control flow, modularity). While some of these studies have informally recognized the difference in importance between categories of style factors, none have attempted to evaluate the groups of factors separately.

Furthermore, there is no clear understanding of the relationship between program style and the complexity and/or maintainability of code. For example, one would expect some relation between good programming style and program errors; but Harrison and Cook [Harr86] found no relation between Berry and Meeking's style score [Berr85] and the error proneness of a collection of C modules. And, Evangelist [Evan84] demonstrated that application of Kernighan and Plauger's style rules had unpredictable affects on five common software complexity metrics. For each complexity metric he gave an example where application of a style rule increased complexity. His results contradict later claims by Arthur [Arth85,p.175] that applying Kernighan and Plauger's rules always reduces complexity.

2.4. Code formatters.

Many code formatters (pretty-printers and syntax directed editors) have been proposed and/or implemented [Bate81, Crid78, Hans85]. Pretty-printing is essentially a typographic rearrangement of source code in order to make it consistent. This effort may be combined with a diagnostic tool (such as Redish and Smyth's ASSESS program [Redi86]), but invariably these systems operate as post-coding reformatters. Syntax directed editors, on the other hand, are code formatters that operate upon source code entry. Macintosh Pascal, an editor/interpreter for the Apple Macintosh [Thin84], dynamically formats all Pascal code upon input while performing its syntax checking operations. Many of the latest microcomputer compiler environments include syntax-directed editors. Alsys Ada [Alsy86] and Logitech Modula 2/86 [Logi87] are two examples.

In addition to improving code readability, a major purpose of code formatters is to provide consistency in the typographic layout of the source code being formatted. Pretty-printers enforce this consistency only upon invocation; code changes made after pretty-printing require another pass through the formatter. In contrast, syntax directed editors maintain typographic consistency throughout the maintenance effort.

However, there is no commonly accepted method of formatting source code to achieve the goals of readability and consistency. For example, Winter [Wint88] illustrated the differences in indentation methods used by three pretty-printers [Bate81, Grog79, Pete77]. He also argues that internal consistency is difficult to gauge, noting that a programmer may consistently indent four spaces within an IF construct, but five spaces within a WHILE construct.

Clearly, the problematic issue with code formatters is identical to that of style guidelines and style analyzers. Namely, that the actions of code formatters are based primarily on the authors' personal preferences for typographic arrangement. Winter suggests that the types of typographic style, "range from the expected to the slightly bizarre." From his review of existing pretty-printers, he concludes:

> "For the most part, the originator of the style offers little support that his/her style rules improve readability. No experimental or empirical data supporting the choice of a particular set of style rules is mentioned."

2.5. Programmer comprehension studies.

Studies investigating the impact of specific style factors on program comprehension are concerned with issues of readability and maintainability. The accepted paradigm for this type of research is controlled administration of a comprehension measure across treatment groups receiving a variety of program styles. But gauging comprehension is a difficult task and, consequently, there are many types of program comprehension measures: verbatim recall, functional recall, debugging tasks, modification tasks, Cloze exercises, and comprehension quizzes [Broo83, Cook84, Mohe81]. The comprehension quiz, or test, is probably the most widely used and commonly accepted method [Cook84]. It usually consists of several short-answer, multiple-choice, or true/false questions pertaining to the program structure and function.

The indentation study by Miara, et al, [Miar83] is a good example of a program comprehension experiment using a comprehension quiz as a data gathering instrument. They measured students ability to read and understand programs that were indented using either 0 space (no indentation), 2 space, 4 space, or 6 space indentation. Subjects were randomly divided into groups and given a 102 line Pascal program and a quiz sheet containing 13 questions (multiple-choice and short essay) about the program. Each group received one of the four indentation levels. The subjects were given 20 minutes to answer the questions while looking at the program. Results from the test show that the groups working with the 2 and 4 space indented programs outperformed the other two groups.

The experimenters concluded that two to four space indentation was optimal, while six space indentation actually interfered with comprehension.

Another exemplary study was conducted by Woodfield, Dunsmore, and Shen [Wood81], who studied the affects of comments and modularity on program comprehension. Their results show that comments and method of modularizing both affect students' ability to comprehend a program as measured by a 20 question objective test. In a later study, Dunsmore [Duns85] found differences in students' ability to understand monolithic, modular, and overly- modularized FORTRAN programs. Subjects scored significantly higher on a comprehension quiz after studying the moderately modularized program than on the other two versions.

The above studies are cited as examples demonstrating the utility of studying the impact of specific stylistic factors on program comprehension. Controlled administration of a comprehension test across treatment groups receiving a variety of program styles is the most appropriate methodology for this type of work. All comprehension experiments reported in this thesis are modeled after the experiments conducted by Miara, et al.

However, results from many previous studies have been clouded by improper methodology [Broo80]. For example, Weissman's attempt to study three different factors (comments, paragraphing, and mnemonic variable names) in two different programs lead to results that were either insignificant or highly suspect [Wei74a, Wei74b]. (His results, for example, suggest that using meaningful variable names has no affect on program comprehension.)

We suggest that these and other such studies have inadequately controlled stylistic variants within and across treatment groups, and therefore have had difficulty establishing the true cause of observed differences. Experiments where the control of indentation is suspect, for example, can be found in [Gilm84, McKe81, Norc78, Norc82, Shep79]. These studies contain uncontrolled changes in indentation, caused by the manipulation of another independent variable (e.g., chunking, commenting, changes in control structures, etc.). The danger is that this lack of control may lead researchers to inappropriate and/or unsupported conclusions. An example can be seen in Shneiderman's book *Software Psychology*, where he reviews the early work on indentation and suggests that its value (positive affect on comprehension) has been overrated [Shne80, p.74]. This conclusion was generally accepted until Miara's later findings demonstrated the positive affect of 2 to 4 space indentation.

2.6. Conclusions.

All four areas of style research have two common traits that affect the results. First, the lack of definition on what constitutes style. For example, precisely what is a style factor and what is a programming practice? And second, the guidelines and materials used in all four areas are essentially micro-style examples and exercises. That is, style guides exemplify their rules with very short code segments, code analyzers are tested and tuned with only short student programs, code formatters never address module reorganization or programming "in-the-large," and comprehension studies usually limit their analyses to short programs (less than 200 lines). None of this addresses the need to establish style standards that aid comprehension of very long or complicated programs.

As a consequence, there are several problems that appear throughout programming style research: lack of definition, consensus, and experimental control; confusion about identifying style factors; differences in method of implementation; author bias and subjectivity; improper methodology. All of which are evident in the literature reviewed in this chapter. In the next chapter we define a taxonomy of programming style factors which can be used to alleviate some of these problems.

Chapter 3

A Programming Style Taxonomy

3.1. Building a style taxonomy.

We developed our style taxonomy through a systematic analysis of programming style guidebooks and code analyzers. We started by compiling a list of every style alteration in the examples from Kernighan and Plauger's book *The Elements of Programming Style* [Kern74], the most cited and widely used book on programming style. They illustrate each of their style lessons by discussing the shortcomings of an example (all examples are taken from programming textbooks), rewriting the example in a better way, and drawing the general rule from the specific case.

In every rewritten code segment there were explicit style alterations (those under discussion) and corresponding implicit changes (unstated alterations) that enhance the differences between good and bad versions of code. For example, over half of the rewritten versions contain unmentioned changes in indentation and/or embedded spacing. Thus, our list of style factors was considerably larger than the list of Kernighan and Plauger's rules.

To this list we added style factors gleaned from other sources [Arth85, Berr85, Coop85, Ledg78, Meek83, Redi86, Rees82, Rose83] and then classified the style factors by combining common principles and factors into style categories. This technique for organizing information has been used to develop other taxonomies [Oman86].

The resulting operational taxonomy is broken down into four major categories: general practices, typographic style, control structure style, and information structure style. As shown in Figure 3.1, the later three categories are further broken down into macro (whole program or intermodule characteristics) and micro (statement or intramodule characteristics) subcategories. Definitions, classification criteria (governing the placement of factors within each category), example subclasses, application principles (governing the use of style factors within the category), and example style rules are shown in Figure 3.2.

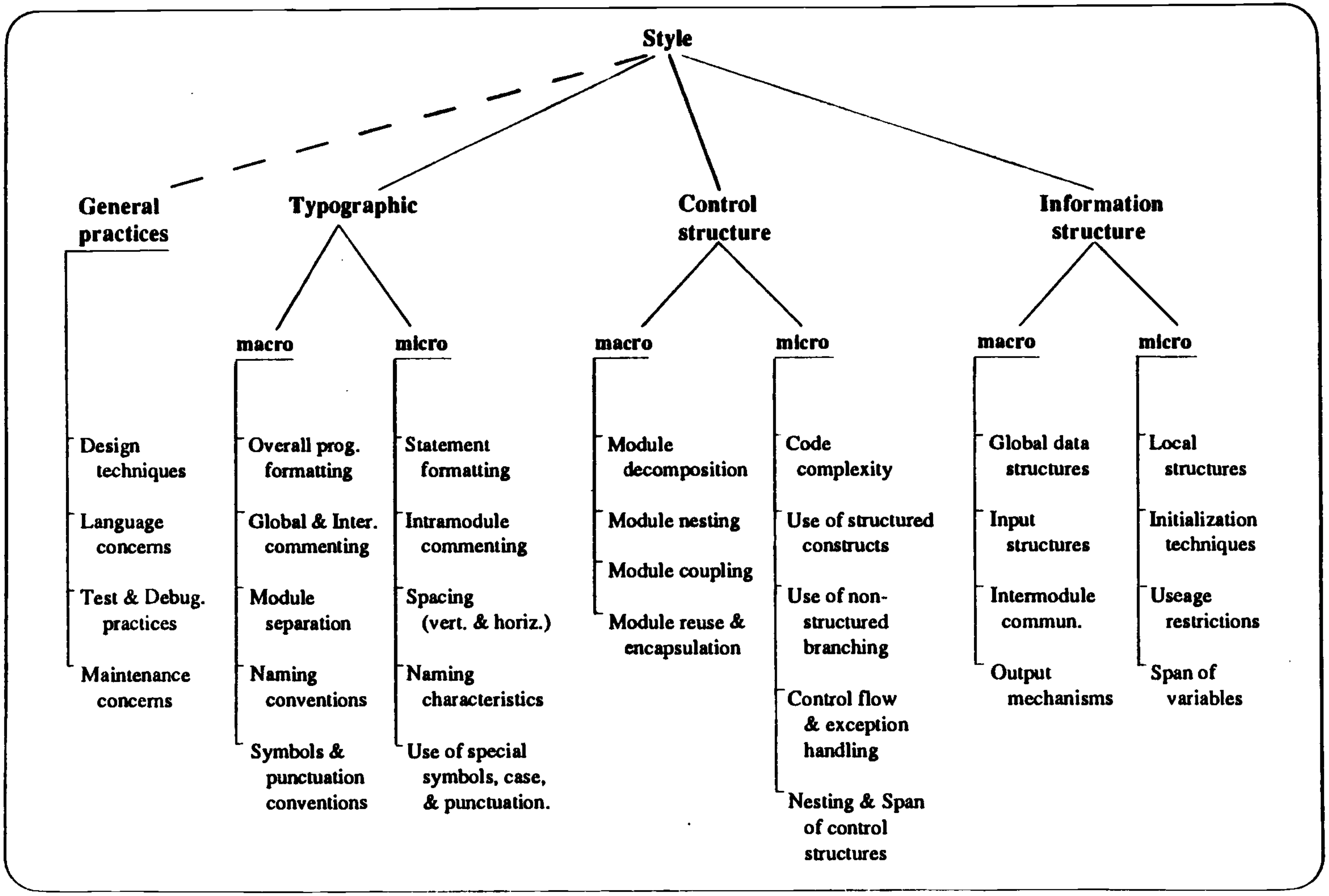


Figure 3.1. A Style Taxonomy

0. General programming practices:

A. Definition: Rules and guidelines pertaining to the programming process that directly affect the style of the product.

B. Classification criteria: Programming methodology constraints, usually temporal in nature.

C. Example subclasses: (a) Design techniques, (b) Language selection and restrictions, (c) Testing and debugging practices, (d) Maintenance concerns.

D. Application guide: < Cannot be specified here because practices are dependent upon local methods and constraints. >

E. Example rules: (a) Define the problem completely; resist the urge to start coding, (b) Use only ANSI standard features, (c) Build in debugging techniques, (d) Don't comment bad code -- rewrite it.

1. Typographic style:

A. Definition: Style characteristics affecting only the typographic layout and commenting of code with no affect on program execution. It is divided into macro- and micro-typographic categories.

1.1. Macro-typographic style:

B. Classification criteria: Style characteristics affecting the overall layout and commenting of a program or system, including systemwide typographic and commenting conventions and intermodular layout and naming characteristics.

C. Example subclasses: (a) Overall program formatting, (b) Global and intermodule commenting, (c) Module separation conventions, (d) Identifier and label naming conventions, (e) Conventions for special symbols, case, fonts and type styles.

D. Application guide: (a) Highlight and document system decomposition and intermodule communication, (b) augment the clarity and readability of the host language through naming and typographic conventions.

E. Example rules: (a) Use verbs to name procedures, (b) Separate modules with at least 3 lines, (c) Use boldface to highlight reserved words.

1.2. Micro-typographic style:

B. Classification criteria: Style characteristics affecting the intramodular typographic layout and commenting of a module within a program or system, including local naming characteristics.

C. Example subclasses: (a) Statement formatting, (b) Vertical spacing, (c) Horizontal spacing (indentation & alignment), (d) Intramodule commenting (content & form), (e) Identifier and label naming, (f) Use of special symbols and case.

D. Application guide: (a) Highlight and document control flow and information flow within a module, (b) provide spatial clues about program constructs such as loops, branches, decisions, and nested statments, (c) highlight related groups of statements, and (d) improve the clarity and readability of the algorithm through naming and the use of special symbols and character case.

E. Example rules: (a) Place only 1 control statement per line, (b) Separate program constructs by at least 1 blank line, (c) Put a space after every comma.

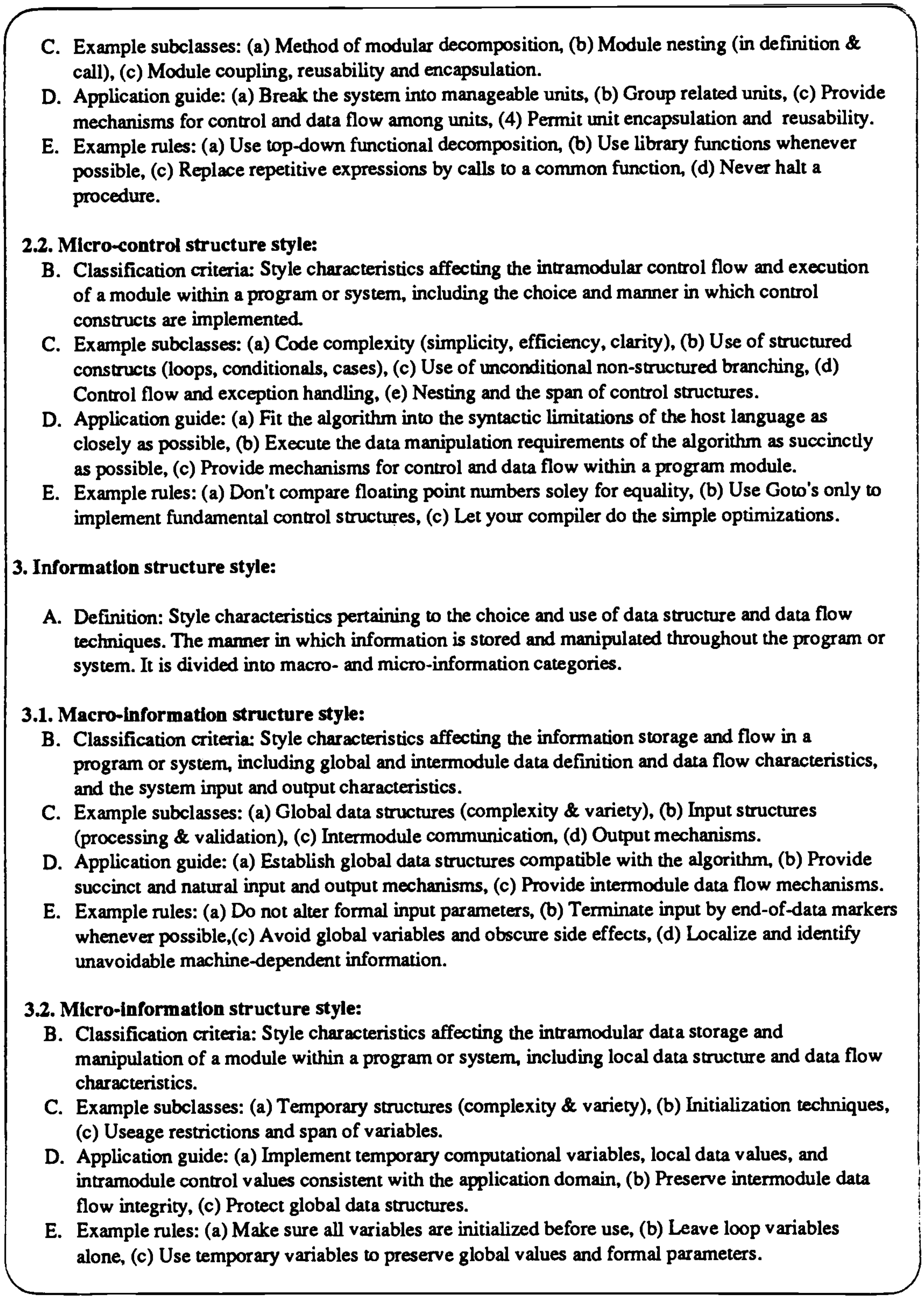
2. Control structure style:

A. Definition: Style characteristics pertaining to the choice and use of control flow constructs, the manner in which the program or system is decomposed into algorithms, and the method in which those algorithms are implemented. This excludes data structure aspects and is divided into macro- and micro-structural categories.

2.1. Macro-control structure style:

B. Classification criteria: Style characteristics affecting the modular structure and execution of a program or system, including intermodule control characteristics.

Figure 3.2. Style Taxonomy Definition



C. Example subclasses: (a) Method of modular decomposition, (b) Module nesting (in definition & call), (c) Module coupling, reusability and encapsulation.

D. Application guide: (a) Break the system into manageable units, (b) Group related units, (c) Provide mechanisms for control and data flow among units, (4) Permit unit encapsulation and reusability.

E. Example rules: (a) Use top-down functional decomposition, (b) Use library functions whenever possible, (c) Replace repetitive expressions by calls to a common function, (d) Never halt a procedure.

2.2. Micro-control structure style:

B. Classification criteria: Style characteristics affecting the intramodular control flow and execution of a module within a program or system, including the choice and manner in which control constructs are implemented.

C. Example subclasses: (a) Code complexity (simplicity, efficiency, clarity), (b) Use of structured constructs (loops, conditionals, cases), (c) Use of unconditional non-structured branching, (d) Control flow and exception handling, (e) Nesting and the span of control structures.

D. Application guide: (a) Fit the algorithm into the syntactic limitations of the host language as closely as possible, (b) Execute the data manipulation requirements of the algorithm as succinctly as possible, (c) Provide mechanisms for control and data flow within a program module.

E. Example rules: (a) Don't compare floating point numbers soley for equality, (b) Use Goto's only to implement fundamental control structures, (c) Let your compiler do the simple optimizations.

3. Information structure style:

A. Definition: Style characteristics pertaining to the choice and use of data structure and data flow techniques. The manner in which information is stored and manipulated throughout the program or system. It is divided into macro- and micro-information categories.

3.1. Macro-information structure style:

B. Classification criteria: Style characteristics affecting the information storage and flow in a program or system, including global and intermodule data definition and data flow characteristics, and the system input and output characteristics.

C. Example subclasses: (a) Global data structures (complexity & variety), (b) Input structures (processing & validation), (c) Intermodule communication, (d) Output mechanisms.

D. Application guide: (a) Establish global data structures compatible with the algorithm, (b) Provide succinct and natural input and output mechanisms, (c) Provide intermodule data flow mechanisms.

E. Example rules: (a) Do not alter formal input parameters, (b) Terminate input by end-of-data markers whenever possible,(c) Avoid global variables and obscure side effects, (d) Localize and identify unavoidable machine-dependent information.

3.2. Micro-information structure style:

B. Classification criteria: Style characteristics affecting the intramodular data storage and manipulation of a module within a program or system, including local data structure and data flow characteristics.

C. Example subclasses: (a) Temporary structures (complexity & variety), (b) Initialization techniques, (c) Useage restrictions and span of variables.

D. Application guide: (a) Implement temporary computational variables, local data values, and intramodule control values consistent with the application domain, (b) Preserve intermodule data flow integrity, (c) Protect global data structures.

E. Example rules: (a) Make sure all variables are initialized before use, (b) Leave loop variables alone, (c) Use temporary variables to preserve global values and formal parameters.

Figure 3.2. Style Taxonomy Definition (continued)

The first major area of the taxonomy, "General programming practices," is not a grouping of style factors, per se. Rather, it is a recognition that attributes of the programming process affect the style characteristics manifested in the code. The remaining three areas are strictly groupings of style characteristics of the source code. Macro concerns are those pertaining to the whole program (or system) and the interaction between the components of the system. Micro concerns are those contained within a single program unit or module. This micro-macro division not only permits the convenient separation of existing rules, but also provides a mechanism to identify style characteristics affecting programming "in-the-large."

3.2. Verifying the taxonomy.

To verify the appropriateness and robustness or our taxonomy, we extracted all the style rules and guidelines from [Fair85, Kern74, Ledg75, Ledg78, Ledg87, Pres82, Schn81] and placed them into our taxonomic structure. The result is an organized composite listing of 236 style rules. It is the largest list of programming style rules compiled to date. The entire composite listing is presented in Appendix A.

As in all taxonomies, there are elements that seem to overlap the defined categories, there is room to include new or unrecognized elements, and there may be disagreements about the placement of specific elements. These problems are not uncommon in biological or physical science taxonomies. We stress that this is an operational style taxonomy that we have found very useful in our style research. Further, it is the most complete organization of programming style factors assembled to date.

However, we do not claim that this classification incorporates all factors comprising programming style. It is a taxonomy that provides organizational structure for the controlled study of programming style attributes. It is a classification system that greatly aided our understanding of style, style factors, and their influences. We recognize that other taxonomies may be devised.

3.3. A taxonomic look at programming style rules.

We then examined the composite listing of rules and the individual style guides with the aim of identifying areas where further studies on style factors are needed. We identified many rules that are not substantiated by empirical evidence, and quite a few that are contradictory both within and between authors.

In the next subsections we present a discussion of these areas. Note that rules and guidelines listed in this paper have been taken out of context, and hence, some of the problems may not be readily apparent. However, a taxonomic inspection of any collection of style rules will reveal instances of inconsistency, implicit style changes in example materials, contradictions, and subjective recommendations not substantiated by empirical evidence.

3.3.1. Widely held but unsubstantiated rules.

To determine just how much authors agreed on various style guidelines we combined all the functionally equivalent rules in the composite rule listing. Figure 3.3 contains a listing of those rules occurring at least three times in the composite listing. The rules were either taken directly from the composite listing or were rewritten for clarity. The number in parenthesis following each rule is the number of times that rule occurs in the composite listing. For example, the rule about rewriting bad code can be found four times in the composite listing.

As can be seen, there are many rules that are widely accepted as principles of good programming style. However, many of these widely held beliefs have not been substantiated and are subject to criticism. Following is a short list of example rules and corresponding questions that needs to be addressed before the validity of the rule can be established.

1. Rule: First understand and define the problem; don't start coding right away.
   Problem: Does rapid prototyping help in problem definition?

2. Rules: Make sure comments and code agree.
   Avoid obvious comments, make every comment count.
   Problem: What role do comments play in programmer comprehension and maintenance activity? Are they looked at, or are they a nuisance?

3. Rule: Prologue comments should appear at the beginning of every module.
   Problem: What role do prologue comments serve? What information belongs in prologue comments?

4. Rule: Do not put more than one control statement per line.
   Problem: How do programmers best read code -- vertically (one statement per line) or horizontally (multiple statements per line).

5. Rules: Use blank lines to improve readability.
   Use spacing to improve readability.
   Problem: What is the best way to "paragraph" code -- chunking or alignment?

6. Rules: Use variable names that mean something.
   Do not use cryptic or confusing variable names.
   Try to make the code read well by choosing appropriate names for identifiers.
   Problem: Should mnemonic names be selected from the application domain or from the programmer's perspective?

7. Rules: Modularize your programs.
   Avoid unnecessary GOTOs.
   Use GOTOs in a disciplined way.
   Avoid heavy nesting of loops and conditionals.
   Problem: Which is easier to understand, unconditional branching or intermodule control parameters?

8. Rules: Avoid global variables.
   Avoid side effects.
   Problem: Which is easier to understand, global variables and side effects or intermodule parameter passing? Can real-world events be simulated without globals and side effects?

As can be seen from Figure 3.3 and the above list, there are many unanswered questions about even the widely held beliefs on good programming style.

0. General programming practices.

- First understand and define the problem; don't start coding right away. (4)
- Build in debugging and data testing techniques. (3)
- Check data and results (against known values) for correctness. (3)
- Don't just patch or comment bad code -- rewrite it. (4)
- Make sure comments and code agree. (3)

1. Typographic style.

- Prologue comments should appear at the beginning of every module. (5)
- Do not put more than one control statement per line. (3)
- Use parentheses to improve clarity and readability. (4)
- Use blank lines to improve readability. (3)
- Use spacing to improve readability. (3)
- Avoid obvious comments, make every comment count. (3)
- Use identifier names that mean something. (5)
- Do not use cryptic or confusing variable names. (6)
- Choose identifier names that make the code read well. (4)

2. Control structure style.

- Modularize your programs. (6)
- Use library functions. (3)
- Don't sacrifice clarity for efficiency. (6)
- Always strive for simplicity and clarity. (8)
- Let the compiler do the simple optimizing. (3)
- Avoid confusing programming tricks. (6)
- Avoid unecessary GOTOs. (6)
- Use GOTOs in a disciplined way. (4)
- Avoid deep nesting of loops and conditionals. (3)

3. Information structure style.

- Localize and identify machine-dependencies. (3)
- Avoid machine dependencies. (3)
- Keep input format(s) uniform and simple. (5)
- Terminate input by end-of-data markers (not by counting). (3)
- Identify the input you are requesting, make data entry easy for the user. (9)
- Validate input for legality and plausibility. (5)
- Allow for default input values. (3)
- Identify bad input and recover if possible. (5)
- Avoid global variables. (3)
- Avoid side effects. (5)
- Initialize variables. (3)

Figure 3.3. Widely Held Style Guidelines

3.3.2. Incompatible style rules.

To determine just how much authors disagree on various style issues, we pulled together all the conflicting style rules. Figure 3.4 contains sets of style rules -- grouped to show conflicts -- taken from the composite rule listing. Again, rules were selected directly from the composite listing or were rewritten for clarity; the number of times the rule occurs in the composite listing is in parenthesis. Figure 3.4 shows many conflicts that need resolution before style standards can be formulated. Following is a short list of questions that need addressing.

1. What should be the form and content of prologue and inline comments? How should comments appear and what type of information should they contain?

2. How should code chunks be indicated (indented, aligned and paragraphed)?

3. Is there a maximum identifier name length, beyond which it interferes with readability?

4. How should character case and boldface fonts be used to distinguish between syntactic and semantic constructs?

5. What type of modularity should be used (i.e., functional decomposition, objected oriented, rule based, or abstract data type)?

6. How should language extensions (i.e., non-standard features) be used?

7. To what degree should programmers optimize computational expressions?

8. What is the acceptable use of GOTOs relative to modularity and information flow?

9. What is the acceptable depth of control statement nesting?

10. What are the trade-offs between formal parameters lists, temporary variables, global variables and side effects?

0. General programming practices. < no apparent contflicts >

1. Typographic style.

- o Ident to highlight nesting depth. (3)
- o Indent so structurally related clauses are aligned. (2)
- o Indent so structures are in comb-like form. (2)
- o Comment any statement whose intent is not immediately obvious. (1)
- o When in doubt, leave the comment out. (1)
- o Avoid inline comments. (3)
- o Use inline comments. (5)
- o Comment blocks of code (rather than lines). (2)
- o Comment lines of code (rather than blocks). (3)
- o Comment for content, not for dazzle (don't highlight). (2)
- o Use borders (or other means) to highlight comments. (3)
- o Do not abbreviate or limit the length of names. (2)
- o Strive for brevity in names. (2)

2. Code structure style.

- o Use (non-standard) library functions. (2)
- o Avoid implementation-dependent features. (3)
- o Let the system do the expression optimization. (3)
- o Use "fast" arithmetic operations. (1)
- o Avoid the use of complicated conditional tests. (1)
- o Use nested IF-THEN-ELSEs to implement multi-way conditionals. (2)
- o Avoid IF-THEN-IF statements (by collapsing conditionals). (1)
- o Use GOTOs only to implement fundamental control structures. (1)
- o Use the GOTO to handle errors, unusual circumstances, or other abnormalities in logic. (2)
- o Do not use the GOTO to jump backwards or exit a loop. (2)

3. Information structure style.

- o Choose a data representation that makes the program simple. (2)
- o Avoid the use of multidimensional arrays. (1)
- o Use data arrays to avoid repetitive control sequences. (1)
- o Avoid the use of pointers and complex lists. (1)
- o Use recursive procedures for recursive data structures. (1)
- o Keep input format(s) simple. (3)
- o Select formats for user-ease, not ease of programming. (2)
- o Keep input format(s) uniform. (2)
- o Do not alter formal input parameters. (2)
- o Make key data values parameters to procedures. (1)
- o Avoid global variables and side effects. (5)
- o Use global variables only to implement "own" variables. (1)
- o Always echo print the input. (1)
- o Avoid producing output within a procedure (unless the sole purpose of the procedure is output). (1)
- o Do not be afraid to use temporary variables. (2)
- o Avoid temporary variables. (1)

Figure 3.4. Incompatible Style Rules

3.3.3. Tacit misconceptions on programming style.

Our taxonomic analysis of programming style brought up many questions about how programmers view and use style factors in their code. In subsequent discussions with professional programmers, we noticed the widespread existence of tacit misconceptions about programming style. For example, there is a common notion that pretty-printers improve the comprehendability of code and it is frequently stated that professional programmers follow widely accepted style "standards," although none exist.

From our conversations we were able to identify several common fallacies regarding style. Figure 3.5 contains a list of tacit misconceptions that we identified. This list suggests that many programmers are unaware of the unsubstantiated and conflicting rules that we identified in the previous two subsections. It is clear that examining programming style in light of the style taxonomy vividly illustrates these problems and raises important questions. That was the major motivation for the taxonomy.

3.4. Conclusions.

We suggest that previous work in programming style resulted in the many contradictions reported in the literature, primarily because it lacked the appropriate paradigm for style research. This taxonomy completes the first stage of our research model presented in Chapter 1; namely, the identification and categorization of programming style factors.

Stage 2 of our research paradigm calls for a detailed analysis of style factors to determine their affects on program complexity and programmer comprehension. In the next chapter we present several experiments analyzing style in light of our taxonomy. The significant results from these studies further demonstrate the value of a taxonomic analysis of style.

- There are programming style "standards" that are widely accepted and adherred to by professional programmers.
- Prettyprinters improve the comprehendability of the code.
- Software complexity metrics measure the overall quality of programming style.
- There is one best style for all applications and/or languages (e.g. There is one best method of indenting control constructs).
- Other programmers (i.e. readers) will implicitly understand your method of module decomposition, your organization of module definition (i.e. structure graph), and your sequence of module execution (i.e. call graph).
- Other programmers will intuitively "see" the information flow and implicitly understand your I/O prompts and formats.
- Your choice of meaningful identifier names will provide the same connotations (meaning) to other programmers.
- Comments should be highlighted by boxes, stars, or other eye-catching structures.
- If a section of code is hard to understand, then it should be commented.
- Comments are for use by new or novice programmers unfamiliar with the application domain.
- All reserved words should be highlighted by either character case or boldface fonts.
- Continued nesting of indentation should be used to exhibit the nested structure of control statements.

Figure 3.5. Tacit Misconceptions About Programming Style.

Chapter 4

Analyzing Typographic Style

4.1. The benefits of taxonomic analysis.

It is generally accepted that control structure style and information structure style affect both program comprehension and maintenance. Numerous studies have investigated the affects of control structure on complexity, maintainability, and error-proneness [Bake79, Basi84, Card85, Davi83, Davi84, Romb87, Vess83]. Similar studies looking at information structure can be found [Bast87, Berg85, Bern84, Doer85, Hutc85]. But aside from the few studies looking at indentation and naming, there are few studies that have been restricted to isolating and studying typographic factors.

In this chapter we present three studies that demonstrate the benefits of analyzing typographic style using our taxonomic breakdown of style factors. Specifically, we show that: (1) typographic measures are useful in style analysis, (2) analyzing experimental results in light of the taxonomy reveals many insights about typographic formatting, (3) typographic style characteristics have an important and influential impact on programmer comprehension and, (4) traditional methods of typographic formatting are inadequate because they obscure structural clues about the underlying code.

4.2. Indices of style.

It is frequently stated that software complexity metrics are indices of good programming style [Auth85, Mart83]. For this reason (and because it is convenient) all existing style analyzers use various complexity measures as indices of good style. But, software complexity metrics are not measures of style per se; rather, they are gross measures of size, control flow, information flow, etc. Hence, the role of software complexity metrics in measuring programming style is questionable.

We wanted to determine what role software complexity metrics could play in the analysis of style; and what typographic measures could be defined and used in style analysis. In the next two subsections we describe a series of experiments showing that software complexity metrics are inadequate indices of style and that typographic metrics can be defined and are useful indices of style.

4.2.1 An early attempt to "capture" style.

Our conjecture was that programmers leave stylistic "footprints" in their code that can be quantified and used for traceability purposes. Hence, measuring these style characteristics would enable us to "capture" the style of the program and allow us to group programs with similar characteristics. In our first experiment we tested software complexity metrics as measures of programmer style. Twelve complexity metrics (delivered source lines, lines of code, lines of declaration, lines of comments, number of tokens, number of arguments, cyclomatic complexity, Halstead's operand and operator counts, and level of nesting) were calculated for the same three Pascal code segments (bubble sort, quicksort, and tree traversal) taken from six different data structure textbooks.

The six different code styles used in the textbooks were sufficiently distinct to allow programmers to easily group the code segments by authorship. In fact, the six textbooks were selected because of their distinct, but not uncommon, style of programming. Furthermore, the three algorithms were equally distinct and could easily be categorized by any programmer of moderate experience. For each code segment the vector of complexity measurements represented an attempt to quantify the style of the code.

Although the data contained multiple instances of the same algorithm written in different styles, and different algorithms written in the same style, repeated statistical analysis for appearance of clustering trends and principal components failed to find any relationship between the complexity measures of the 18 code segments. That is, authorship and algorithm domain characteristics were not represented in the complexity vectors. In fact, the lack of any discernible relationship was quite startling; measurements of code complexity seem independent of either authorship or application domain. (We show in subsequent experiments that this can be done, with exactly the same data, when using typographic style measures).

We concluded that software complexity metrics are poor measures of programming style. This conclusion contradicts claims made by [Arth85, Mart83], but is supported by two previous studies. A plagiarism detection study by Berghel and Sallach [Berg84] concluded that software metrics were of limited use in that application. Their study investigated 15 common complexity metrics. After conducting a factor analysis of the metrics representing student programs, they concluded that there was nothing unique about the program features isolated by the metrics. The other study supporting our conclusion is Evangelist's [Evan84] analysis of complexity metrics with respect to style rules. He showed that application of 26 rules from Kernighan and Plauger's style guide had unpredictable affects on five widely used complexity metrics: Halstead's effort, McCabe's cyclomatic complexity, Henry and Kafura's information flow, level of nesting, and number of lines of code. Some rules increased complexity (as measured by the metrics) while others decreased complexity and others had inconsistent affects on the different metrics. He concludes, "current complexity metrics are improper indices of program quality, as measured by style."

4.2.2. Typographic style analysis.

We then conducted some experiments to determine how a programmer's style could be recognized from analysis of source code. Talks with programmers uncovered the belief that programmers can identify authorship from typographic characteristics. To test this hypothesis we took the same code segments used in the software complexity experiment and copied them verbatim onto a microcomputer and printed them one per page to eliminate all publisher differences (i.e. typesetting). The 18 pages were then shuffled and given to eleven programmers with instructions to group the code by author. Each author's collection would contain one bubble sort, one quicksort, and one set of tree traversals. All but one of the subjects grouped the code perfectly, the other subject made one mistake by switching the tree traversals on the two most inconsistent authors.

An informal protocol analysis was conducted while the subjects were grouping the code listings. The protocol analysis and subsequent post-test discussions identified a number of typographic style characteristics that could easily be measured. We then designed and implemented a style analyzer based on 16 metrics for comments, indentation, character case, extended identifier names, statement formatting, and the use of blank lines.

For each of the 16 measures a boolean value is true if the characteristic is present in the code under analysis, and false if the characteristic is absent. The analysis proceeds on a module-by-module basis with the output being a boolean matrix with each row being a typographic bit vector for a module (program, procedure, or function) and each column representing one of the typographic conditions. To obtain a typographic bit vector for algorithms with embedded modules, the bit vector for each embedded module was OR-ed together with the main module. This is functionally equivalent to processing the entire algorithm as one block (except that multiple indentation methods may appear).

Table 4.1 shows the boolean typographic style measures for the data used in the protocol study. A simple index of typographic style consistency can be derived by counting the number of typographic style factors showing any instance of inconsistency (i.e., differing in a bit position). Inconsistency Ratings (IR) for each textbook are also shown in Table 4.1.

Table 4.1.

Typographic Style Vectors for Textbook Data

	INL	BLK	BOR	KEY	I2	I4	I5	LCO	UCO	<C>	U_S	BGN	THN	;;;	BLD	BLB
Text A (IR = 0)																
Bubblesort	1	1	0	1	1	0	0	0	1	0	0	0	0	0	1	1
Quicksort	1	1	0	1	1	0	0	0	1	0	0	0	0	0	1	1
Tree Traversal	1	1	0	1	1	0	0	0	1	0	0	0	0	0	1	1
Text B (IR = 1)																
Bubblesort	1	1	1	1	1	0	0	0	0	1	0	0	1	0	0	0
Quicksort	1	1	1	1	1	0	0	0	0	1	0	0	1	0	0	1
Tree Traversal	1	1	1	1	1	0	0	0	0	1	0	0	1	0	0	0
Text C (IR = 2)																
Bubblesort	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
Quicksort	0	0	0	0	0	1	0	0	0	1	0	0	0	0	1	1
Tree Traversal	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
Text D (IR = 3)																
Bubblesort	1	1	0	1	0	0	1	1	0	0	0	0	1	0	0	0
Quicksort	1	0	0	1	0	0	1	1	0	0	0	0	1	0	1	1
Tree Traversal	1	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0
Text E (IR = 6)																
Bubblesort	1	0	0	0	1	0	0	1	0	0	0	0	1	1	1	1
Quicksort	1	1	0	1	1	0	0	1	0	0	0	0	1	1	1	1
Tree Traversal	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0
Text F (IR = 6)																
Bubblesort	1	0	0	1	0	0	1	1	0	0	0	1	0	1	0	0
Quicksort	1	0	0	1	0	1	0	1	0	0	0	1	0	1	0	0
Tree Traversal	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0

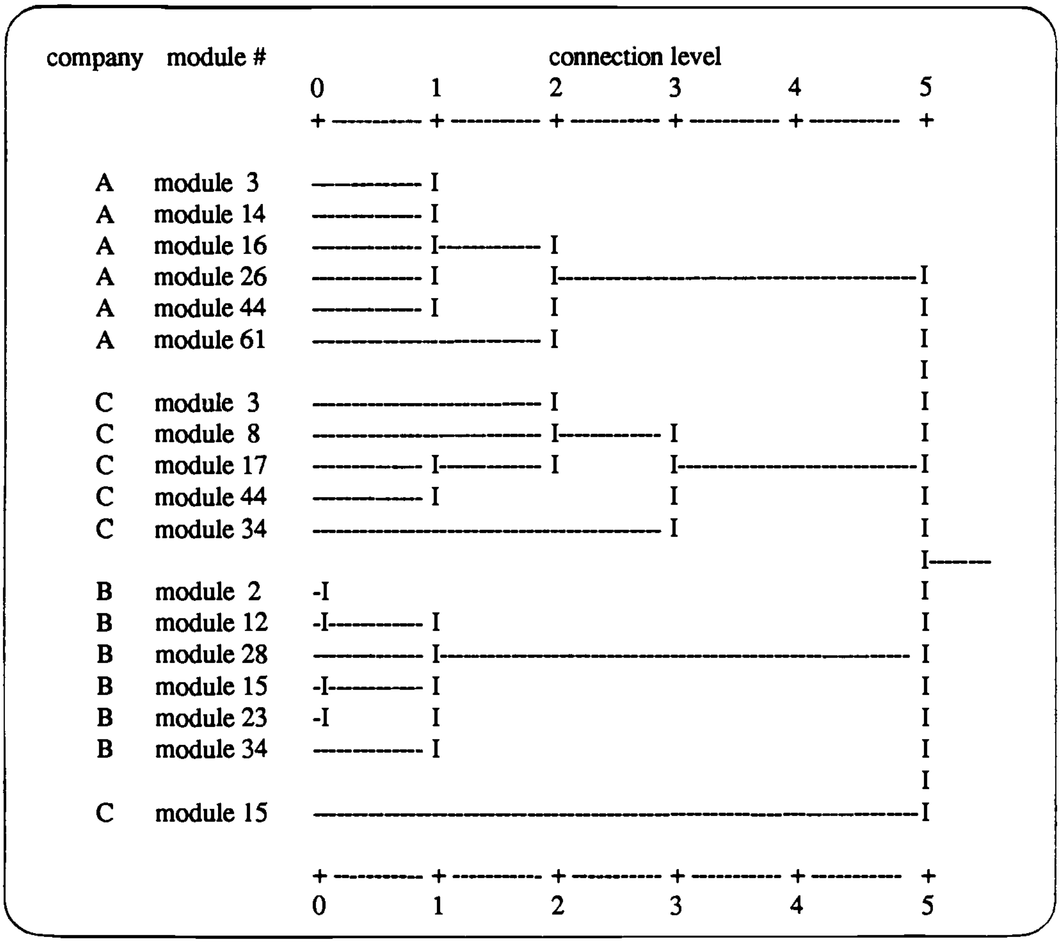
INL	- inline comments	I5	- 5 & greater indentation	THN	- Then & statement on 1 line
BLK	- block of comments	LCO	- lower case only	;;;	- many statements per line
BOR	- bordered comments	UCO	- upper case only	BLD	- blank lines in declarations
KEY	- comments after keywords	<C>	- case distinguishes keyword	BLB	- blank lines in the body
I2	- 1 & 2 space indentation	U_S	- underscore used		
I4	- 3 & 4 space indentation	BGN	- Begin & statement on 1 line	IR	- Inconsistency Rating

We then conducted a clustering analysis to determine if distinct typographic styles could be grouped by programmer. Using the textbook data, the proximity of each algorithm's style measurements to all other algorithms was computed by taking the Hamming distance between the typographic style bit vectors for each algorithm. The resulting N x N distance matrix was then analyzed using the SPSS-X Cluster procedure, with a minimum distance clustering criteria [Noru85], to produce the cluster graph in Figure 4.1. The cluster graph shows "degree of similarity," with identical items connected at level 0 and decreasing similarity with increasing connection levels. The statistical clustering of the textbook data follows that of our protocol study; algorithms written by consistent authors are tightly clustered, while others are somewhat scattered.

```
text  algorithm                             connection level
                        0         1         2         3         4         5
                        +---------+---------+---------+---------+---------+

C     bubblesort       -I-------------------I
C     tree traversal   -I                   I-------------------I
C     quicksort        ---------------------I                   I
                                                                I
F     bubblesort       ---------------------I---------I         I
F     quicksort        ---------------------I         I---------I
                                                      I         I
D     bubblesort       -----------I---------I         I         I
D     tree traversal   -----------I         I---------I         I
D     quicksort        ---------------------I                   I
                                                                I---------I
F     tree traversal   -----------------------------------------I         I
                                                                I         I
E     bubblesort       ---------------------I-------------------I         I
E     quicksort        ---------------------I                   I         I
                                                                I         I
E     tree traversal   -----------------------------------------I         I
                                                                I         I------
A     bubblesort       -I                                       I         I
A     quicksort        -I---------------------------------------I         I
A     tree traversal   -I                                                 I
                                                                          I
B     bubblesort       -I---------I                                       I
B     tree traversal   -I         I---------------------------------------I
B     quicksort        -----------I

                        +---------+---------+---------+---------+---------+
                        0         1         2         3         4         5
```

Figure 4.1. Cluster Analysis of Textbook Authors

We then ran several clustering analyses on industrial programs. Pascal source code was obtained from two international computer manufacturers and one microelectronics research laboratory. The length of the three programs was 6024, 1445, and 2711 lines of source code. As an example test, six modules were selected at random from each of the three programs, run through the style checker, reduced to a distance matrix, and then clustered using the minimum distance criteria. The results are shown in Figure 4.2.



```
  company   module #                    connection level
                              0         1         2         3         4         5
                              +---------+---------+---------+---------+---------+

    A    module 3             ----------I
    A    module 14            ----------I
    A    module 16            ----------I---------I
    A    module 26            ----------I         I-----------------------------I
    A    module 44            ----------I         I                             I
    A    module 61            --------------------I                             I
                                                                                I
    C    module 3             --------------------I                             I
    C    module 8             --------------------I---------I                   I
    C    module 17            ----------I---------I         I-------------------I
    C    module 44            ----------I                   I                   I
    C    module 34            ------------------------------I                   I
                                                                                I-----
    B    module 2             -I                                                I
    B    module 12            -I--------I                                       I
    B    module 28            ----------I---------------------------------------I
    B    module 15            -I--------I                                       I
    B    module 23            -I        I                                       I
    B    module 34            ----------I                                       I
                                                                                I
    C    module 15            --------------------------------------------------I

                              +---------+---------+---------+---------+---------+
                              0         1         2         3         4         5
```

Figure 4.2. Cluster Analysis of Industrial Code

As shown in Figures 4.2, our methodology identifies and clusters the modules from Company A and Company B perfectly. The Company C code is least consistent; joining at higher connection levels and containing anomalies like Module #15.

The cluster graphs in Figures 4.1 and 4.2 demonstrate the utility in analyzing typographic style factors. We have applied this system to other industrial code, individual student projects, and code from teams of students working in software engineering practicums. Our results show that typographic style measures are useful in: (1) measuring the internal consistency of code styles, (2) identifying style anomalies among modules, (3) comparing styles across companies, and (4) clustering code segments by authorship.

4.3. The affects of style on programmer comprehension.

Convinced that typographic metrics are useful indices of programming style, we then ran some experiments to determine how specific styles affected programmer comprehension. The following two experiments demonstrate that typographic style characteristics do affect program comprehension and that some forms of typographic style are better suited to portray the underlying code structure.

4.3.1. First nested IF experiment.

To demonstrate that typographic style affects program comprehension, we conducted an experiment comparing three versions of nested IF statements taken from Kernighan & Plauger's *Elements of Programming Style*, [Kern74,p.123]. The three versions are listed in Figure 4.3. Kernighan & Plauger present version 1 as an example of an "ill-chosen layout," and suggest that simple reformatting can improve understandability. Version 2 is their typographic rearrangement of version 1; the only changes are method of indentation and use of embedded spaces. They then argue that a further transformation yields a better version analogous to a CASE statement. By combining two logical conditions they eliminate one nested IF-THEN-ELSE and derive version 3.

We emphasize that the difference between versions 1 and 2 is entirely typographic arrangement: indentation, embedded spacing, and alignment. The structural combination of two logical conditions in versions 3 causes differences in indentation between version 2 and 3. Hence, the latter two versions differ only slightly in both structure and typographic style.

```
Version 1:  IF A > B
              THEN S := 1
              ELSE IF A = B
                THEN IF C > D
                  THEN S := 2
                  ELSE S := 3
                ELSE IF C > D
                  THEN S := 4
                  ELSE IF C = D
                    THEN S := 5
                    ELSE S := 6;

Version 2:  IF          A>B   THEN   S := 1
            ELSE  IF    A=B   THEN
                  IF    C>D   THEN   S := 2
                              ELSE   S := 3
            ELSE  IF    C>D   THEN   S := 4
            ELSE  IF    C=D   THEN   S := 5
            ELSE                     S := 6;

Version 3:  IF          A>B                   THEN   S := 1
            ELSE  IF    (A=B) AND (C>D)       THEN   S := 2
            ELSE  IF     A=B                  THEN   S := 3
            ELSE  IF                C>D       THEN   S := 4
            ELSE  IF                C=D       THEN   S := 5
            ELSE                                     S := 6;
```

Figure 4.3. IF Statements for First Nested IF Experiment

A simple comprehension test, consisting of five short answer and two subjective questions, was designed and administered to 36 intermediate computer science students. Appendix B contains a sample experimental packet. The experiment was conducted during the first ten minutes of class. Subjects were randomly assigned into three treatment groups of 12 students; each group receiving one version of the nested IF statements. They were asked to read the IF statements, answer the five questions that followed, record the time necessary to do so, and then subjectively rate the indentation and structure of the code. Thus, the independent variable was the "style" of the nested IF statements (Kernighan and Plauger's three versions). Dependent measures for each subject were: score (0 to 5 points), time required to answer the five questions (0 to 5 minutes), and subjective ratings for indentation and structure on a 5 point forced-choice scale (1- very poor, 5-very good). The materials distributed to subjects consisted of a page of instructions followed by a listing of the IF statements and the test questions.

Average scores, times, and ratings for all three groups are shown in Table 4.2. In support of Kernighan & Plauger, both scores and times improve across the three versions. This is illustrated by the plot of scores and times in Figure 4.4. Interestingly, the subjective ratings show virtually no change across the styles. Univariate and multivariate analysis of variance showed a significant difference for the time measure (F=4.99, p<.01, d.f.=2,33) [Dixo81]. Although the trend is for scores to improve across the versions, these differences are not strong enough to report significance. This is not unreasonable in light of the small number of subjects and the brevity of the task.

The results of this experiment provide strong evidence that stylistic variations affect program comprehension and that typographic style characteristics are more than cosmetic. We speculated that the improvement in comprehension observed across the three versions was due to improved spatial arrangement provided by the typographic reformatting. In each successive version there is a refinement toward an arrangement that displays the 1:1 relationship between logical conditions and subsequent action. This 1:1 relationship first takes shape in Kernighan and Plauger's second version and becomes more apparent in their third version. This refinement in clarity corresponds exactly with our observed improvements in comprehension. To test this assertion we conducted another experiment with materials having a wider range of spatial differences.

Table 4.2.

Results from First Nested IF Experiment

	Version		
Averages:	K&P #1	K&P #2	K&P #3
test score	3.41	3.66	4.33
test time	3.85	3.24	2.69 *
indentation rating	2.25	2.75	2.81
structure rating	2.75	2.91	2.91

* significant: (F=4.99, p<.01, d.f.=2,33)

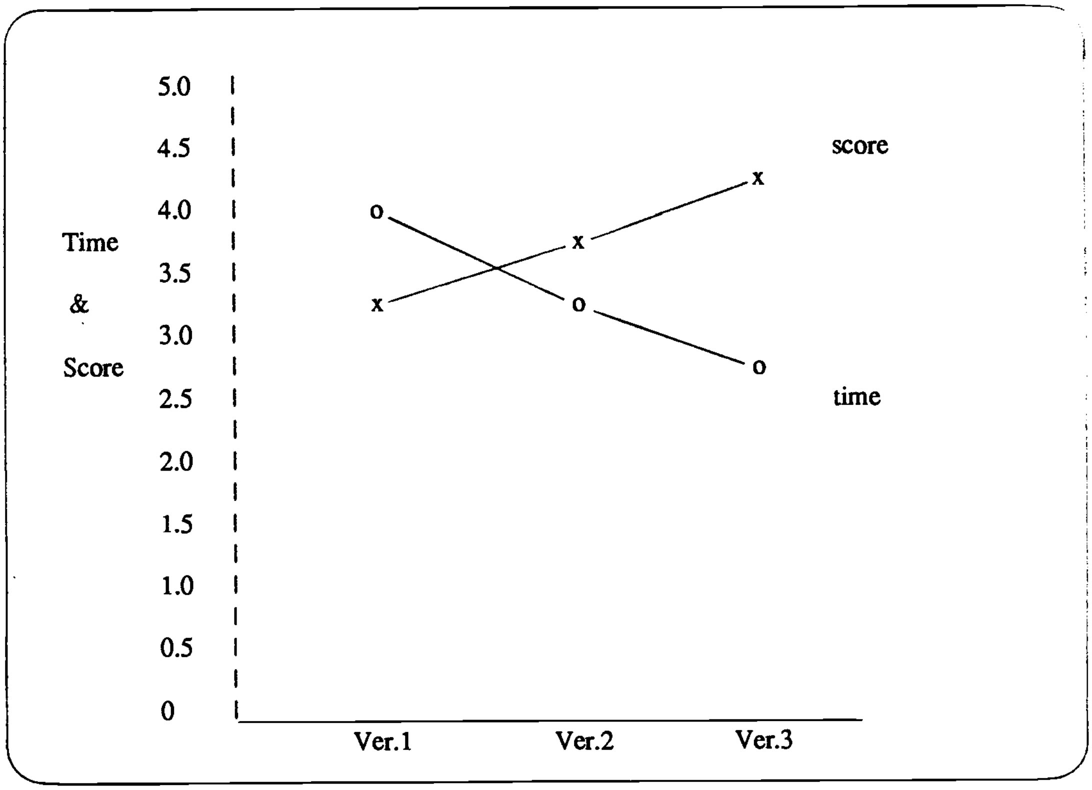


Figure 4.4. Scores and Times from First Nested IF Experiment

4.3.2. Second nested IF experiment.

To test our assertion that typographic spatial arrangement should highlight the structural arrangement of code, and to illustrate the implications on code formatting tools like syntax directed editors and pretty-printers, we repeated the nested IF experiment using three different versions of IF statements. The three new versions are listed in Figure 4.5. Version 1 was formatted by Macintosh Pascal, a syntax directed Pascal editor/interpreter for the Apple Macintosh. Version 2 is the "best" of the Kernighan & Plauger versions (version 3 in the previous experiment). And version 3 is a sequential implementation of version 2 formed by eliminating the ELSE clauses and repeating logical conditions as necessary.

```
Macintosh Pascal:     IF A > B THEN
                        S := 1
                      ELSE IF A = B THEN
                        IF C > D THEN
                          S := 2
                        ELSE
                          S := 3
                        ELSE IF C > D THEN
                          S := 4
                        ELSE IF C = D THEN
                          S := 5
                        ELSE
                          S := 6;

K & P's Version 3:    IF         A>B                THEN  S := 1
                      ELSE  IF  (A=B) AND (C>D)     THEN  S := 2
                      ELSE  IF   A=B                THEN  S := 3
                      ELSE  IF               C>D    THEN  S := 4
                      ELSE  IF               C=D    THEN  S := 5
                      ELSE                                S := 6;

Sequential IF's:      IF         A>B            THEN S := 1;
                      IF  (A=B)  AND  (C>D)     THEN S := 2;
                      IF  (A=B)  AND  (C<=D)    THEN S := 3;
                      IF  (A<B)  AND  (C>D)     THEN S := 4;
                      IF  (A<B)  AND  (C=D)     THEN S := 5;
                      IF  (A<B)  AND  (C<D)     THEN S := 6;
```

Figure 4.5. IF Statements for Second Nested IF Experiment

Version 1 represents the typical nested indentation commonly produced by code formatters. Its spatial arrangement does not portray the underlying code structure. Version 2 represents an intelligent method of using embedded spacing and alignment to highlight the 1:1 relationship between conditions and actions. Version 3 represents the most complete implementation in terms of displaying the 1:1 relationship. It has no nested statements and thus represents the most primative (inefficient) implementation but, as will be seen, the best in terms of program comprehension.

The differences between version 1 and version 2 are typographic and structural style combined. The indentation, use of embedded spacing, and logical structure are different. Version 3 was written in roughly the same typographic style as version 2, only the ELSE clauses are missing and the logical conditions are repeated. Hence, there are slight typographic and structural differences between the two.

Except for the new IF structures, the materials used in this experiment were identical to that used in the first experiment. That is, the test contained five short answer and two subjective questions concerning the content and form of the IF structures. The test was then administered to 33 intermediate computer science students. Procedures were identical to the first experiment except that treatment groups consisted of 11 students. Again, the independent variable was the "style" of the IF statements and the dependent variables were score, time, and subjective ratings.

Average scores, times, and ratings for all three groups are shown in Table 4.3. As expected, both scores and times improve across the three versions as illustrated in Figure 4.6. Again, the subjective ratings show little change across the styles. Analysis of the data showed significant differences for both score (F=5.02, p<.01, d.f.=2,30) and time (F=3.12, p<.05, d.f.=2,30) [Dixo81]. What is surprising is the relatively large time and score differences for such a short and small task.

The differences observed emphasize the importance of the effects of style on program comprehension. Version 1, formatted by a widely used syntax directed editor, actually obscures the nature of the code. The style in versions 2 and 3, although not widely used, enhances the code by supplying spatial cues to the programmer.

Table 4.3.

Results from Second Nested IF Experiment

	Version		
Averages:	Mac Pas.	K&P #3	Sequential
test score	3.36	3.90	4.90 *
test time	2.87	2.52	1.93 **
indentation rating	3.63	3.18	3.45
structure rating	3.45	2.90	3.45

* significant: (F=5.02, p<.01, d.f.=2,30)
** significant: (F=3.12, p<.05, d.f.=2,30)

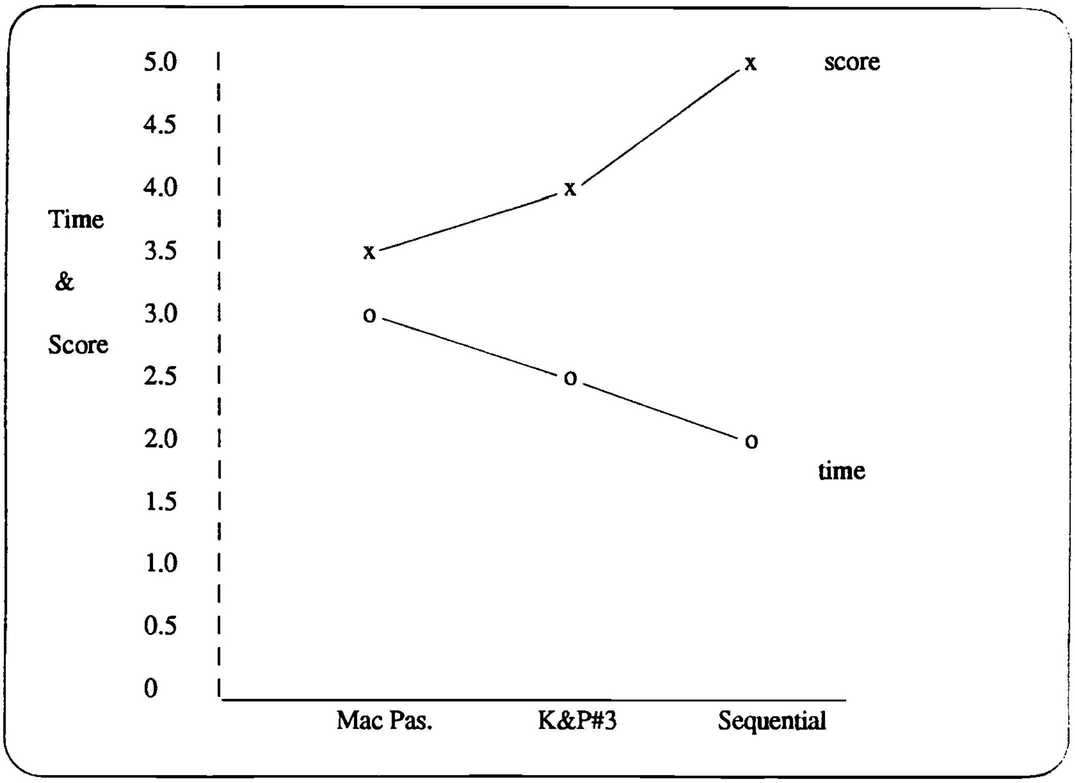


Figure 4.6. Scores and Times from Second Nested IF Experiment

It is worthy to note that many formatting tools are designed to produce code that is pretty (i.e., pleasing to the eye) with little or no attempt to highlight the visual cues that programmers use to recognize the underlying code structure. Kernighan and Plauger's first version and the Macintosh Pascal version are examples of this approach. They display the nested structure of the IF statements but obscure the 1:1 relationship between logical condition and action. In this instance, recognition of the 1:1 relationship provides the key to understanding the structure, while nesting is almost irrelevant. To our knowledge, no existing syntax-directed editor or pretty- printer can effectively use embedded spaces and indentation to spatially arrange code to display its underlying code or information structure (e.g., produce Kernighan and Plauger's second or third versions).

One further point should be raised concerning the difference between Kernighan and Plauger's third version and the sequential implementation of that same code. The improvement in comprehension comes at the cost of machine efficiency. This is evidence of the efficiency versus maintainability trade-off (analogous to the classic time versus space trade-off) which has frequently been discussed but rarely demonstrated.

4.4. Conclusions.

Results from the experiments presented in this chapter demonstrate the importance and value of taxonomic analyses of programming style. From our studies we were able to define useful typographic metrics, determine why some styles facilitate comprehension, and identify some principles governing the use of typographic style. (For example, the importance of using spatial arrangement to highlight the underlying code structure.)

In the next chapter we pursue this line of reasoning by analyzing theories of programmer comprehension and maintenance activity, to see how typographic style can be used to improve readability. We identify several principles of good typographic style that are compatible with comprehension and maintenance behavior.

Chapter 5

Towards Better Typographic Style

5.1. The problems with today's typographic style.

Yourdon, in his book *Techniques of Program Structure and Design*, [Your75] lists seven major problems facing maintenance programmers. Number six on his list concerns programming style:

> "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."

As discussed earlier, much of this disorganization stems from the subjectivity in style guidelines. Furthermore, todays pretty-printers and syntax directed editors are no help because their output is designed to be pretty, with little thought as to whether it really aids programmer comprehension. What is needed is a method of typographic formatting consistent and compatible with programmer comprehension strategies and maintenance activity.

Principles of good typographic style need to be identified, and methods of implementing those principles need development and tested in empirical studies. In the next two subsections we review the literature on program comprehension theory and maintenance behavior. We then introduce a generic comprehension and maintenance activity model and identify principles of good typographic formatting. These principles are implemented and tested in subsequent chapters.

5.2. An overview of programmer comprehension theory.

The role of mental schemata in computer science is not well understood. However, the importance of "chunks," "plans" and "beacons" is now recognized as an integral part of the process in understanding and remembering code.

Studies on general problem solving behavior have demonstrated expert-novice differences in the representation and recall of textual problems [Chas73, ChiF81, DeGr65, Maye82, Scho82]. Experts perceive problems on the basis of "deep structure" which represents the principles used to solve the problem, while novices organize the same problems on the basis of "surface structure", determined by the objects and terms within the problem.

Adelson [Adel81] replicated these results using expert and novice programmers' recall of meaningful and meaningless program segments. She demonstrated that expert programmers organize their memory for meaningful program segments into logically meaningful chunks while novices organize their memory based on the syntactic similarity of the stimulus. She concludes that expert programmers organize their working memory into functional schemata based on the functional purpose of the code.

In a subsequent study, Adelson [Adel84] experimentally varied two levels of problem representation during a program related problem solving exercise. By suggesting that subjects organize their thinking in a specific manner, she was able to induce either an "abstract mental set", based on the functional utility of a program, or a "concrete mental set", representing how the program functions. She concludes that expert programmers use a superior method of programming problem representation (i.e., a plan) based on the functional purpose of the program under study.

Further evidence of programmers using mental plans that guide their programming comes from Soloway, et al, [Solo82, Solo84]. Given programs with missing statements, expert and novice programmers were asked to fill in the missing code. Experts looked beyond the obvious and added something that was logically necessary as required by the overall purpose of the code; novices usually added the most obvious statement without regard to code interaction.

The importance of "beacons" to guide a programmer's mental plans was first discussed in Brooks' paper *Towards a theory of the comprehension of computer programs* [Broo83]. He suggested that programmers recognize certain code structures as beacons and use those beacons for searching, chunking, and hypothesis checking. Subsequent work by Weidenbeck [Weid86] has supported Brooks' theory about beacons. She demonstrated the existence of easily recognized beacons within the code that programmers use to locate meaningful chunks of code while verifying their mental plans.

In summary we point out that all research in programmer comprehension supports the existence of: (1) mental schemata, or plans, that guide programmers' comprehension of code, (2) chunks, or meaningful units of information, that programmers use to organize and remember code, and (3) beacons, or highlighted semantic clues, that are used to direct the review and recognition of code.

5.3. An overview of program maintenance activity.

Maintenance activity is often broken down into three types: corrective, adaptive, and perfective [Bend87, Mart83]. Corrective, also called repair maintenance, entails the correction of logical flaws in a program or system [Grem84, Vess83]. Although corrective maintenance accounts for only 20 to 25 percent of the total maintenance effort [Bend87, Mart83], it has received much attention in the literature and, perhaps, is the best understood of all the maintenance types. It is generally accepted that corrective maintenance requires a detection-isolation-correction sequence of activities [Bend87, Nanj88, Oman89, Vess84]; and it has been demonstrated that once programmers can detect and isolate a fault, they can invariably correct it [Goul74, Oman89].

The other two types of maintenance activity, adaptive and perfective, address discrepancies between the existing system and new (or missing) requirements and specifications. Although 75 to 80 percent of the total maintenance effort is spent in these areas, little is known about the characteristics of this activity. For example, it is not clear if the detection- isolation-correction paradigm is applicable for this type of maintenance. It is generally accepted, however, that approximately 50 percent of the effort spent in adaptive and perfective maintenance is devoted to code comprehension.

Early work by Sheppard, et al, [Shep79] suggested that some programmers attempt to understand the entire program prior to making code changes, while others used clues to zero-in on the code segments needing change (and ignoring other code). More recently, this was supported by Vessey's [Vess85] work showing that programmers use different strategies when going about maintenance tasks on small programs; and by the work of Littman, et al, [Litt86] who conducted empirical studies of programmers doing adaptive maintenance. Through protocol analyses, they identified two strategies used by programmers: (1) a systematic strategy where the programmer trys to understand how the whole program works, and (2) an as-needed strategy that attempts to minimize the time spent in program comprehension.

Further support for the existence of multiple strategies appears in Letovsky's [Leto86] protocol analyses of professional programmers. The programmers were asked to perform an enhancement to a program. While studying the program, transcripts of their "thinking aloud" protocol were gathered. These transcripts point out two interesting aspects to programmer maintenance behavior: (1) programmers frequently form and test conjectures about the code under study (consistent with the discussion in the previous subsection), and (2) programmers browse through code with multiple access paths while formulating and testing assertions.

In summary we point out that different types of maintenance activity place different requirements on the programmer. For instance, corrective maintenance may require little or no overall comprehension, while perfective maintenance may require both overall comprehension and focused, as-needed, on-the-spot comprehension of code segment details. Program size is another factor affecting this behavior. It is not clear how program comprehension and maintenance behavior changes as program size increases, but it is doubtful that an overall comprehension strategy could be employed when working with more than 10,000 lines of code.

5.4. A generic comprehension and maintenance model.

It is clear that understanding a computer program is a multidimensional process; there is no single strategy or approach that can be employed throughout all problem and implementation domains. For this reason, we have developed a generic comprehension/maintenance activity model. The following model is consistent with other published models [Mart83, Shne86], and applies to all types of comprehension and maintenance work:

1. Analysis of the requirements and specifications for comprehension and/or maintenance (i.e., the underlying responsibility and motivation).

2. A heuristically guided analysis of the program code. This may be top-down, bottom-up, pin-point focused, or simply browsing. It results in some degree of comprehension, whether overall or as-needed; the strategy and access path is dependent upon problem domain, implementation domain, and individual differences.

3. Isolation of code segment(s) requiring further study and/or change. (This may require an iterative cycle with # 2).

4. Formulate plans, conjectures, design modifications (if necessary) and check for ramifications (side effects). (This may require an iterative cycle with # 3).

5. Implement modifications as required. (maintenance only)

6. Revalidate program(s). (maintenance only)

7. Update documentation and libraries. (maintenance only)

We wish to emphasize two points: (1) it is estimated that maintenance programmers spend between 47 and 62 percent of their time trying to comprehend code [Pari83], and (2) comprehension and maintenance calls for programmers to look through source code using a variety of access paths. Hence, typographic formatting should, at the very least, assist in program comprehension and permit a variety of traversals and access mechanisms.

5.5. Principles of typographic style.

Programmers use multiple strategies and multiple access paths, all guided by a variety of plans and conjectures, when working with non-trivial programs. This interaction is dependent upon: (1) individual differences, (2) the application at hand, and (3) the implementation of the code and supporting system. Good typographic formatting of source code should support (or be compatible with) all of these recognized differences.

Using our generic comprehension/maintenance activity model as a reference point, we identified several general principles of good typographic style consistent with our model. As per our taxonomy, they are divided into macro- and micro-typographic principles:

Macro-typographic:

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 areas of 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).

Micro-typographic:

5. Make the sections and organization of the module obvious. This means dividing modules into easily recognizable parts (e.g., constants, data declaration, and code body).

6. Identify the purpose and use of each section.

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

8. 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. It is not meant to be a complete list, nor does it address the implementation of these principles;

rather, it demonstrates a separation of principle and implementation. Without concern for implementation techniques we have enumerated several principles of good typographic style that are consistent with all models of programmer comprehension and maintenance activity. Note that all of the above principles can 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 chapter we introduce the "book" paradigm, a mechanism of implementing the principles outlined above. We demonstrate, through code examples, that principles of good typographic style can be achieved by arranging source code in the form of a book.

Chapter 6

The Book Paradigm for Typographic Style

6.1. Introduction.

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

In this chapter we define and describe the book paradigm for source code formatting. Formatting source code like a book is an implementation technique that incorporates all of the principles of good typographic style outlined in the previous chapter. It is a familiar and easily understood paradigm. It should not be confused with Knuth's style of "literate programming" [Ben86a, Ben86b, Knut84], which calls for a change in the process of programming. Knuth is advocating a style of programming that: " allows the programmer to think at a high level, and has the computer do the dirty work of translating the literate description into an executable program." [Ben86a].

The result of Knuth's literate programming process is a book-like description of the analysis, design, and implementation of a program. 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 advocating a change in how source code is formatted, not a change in the whole programming process.

6.2. The book 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 are all designed to facilitate rapid information access and transfer:

- Preface -- an introduction to the book, from the author to the reader; similar to introductory header comments in a program.

- Table of Contents -- a high-level "map" of the book's contents; similar to a structure chart showing the main components of a program.

- Indices and Pagination -- low-level "maps" of the book's contents; analogous to program cross-reference maps with line numbers.

- 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.

- 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).

- Paragraphs -- Chunks of information in the form of grouped sentences; similar to nested or related programming statements (e.g., loops, ifs, cases).

- Sentences -- Statements and queries delimited and defined by punctuation, type style, character case, etc.; analogous to programming statements and declarations.

- 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.

As demonstrated from the above list, there are 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 (e.g., it is trivial to distinguish between names, sentences, paragraphs, etc.). 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.

The book paradigm of source code formatting incorporates both macro-typographic style and micro-typographic style. By definition, it does not change the control flow or information structure of the program. That is, it is an entirely typographic arrangement of program source code. Hence, this process can be automated and is roughly analogous to an expert code formatter.

6.3. Macro-typographic book paradigm formatting.

Macro-typographic factors used in the book paradigm for source code formatting 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. -- essentially the program header comments -- which can be generated by the host compiling system.

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 the chapter breakdowns. Chapters can be 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.

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. There are a variety of ways to create the index and table of contents: (1) it could be built into the host compiling system, (2) as a part of a language directed editor, (3) as a function of a pretty- printer, or (4) as part of a version control archiving and librarian system. In any case, the key to the viability of the index and table of contents is that they are inserted (as comments) into the source code file that corresponds to the executable object file.

6.4. Micro-typographic book paradigm formatting.

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

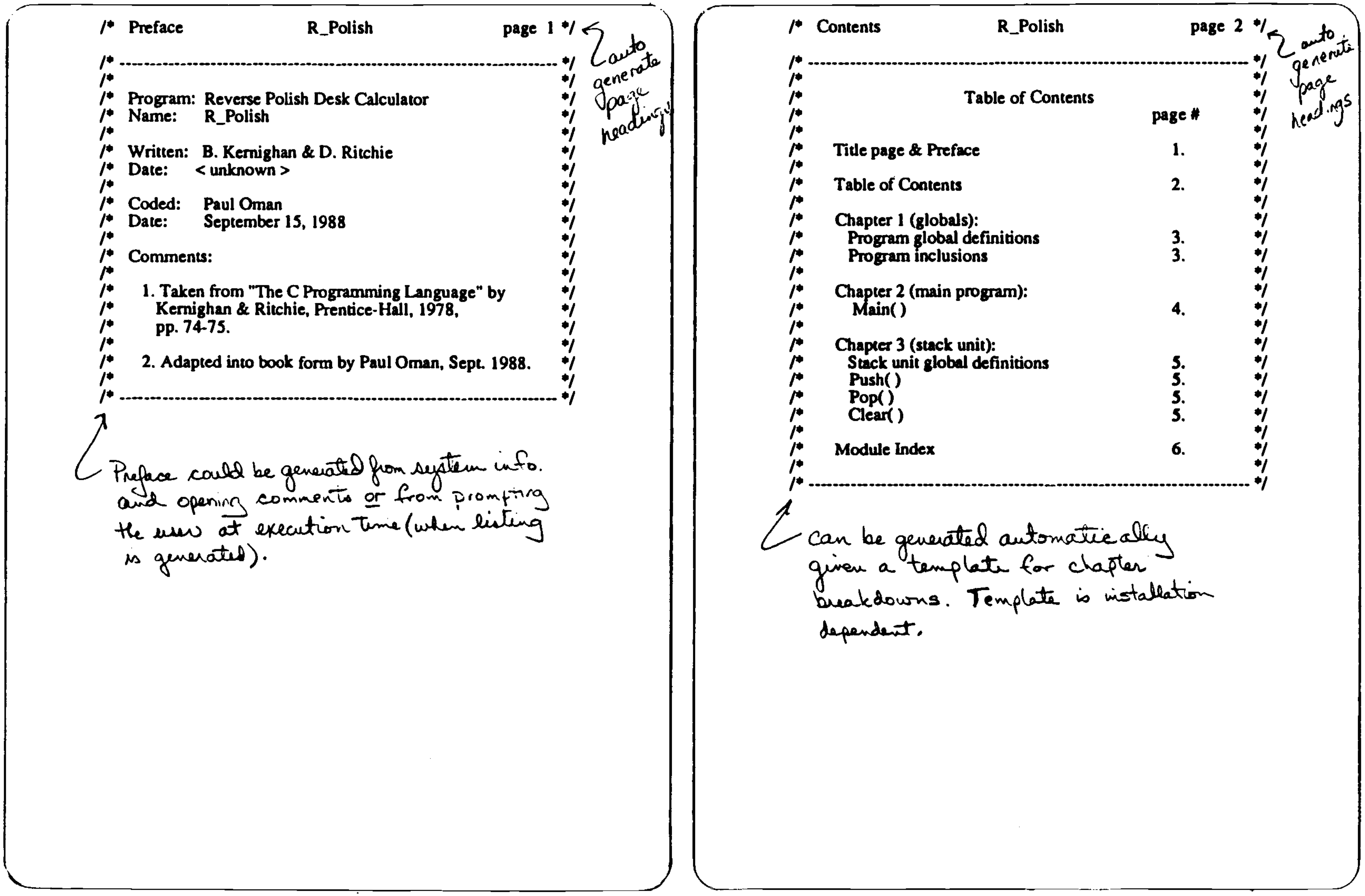
Code sections can be separated into easily recognizable units by using blanks, beacons, alignment, and in-line comments to show the beginning and ending of the code sections. For example, the Pascal Const and Var sections could be delimited by placing those reserved words in boldface (or all capitalized letters) on separate lines preceded with 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 (note that this 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, and (5) highlight well-defined code segments like data declaration areas, and (6) highlight globally defined identifiers. There are many such micro-typographic factors that could be used by intelligent source code formatting programs to aid program comprehension. As in our macro- typographic formatting, the implementation could occur in a compiling system, an editor, a printing utility, or a version control system. The key is to maintain consistency between the source code file, the object code file, and the printed code listing.

6.5. An example "book" program listing.

An example "book" for a C program appears in Figure 6.1. The program is a reverse Polish desk calculator taken from [Kern78, pp. 74-75] and rewritten into book form. This small annotated example illustrates most of the macro- and micro-typographic implementations of our typographic style principles. That is, using the book paradigm as a model for source code formatting gives us the means for implementing our principles of good typographic style.



```
/*  Preface                R_Polish                page 1 */
/* ------------------------------------------------------ */
/*                                                        */
/*  Program:  Reverse Polish Desk Calculator              */
/*  Name:     R_Polish                                    */
/*                                                        */
/*  Written:  B. Kernighan & D. Ritchie                   */
/*  Date:     < unknown >                                 */
/*                                                        */
/*  Coded:    Paul Oman                                   */
/*  Date:     September 15, 1988                          */
/*                                                        */
/*  Comments:                                             */
/*                                                        */
/*    1. Taken from "The C Programming Language" by       */
/*       Kernighan & Ritchie, Prentice-Hall, 1978,        */
/*       pp. 74-75.                                       */
/*                                                        */
/*    2. Adapted into book form by Paul Oman, Sept. 1988. */
/*                                                        */
/* ------------------------------------------------------ */
```

auto generate page headings

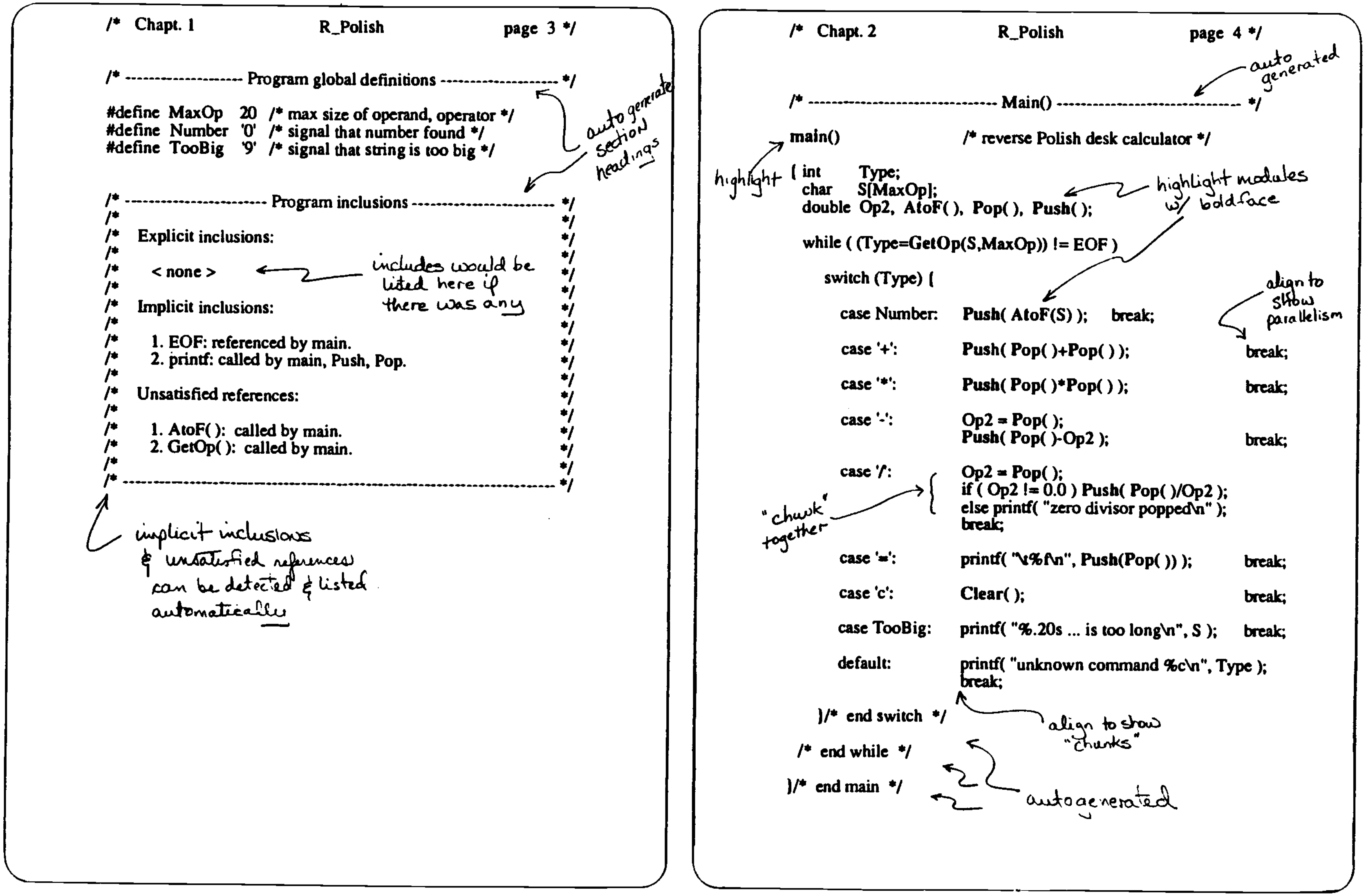
Preface could be generated from system info. and opening comments or from prompting the user at execution time (when listing is generated).

```
/*  Contents               R_Polish                page 2 */
/* ------------------------------------------------------ */
/*                                                        */
/*                  Table of Contents                     */
/*                                          page #        */
/*                                                        */
/*   Title page & Preface                     1.          */
/*                                                        */
/*   Table of Contents                        2.          */
/*                                                        */
/*   Chapter 1 (globals):                                 */
/*     Program global definitions             3.          */
/*     Program inclusions                     3.          */
/*                                                        */
/*   Chapter 2 (main program):                            */
/*     Main( )                                4.          */
/*                                                        */
/*   Chapter 3 (stack unit):                              */
/*     Stack unit global definitions          5.          */
/*     Push( )                                5.          */
/*     Pop( )                                 5.          */
/*     Clear( )                               5.          */
/*                                                        */
/*   Module Index                             6.          */
/*                                                        */
/* ------------------------------------------------------ */
```

auto generate page headings

can be generated automatically given a template for chapter breakdowns. Template is installation dependent.

Figure 6.1. An Example Book Format Listing

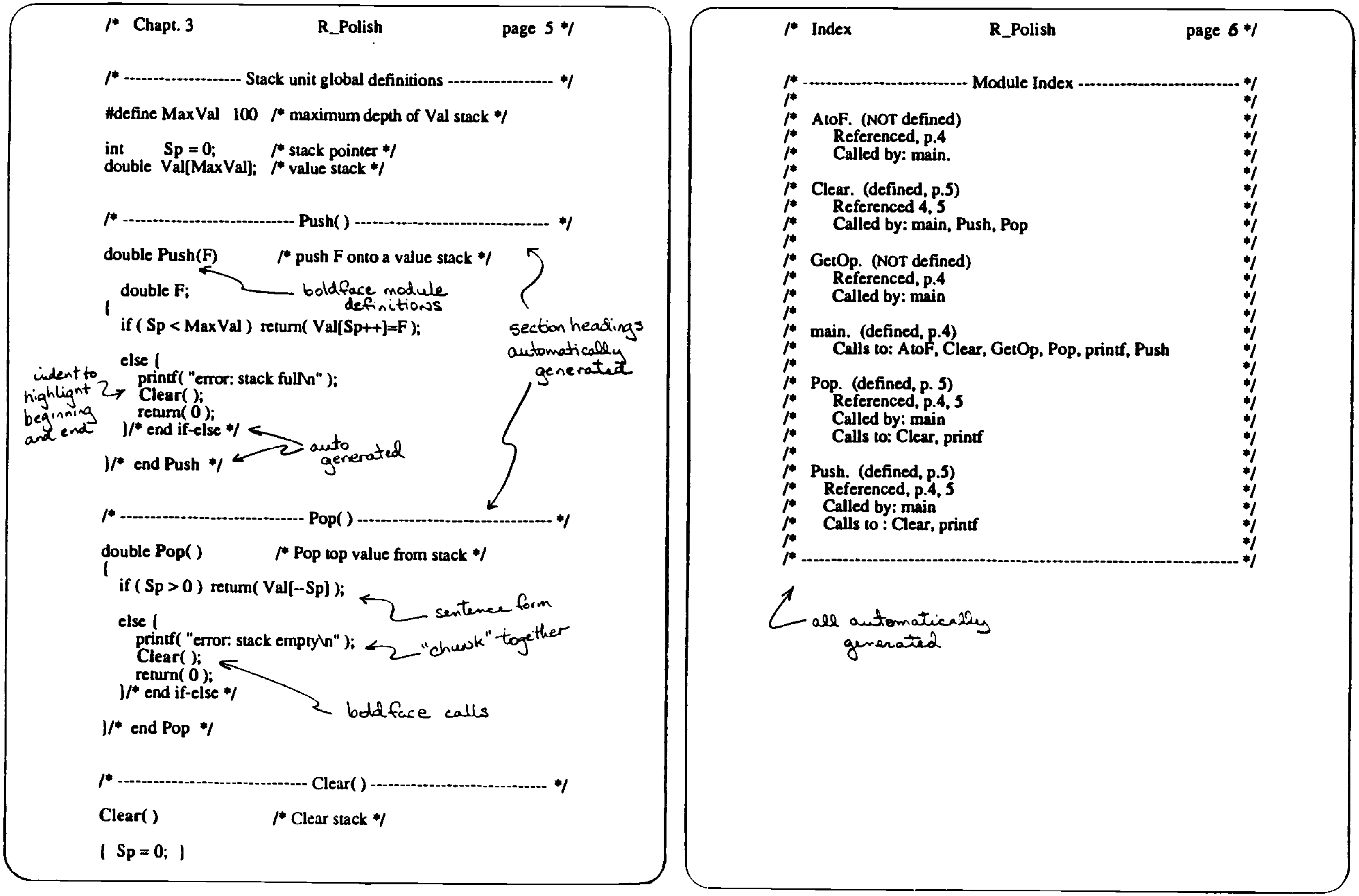


```
/*  Chapt. 1                    R_Polish                    page  3 */

/* ------------------ Program global definitions ------------------- */

#define  MaxOp    20  /* max size of operand, operator */
#define  Number  '0'  /* signal that number found */
#define  TooBig  '9'  /* signal that string is too big */

/* ---------------------- Program inclusions ---------------------- */
/*                                                                 */
/*  Explicit inclusions:                                           */
/*                                                                 */
/*    < none >                                                     */
/*                                                                 */
/*  Implicit inclusions:                                           */
/*                                                                 */
/*    1. EOF: referenced by main.                                  */
/*    2. printf: called by main, Push, Pop.                        */
/*                                                                 */
/*  Unsatisfied references:                                        */
/*                                                                 */
/*    1. AtoF( ):  called by main.                                 */
/*    2. GetOp( ):  called by main.                                */
/*                                                                 */
/* --------------------------------------------------------------- */
```

auto generate section headings
includes would be lited here if there was any
implicit inclusions & unsatisfied references can be detected & listed automatically

```
/*  Chapt. 2                    R_Polish                    page  4 */

/* ------------------------------ Main() ------------------------------ */

main()                        /* reverse Polish desk calculator */

{ int     Type;
  char    S[MaxOp];
  double  Op2,  AtoF( ),  Pop( ),  Push( );

  while ( (Type=GetOp(S,MaxOp)) != EOF )

     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;

        case '=':       printf( "\t%f\n", Push(Pop( )) );       break;

        case 'c':       Clear( );                               break;

        case TooBig:    printf( "%.20s ... is too long\n", S );  break;

        default:        printf( "unknown command %c\n", Type );
                        break;

     }/* end switch */

  /* end while */

}/* end main */
```

Figure 6.1. An Example Book Format Listing (continued)



```
/*  Chapt. 3                    R_Polish                    page 5 */

/* ------------------ Stack unit global definitions ----------------- */

#define MaxVal   100  /* maximum depth of Val stack */

int       Sp = 0;          /* stack pointer */
double  Val[MaxVal];   /* value stack */


/* --------------------------- Push( ) ------------------------------- */

double Push(F)          /* push F onto a value stack */

   double F;
{
   if ( Sp < MaxVal ) return( Val[Sp++]=F );

   else {
      printf( "error: stack full\n" );
      Clear( );
      return( 0 );
   }/* end if-else */

}/* end Push */


/* ---------------------------- Pop( ) -------------------------------- */

double Pop( )             /* Pop top value from stack */
{
   if ( Sp > 0 ) return( Val[--Sp] );

   else {
      printf( "error: stack empty\n" );
      Clear( );
      return( 0 );
   }/* end if-else */

}/* end Pop */


/* ---------------------------- Clear( ) ----------------------------- */

Clear( )                    /* Clear stack */

{ Sp = 0; }
```

```
/*  Index                    R_Polish                    page 6 */

/* ------------------------- Module Index ------------------------- */
/*                                                                  */
/*  AtoF.  (NOT defined)                                            */
/*     Referenced, p.4                                              */
/*     Called by: main.                                             */
/*                                                                  */
/*  Clear.  (defined, p.5)                                          */
/*     Referenced 4, 5                                              */
/*     Called by: main, Push, Pop                                   */
/*                                                                  */
/*  GetOp.  (NOT defined)                                           */
/*     Referenced, p.4                                              */
/*     Called by: main                                              */
/*                                                                  */
/*  main.  (defined, p.4)                                           */
/*     Calls to: AtoF, Clear, GetOp, Pop, printf, Push              */
/*                                                                  */
/*  Pop.  (defined, p. 5)                                           */
/*     Referenced, p.4, 5                                           */
/*     Called by: main                                              */
/*     Calls to: Clear, printf                                      */
/*                                                                  */
/*  Push.  (defined, p.5)                                           */
/*    Referenced, p.4, 5                                            */
/*    Called by: main                                               */
/*    Calls to : Clear, printf                                      */
/*                                                                  */
/* ---------------------------------------------------------------- */
```

boldface module definitions
section headings automatically generated
indent to highlight beginning and end
auto generated
sentence form
"chunk" together
boldface calls
all automatically generated

Figure 6.1. An Example Book Format Listing (continued)

Macro-typographic style principles implemented in Figure 6.1 include:

1. Program heading comments are in a preface to the reader.

2. A table of contents, which is a map to the following code, is included to give the reader a top-level overview of the code contained in the source file.

3. The program code is separated into chapters for globals and includes, the main body, and the stack unit support routines and definitions. This is not a reordering of the code, just a partitioning of the original ordering.

4. Components are separated by blank lines and a comment line containing the component (or module) name. The end of modules are highlighted by comments.

5. A module index with a cross-reference list of calls is included at the end.

Micro-typographic style principles implemented in Figure 6.1 include:

1. Statements are written as sentences whenever possible. This is merely a preference for horizontal (versus vertical) formatting; for example, IF statements are written on one line when possible.

2. Related and/or nested statements are chunked (sometimes horizontally), with blank lines separating chunks.

3. Parallel constructs and tabular structures are aligned (through indentation and embedded spacing) to show relationships.

4. Compound statements are aligned in Ledgard's "comb structure" [Ledg87]. That is, the first and last line of the body are aligned and the intervening lines are all indented 2 to 4 spaces in from them. This is a method of highlighting the beginning and ending of a chunk. Here, we also highlight the end of statements with comments.

5. Module names are highlighted in boldface; identifiers are treated as proper names (highlighted by beginning capital letters).

6. Statement operators are delimited by spaces except when part of an argument. That is, expressions are written without embedded spaces when they are parameters; this is a form of horizontal chunking.

We emphasize that this example is just a typographic rearrangement of the original program. We suggest that it is an obvious and natural form for listing source code and places no additional burden on the programmer. Furthermore, it is (fairly) language independent and improves program comprehension (as demonstrated in the next chapter) by providing structural clues and multiple access paths.

6.6. Conclusions.

Analyses of programmer behavior clearly show a multitude of strategies and approaches used when working with source code. It is important to note that the source code file (program listing) is usually the only reliable documentation that a programmer has to work with. But a program listing is essentially a linear ordering of a non-linear entity (for all but trivial and monolithic programs). Programmers need a variety of access paths into program listings. The book paradigm provides this.

Organizing program source code into a book format provides programmers with: (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.

In the next chapter we demonstrate, through controlled experiments and empirical studies, the value of our principles of good typographic style. These principles are encapsulated into the book paradigm.

Chapter 7

Testing the Book Program Listing

7.1. Introduction.

In this chapter we present four studies that demonstrate the benefits of the book paradigm of typographic source code formatting. Specifically, we 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 paradigm is an easy and natural representation of source code that improves progammer comprehension and performance.

7.2. Testing macro-typographic principles.

In order to test our assertions about the book paradigm for macro-typographic style we conducted a controlled study measuring students ability to perform maintenance tasks using two different versions of a large Pascal program -- one a traditional listing, the other our book paradigm listing.

7.2.1. Experiment 1: A macro-typographic comparison.

Materials: The basis for the maintenance exercise was a 1543 line Pascal program taken from [Schn81, pp. 379-415]. The program is a working text editor. The program was modified by removing the Skip_Blanks procedure and the five calls to it. The Skip_Blanks procedure (shown in Figure 7.1) was used to skip over leading blanks at various places on the editor command input line. The resulting modified program still worked; it was just incapable of handling free-form command inputs. The program was then reduced to 1011 lines of code 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.)

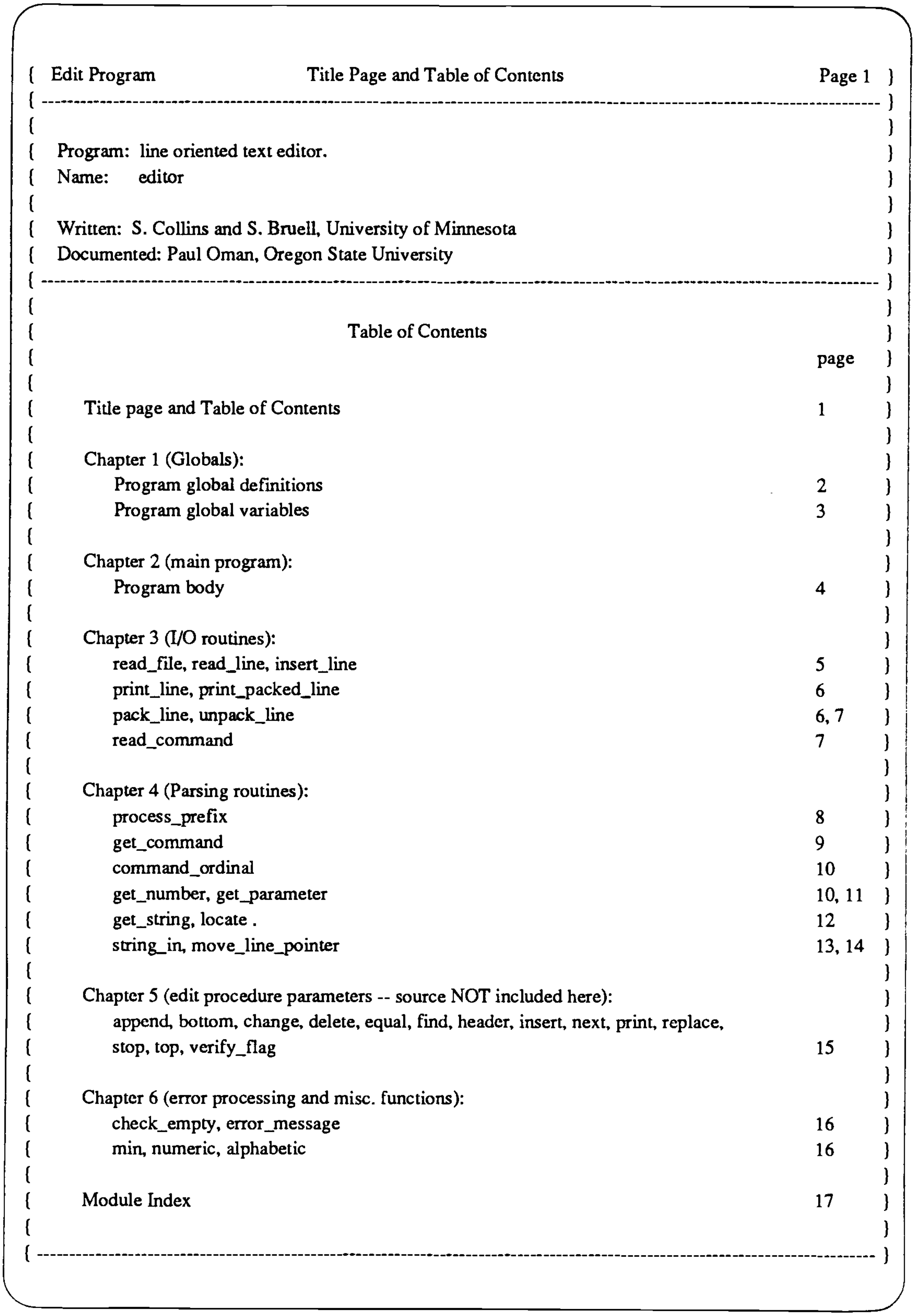
```
procedure skip_blanks ( var line : line_def );
begin
    with line do
        while ( position <= length )
            and (chars[position] = ' ') do
                position := position + 1;
end; {procedure skip_blanks}
```

Figure 7.1. Experiment 1: Skip_Blanks Procedure

The modified program was then ported into 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 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. The table of contents for the book listing appears in Figure 7.2.

Experimental Task: Subjects were given one of the program versions, asked to 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 maintenance exercise was essentially to restore the original program 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.

Upon completion of the maintenance task subjects were asked to record the time from the clock at the front of the room. Thus, the independent variable was the macro-typographic style of the code (traditional and book format listing). Dependent measures for each subject were: ability to write the Skip_Blanks routine, 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. The materials distributed to subjects consisted of a page of instructions followed by the listing. The page on instructions appears in Appendix C.



```
{ Edit Program            Title Page and Table of Contents                 Page 1 }
{ -------------------------------------------------------------------------------- }
{                                                                                  }
{  Program: line oriented text editor.                                             }
{  Name:    editor                                                                 }
{                                                                                  }
{  Written: S. Collins and S. Bruell, University of Minnesota                      }
{  Documented: Paul Oman, Oregon State University                                  }
{ -------------------------------------------------------------------------------- }
{                                                                                  }
{                              Table of Contents                                   }
{                                                                         page     }
{                                                                                  }
{     Title page and Table of Contents                                    1        }
{                                                                                  }
{     Chapter 1 (Globals):                                                         }
{         Program global definitions                                      2        }
{         Program global variables                                        3        }
{                                                                                  }
{     Chapter 2 (main program):                                                    }
{         Program body                                                    4        }
{                                                                                  }
{     Chapter 3 (I/O routines):                                                    }
{         read_file, read_line, insert_line                               5        }
{         print_line, print_packed_line                                   6        }
{         pack_line, unpack_line                                          6, 7     }
{         read_command                                                    7        }
{                                                                                  }
{     Chapter 4 (Parsing routines):                                                }
{         process_prefix                                                  8        }
{         get_command                                                     9        }
{         command_ordinal                                                 10       }
{         get_number, get_parameter                                       10, 11   }
{         get_string, locate .                                            12       }
{         string_in, move_line_pointer                                    13, 14   }
{                                                                                  }
{     Chapter 5 (edit procedure parameters -- source NOT included here):           }
{         append, bottom, change, delete, equal, find, header, insert, next, print, replace, }
{         stop, top, verify_flag                                          15       }
{                                                                                  }
{     Chapter 6 (error processing and misc. functions):                            }
{         check_empty, error_message                                      16       }
{         min, numeric, alphabetic                                        16       }
{                                                                                  }
{     Module Index                                                        17       }
{                                                                                  }
{ -------------------------------------------------------------------------------- }
```

Figure 7.2. Experiment 1: Editor Table of Contents

Subjects: The program listings and instructions explaining the maintenance task were given to 53 advanced computer science students enrolled in a senior/graduate level operating systems course. Subjects were randomly assigned into two treatment groups, with roughly half the class (28 subjects) receiving the traditional listing while the other half (25 subjects) received the book listing. They were asked to read the instructions, study the code, and then write the Skip_Blanks routine and indicate where (on the listing) it would be called. No special instructions or explanations were given to subjects receiving the book listing. This was deliberately done as a test to see if subjects could "naturally" use the book listing (i.e., without training).

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., 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 7.1) 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) [Dixo81].

Table 7.1.

Experiment 1: Ability to Write Skip_Blanks

	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 %

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). A Chi-square test of independence on the resulting 2 by 2 design (using Pearson's computed expectency values) shows a significant difference between the two versions (X=3.73, p<=.06, d.f.=1) [Dixo81, OttL77]. 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! Another interesting point is that 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 Skip_Blanks routine. Results for the call identification task are shown in Table 7.2. 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 7.2.

Experiment 1: Identifying Skip_Blanks Calls

Dependent measure	Traditional listing	Book listing
Number getting Skip_Blanks 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 %

No significant differences in times were observed between the two groups. The average time for the traditional listing group was 53.5 minutes, for the book listing it was 52.2 minutes. Because of the complexity of the maintenance task, many subjects were not able to complete the task in the allotted 55 minutes and those that did were just barely able to do so. Hence, the group averages are skewed towards the maximum time.

7.2.2. Conclusions from Experiment 1.

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. And, it should also be emphasized, that subjects using the book listing performed better without any explanation, description, or justification of the book listing.

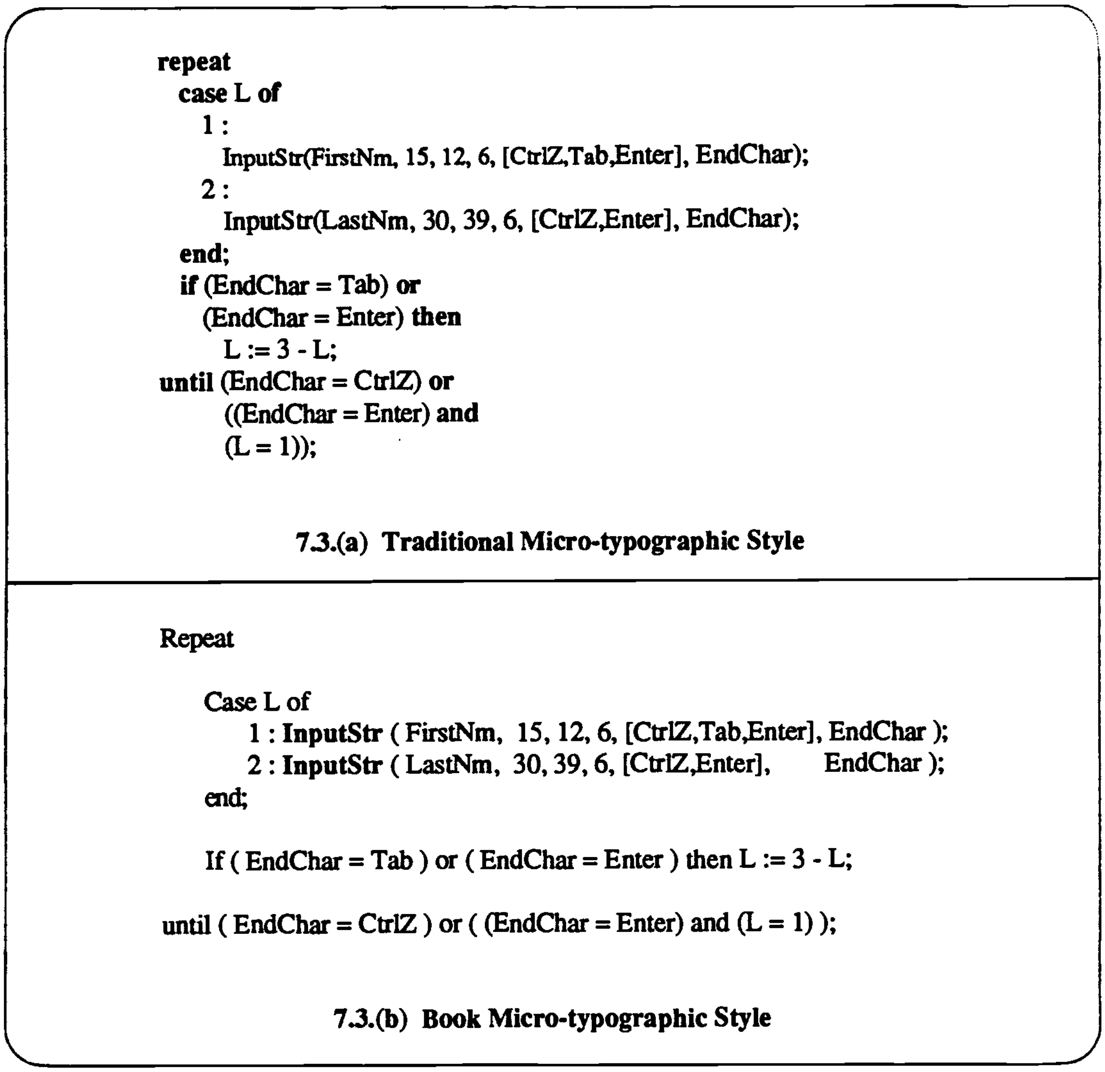
7.3. Testing micro-typographic principles.

Convinced that our book paradigm for macro-typographic style was compatible with mechanisms of programmer comprehension, we then ran some experiments to determine the affects of our micro-typographic style on the comprehension of small code segments. The following two experiments demonstrate that implementations of our micro-typographic style principles will improve comprehension of both Pascal and C source code.

7.3.1. Experiment 2: Micro-typographic Pascal formatting.

To demonstrate that our micro-typographic style is better than existing methods of Pascal source code formatting, we conducted an experiment comparing two different stylistic versions of industrial strength code.

Materials: Two procedures (94 lines of code) were extracted from Borland's Turbo Pascal Toolbox. The original code taken from the toolbox is formatted in a traditional method similar to the way Macintosh Pascal and Lightspeed Pascal formats code [Thin84, Thin86]. This was version 1. Version 2 was a typographic rearrangement of that code using our book paradigm 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 versions are listed in Figure 7.3. 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.



```
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));
```

7.3.(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) );
```

7.3.(b) Book Micro-typographic Style

Figure 7.3. Experiment 2: Pascal Code Excerpts

Experimental Task: Subjects were given one of the two code versions and asked to complete a short comprehension test using the code listing. The test consisting of 10 multiple choice and short answer questions. Some of the questions had multipart answers; consequently, there were 14 distinct (scoreable) answers for the 10 questions. Subjects were asked to complete the test, record the time necessary to do so, and then subjectively rate the readability of the code. Subjects were given 10 minutes to complete the questions.

One additional measure, the average number of correct answers per minute (score / time), was calculated for each subject as a gross measure of performance. Thus, the independent variable was the micro- typographic style of the code (traditional and book format). Dependent measures for each subject were: score (0 to 14 points), time required to answer the questions (1 to 10 minutes), performance (score / time), and a subjective rating for readability on a 5 point forced-choice scale (1-very poor, 5-very good). The materials distributed to subjects consisted of a page of instructions followed by two pages of code and then a page of questions. Appendix D contains a sample experimental packet.

Subjects: The test was administered to 36 intermediate computer science students during the first 15 minutes of a junior level data structures class. Subjects were randomly assigned into two treatment groups of 18 students; each group receiving one of the two code versions, either the traditional format or the book format. Subjects were asked to answer the questions to the best of their ability, record the time as accurately as possible (the experimenter wrote the elapsed time on the blackboard, at the front of the room, every 15 seconds), and rate the code readability. Subjects not finishing after 10 minutes were instructed to record a time of 10 minutes and then complete the readability rating.

Results: Average scores, times, performance indexes, and ratings for both groups are shown in Table 7.3. In support of our micro-typographic principles, all four measures -- scores, times, correct answers per minute, and ratings -- improve with the book format listing. This is illustrated by the plot of scores and times in Figure 7.4. Univariate analysis of variance showed significant differences for score (F=10.57, p<.005, d.f.=1,34), performance (F=8.57, p<.01, d.f.=1,34), and readability rating (F=4.45, p<.05, d.f.=1,34) [Dixo81]. Although the trend is for time to improve with the book format, these differences are not strong enough to report significance (p<.31).

Table 7.3.

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)

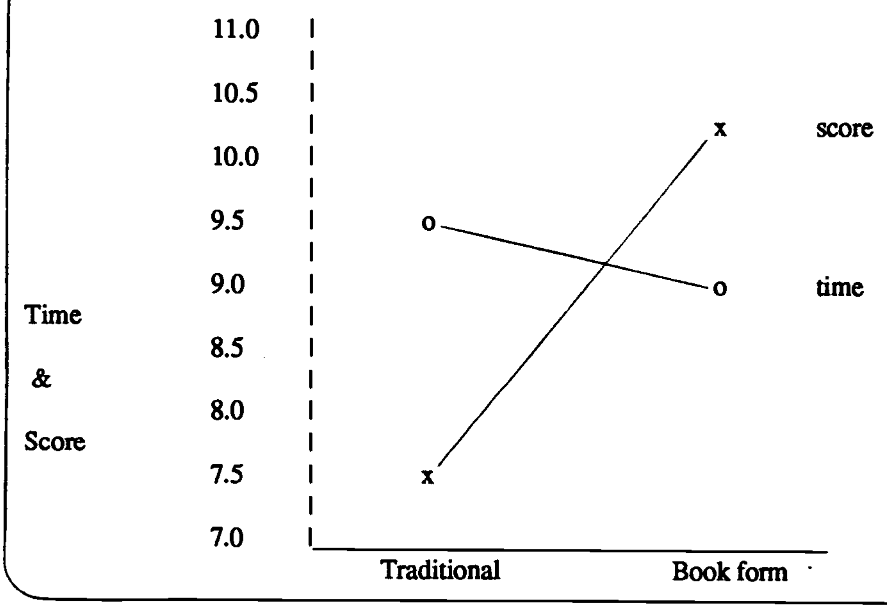


Figure 7.4. Scores and Times from Experiment 2

Group differences 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 group.

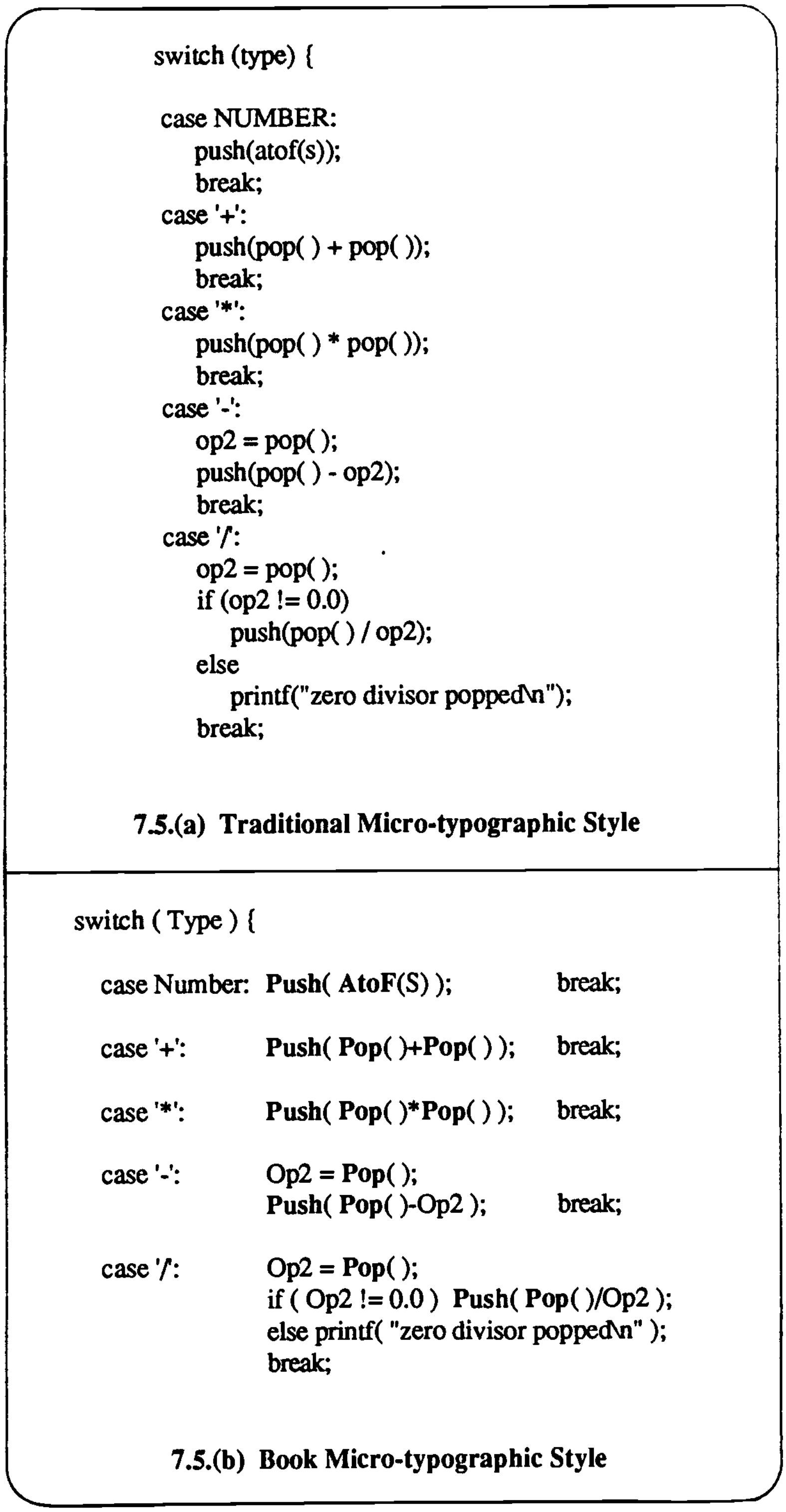
Although the readability rating difference is marginally significant, we question it's validity. This is the first (and only) experiment we conducted where subjective readability ratings differed significantly between groups. Hence, we are not convinced of students ability to judge "style" and "readability."

Experiment 2 Conclusions: 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 percent for the group (7.39 / 14). The same accuracy rate for the book listing group is 74 percent (10.39 / 14); a increase in accuracy of 21 percent. Similar ratios for the time measure show the traditional listing group used 93 percent of the available time, while the book listing group used 89 percent; a 4 percent savings in time even though they were more accurate. These comparisons demonstrate that the group working with the book listing did better and faster than those working with the traditional listing.

7.3.2. Experiment 3: Micro-typographic C formatting.

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

Materials: A reverse Polish desk calculator program written in C (illustrated in Chapter 6, Figure 6.1) was taken from Kernighan and Ritchie's book, *The C Programming Language* [Kern78, pp. 74-75]. 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 paradigm principles of micro-typographic style to guide the formatting. Excerpts from the two versions are listed in Figure 7.5. Again, we emphasize that the difference between versions 1 and 2 is entirely typographic arrangement: indentation, embedded spacing, alignment, and the use of bold-face font.



```
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;
```

7.5.(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;
```

7.5.(b) Book Micro-typographic Style

Figure 7.5. Experiment 3: C Code Excerpts

Experimental Task: Except for the materials, the procedures used in this experiment were identical to that used in the previous experiment. Subjects were given one of the two code versions and asked to complete a short comprehension test using the code listing. The test contained 9 questions concerning the content and form of the C code. One of the questions had a two part answer; hence, there were 10 distinct (scoreable) answers for the 9 questions. As in the previous experiment, subjects were asked to complete the test, record the time necessary to do so, and then subjectively rate the readability of the code. Subjects were given 10 minutes to complete the questions. The average number of correct answers per minute (score / time), was again calculated for each subject as a gross measure of performance. Again, the independent variable was the micro-typographic style of the code (traditional and book format); dependent measures for each subject were: score (0 to 10 points), time (1 to 10 minutes), performance (score / time), and readability rating (1-very poor, 5-very good). The materials distributed to subjects consisted of a page of instructions followed by two pages of code and then a page of questions. Appendix E contains a sample experimental packet.

Subjects: The test was administered to 44 advanced computer science students during the first 15 minutes of a senior/graduate level operating systems class. Subjects were randomly assigned into two treatment groups of 22 students; each group receiving one of the two code versions, either the traditional format or the book format. As in the previous experiment, subjects were asked to answer the questions to the best of their ability, record the time as accurately as possible (the experimenter wrote the elapsed time on the blackboard, at the front of the room, every 15 seconds), and rate the code readability. Subjects not finishing after 10 minutes were instructed to record a time of 10 minutes and then complete the readability rating.

Results: Average scores, times, performance indexes, and ratings for both groups are shown in Table 7.4. As expected, scores, times, and number of correct answers per minute improve with the book format listing. This is illustrated by the plot of scores and times in Figure 7.6. Analysis of the data showed significant differences for score (F=9.40, p<.005, d.f.=1,42), time (F=3.71, p<.06, d.f.=1,42), and performance (F=11.41, p<.005, d.f.=1,42) [Dixo81]. There was essentially no difference in readability rating.

Table 7.4.

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)

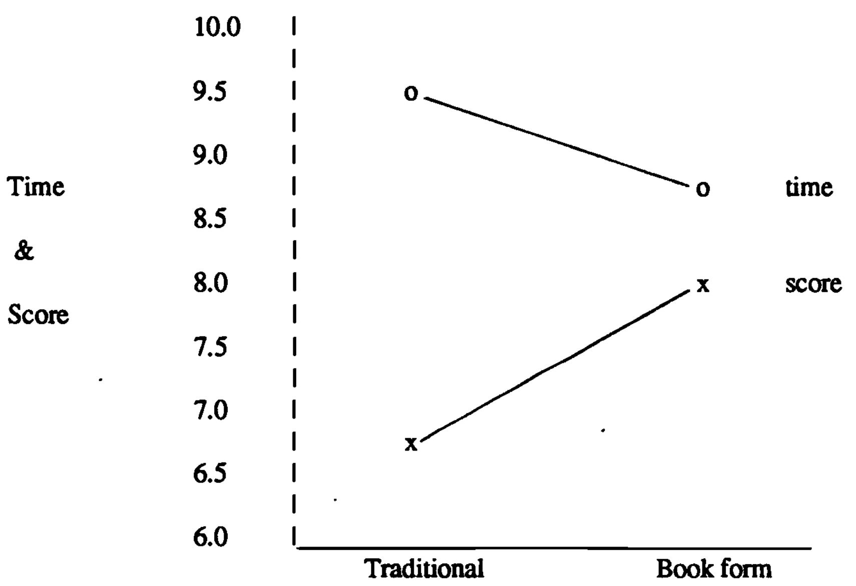


Figure 7.6. Scores and Times from Experiment 3

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 for the book 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. This knowledge probably biased their ratings.

Experiment 3 Conclusions: Group differences can again be seen by calculating the percentage difference between them. An average score of 6.73 for the traditional listing group represents an accuracy rate of 67 percent for the group (6.73 / 10). The same accuracy rate for the book listing group is 78 percent (7.89 / 10); a increase in accuracy of 12 percent. Ratios for the time measure show the traditional listing group used 95 percent of the available time, while the book listing group used 88 percent; a 7 percent savings in time while being more accurate. Again, the group working with the book listing did better and faster than those working with the traditional listing.

7.3.3. Conclusions from Experiments 2 & 3.

The results from experiments 2 and 3 provide strong evidence that our principles of micro-typographic style are compatible with programmer comprehension and, in fact, improve comprehension of small code segments. In both experiments, the groups of students working with code formatted according to our principles of micro-typographic style outperformed the groups working with traditional listings. We point out that both traditional listings were formatted in well known and widely accepted styles (Lightspeed Pascal formatting and Kernighan & Ritchie's style of C). We emphasize that the improved versions were attained by simple micro-typographic rearrangements of the original code.

7.4. Testing the complete book format listing.

Thus far we have demonstrated the value of our book paradigm for macro- and micro-typographic style in separate studies. In the next section we conduct an experiment testing the complete book format listing with professional programmers. We wanted to show that using our combined principles of macro- and micro- typographic style can improve programmer comprehension and maintenance on large programs.

7.4.1. Experiment 4: Empirical studies with "real" programmers.

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.

Materials: A portion of the X_Windows package was obtained from the Hewlett-Packard (HP) Corporation. X_Windows is a window and mouse management system originally developed at M.I.T. and now bundled with the HP Unix system which runs on the HP-9000 series minicomputers. The C code we obtained was the X_Windows Information program which consists of the xwininfo.c main program file and its two include files dsimple.h and dsimple.c. There are 1057 lines of commented C code in the three files.

Two printed listings of the X_Windows Information program were created. As in our previous experiments, version 1 was the original listing as received from Hewlett-Packard except that it was laser printed with pagination for readability. Version 2 was our typographic rearrangement of the code. The book paradigm for both macro- and micro-typographic style was used to guide the code reformatting. All changes were simple typographic alterations that could be automated. 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 rearrangement sufficiently compressed the original source code such that the end result was only 41 lines longer than the original code!

Experimental Task: Subjects were given one of the two code versions and asked to complete a four part 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 (30 minutes per part) 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 required to complete the call graph (1 to 30 minutes). 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.

We emphasize that all subjects received exactly the same instructions; that is, subjects working with the book listing received no explanation or justification about the book listing. All subjects were given a test booklet containing written instructions (and test questions) for each of the four parts. The instruction booklet appears in Appendix F.

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 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 with only the experimenter present.

Two of the subjects were deliberately chosen because they were HP maintenance programmers responsible for portions of the HP 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; hence, they were used to establish a 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 HP Unix systems experience. Background characteristics for the subjects appears in Table 7.5. The subject pairs are listed in decreasing order of HP Unix and C experience. The first pair, labeled X_t and X_b, are the two X_Windows experts.

Table 7.5.

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	HP 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.

The following four subsections explain each part of the experimental task and the results from that task.

7.4.2. Studying with "Think aloud" protocols.

Task: Subjects were given one of the two X_Windows Information 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.

The scenario described above was first used by Pennington [Penn87] as a means of establishing the motivation for studying 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. The use of "think aloud" protocols is a data gathering mechanism used by many empirical studies of programmers (see [Olso87, Solo86]).

Subjects then studied the code listing, "thinking out load," for 30 minutes. The experimenter's role was to remain unobtrusive, so as to not 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.

Results: As expected, the 12 subjects showed a variety of study patterns. Individual differences will not be reported here; rather, we will limit our discussion to clearly observable trends that either characterize all subjects or serve to separate the six with the book listing from the six with the traditional listing. All subjects:

1. Initially reported that the code seemed well structured and consistent. Later in the study period they all pointed out areas where they thought the overall organization was incorrect and the (detailed) code could have been 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. Some browsed everything and then did detailed study, others did some detailed study first (on global definitions and the main routine) and then browsed. 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. That is, all subjects skipped (never studied) certain modules. The choice of modules to skip was determined by the name (e.g., "Get_Error, I can ignore that.") or it's location (e.g., "Oh, this is the dsimple include file, I don't need to look at these.").

5. Studied the main program body and it's support routines prior to studying the included 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 included code.

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

1. Noticed and commented on the "documentation" provided by the table of contents and module index. All subjects receiving the traditional listing expressed a desire to have a cross reference map, while all those who got the book listing 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. In detailed study they used the index to trace intermodule execution flow. While browsing they checked the table of contents and/or the index to see if any related modules existed in the code (relationships were determined by name or location).

3. Maintained separate piles of code pages with the index in one pile and the table of contents face-up off to one side. The rest of the code was divided into separate groups; usually the main routine and it's support code was separated from the included code, but there were other orderings.

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."). It's interesting to note that four of the six programmers working with the traditional listing went through and highlighted module names by underlining and/or circling them. All three indicated that they normally did this when working with printed listings.

5. Noticed and commented on the use of horizontal statement spacing (sentential form) 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. (Two exceptions were X_b and E_b who made no such claim). It's worthy to note that none of the subjects receiving the traditional listing made this claim.

At the end of 30 minutes the experimenter indicated that time was up and asked the subjects if that was a sufficient amount of time to "get a feel" for the system. Of the programmers receiving the book listing all but E_b (the least experienced) indicated that it was; only two of the programmers working with the traditional listing said the same (X_t and B_t). Four programmers (A_t, C_t, D_t, and E_b) had not competed their study of the dsimple.c include file during the 30 minutes study period. They were granted a few extra minutes to do so before going on to part 2. This was done to ensure that all programmers had at least seen all the modules in the system prior to the comprehension test.

7.4.3. The oral comprehension test.

Task: For this part of the experimental task subjects were given a written instruction page containing the questions and were asked to read along while the experimenter read the instructions out loud. The questions were then read out loud with the subjects verbally answering each question before going on to the next. Subjects were allowed to keep the written question sheet and could refer to it as often as desired.

The comprehension test consisted of 7 questions, three of which had two-part answers, so there was a total of 10 points on the test. The questions ranged from high-level general questions (e.g., #4 - What are the two ways users can select windows?) to low-level specific questions (e.g., #2 - Why does Select_Window_Args call Window_With_Name?). Answers for all questions are included in Appendix F.

The experimenter's role was limited to reading the question 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 as 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 (traditional or book) and the dependent variables were score (0 to 10 points) and time (1 to 30 minutes) on the oral exam.

Results: The comprehension test results are presented in matrix form in Table 7.6. The answers to individual questions are shown as well as the overall test score and time. As can be seen, programmers working with the book listing scored better, and did so faster, than the programmers working with the traditional listing. There were no noticeable patterns in the ability to answer specific questions.

A comparison between the two groups can be seen in Figure 7.7, 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.

Table 7.6.

Experiment 4: Results from Comprehension Test

	test questions										total	total
subject	1	2	3	4a	4b	5	6a	6b	7a	7b	score	time
X_t:	c	x	x	c	c	c	c	c	c	c	8	13
At:	x	c	c	c	c	x	c	x	c	x	6	18
B_t:	c	c	x	c	c	c	c	x	c	x	7	17
C_t:	x	x	c	c	c	x	c	x	c	x	5	16
D_t:	c	x	x	c	c	c	c	x	c	x	6	16
E_t:	c	x	c	c	x	x	c	x	c	c	6	26
										traditional list averages:	6.33	17.6
X_b:	c	c	x	c	c	c	c	x	c	c	8	7
A_b:	c	c	c	c	c	x	c	x	c	c	8	17
B_b:	c	c	x	c	c	c	c	c	c	x	8	10
C_b:	c	c	x	c	c	c	c	x	c	c	8	13
D_b:	c	c	x	c	c	c	c	x	c	c	8	12
E_b:	c	x	c	c	c	x	c	x	c	c	7	13
										book list averages:	7.83	12.0

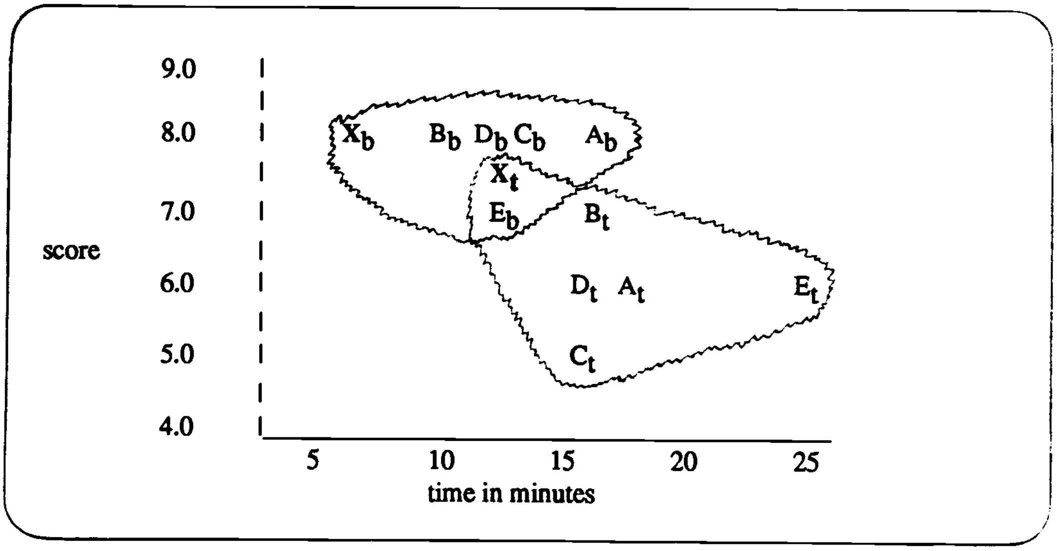


Figure 7.7. Scores and Times from Experiment 4 Comprehension Test

7.4.4. Creating a call graph.

Task: For this part of the experiment 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 represents the top-level calls from main to its support routines. 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 39 new nodes and edges are indicated by dashed lines in Appendix F.

The experimenter's role was limited to reading the instructions, tracing through the incomplete 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 for the task was the listing version (traditional or book) and the dependent variables were score (0 to 39 nodes and arcs) and time (1 to 30 minutes).

Before starting the exercise, five of the six subjects working with the book listing noticed (and verbalized) that they could use the index to complete the call graph without even looking at the code. They were told that it was an exercise in code reading and asked 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.

Results: The call graph exercise results are presented in Table 7.7. 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 be seen in Figure 7.8., 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 minute longer, than those working with the book format listing. There is little difference, however, between the two experts; their times and scores can be considered as indices of top-line performance by knowledgeable professionals. Once again, subjects working with the book format listing performed as well or better than the experts; those working with the tradtional listing performed noticeably worse.

Aside from the increased accuracy of the book format group, no noticeable differences could be seen characterizing either the version 1 or version 2 groups. There were, however, some interesting individual differences in how the programmers traced through the code while creating the call graph. As shown in Table 7.7, three programmers did complete depth-first execution traversals through the code, one did a complete breadth-first traversal, two did linear traces through the code listing, and the others did combinations of traversals and/or heuristically guided traversals. This later group jumped around the call graph based on "what seems interesting."

Table 7.7.

Experiment 4: Results from Call Graph Exercise

traditional listings				book format listings		
score	time	traversal	subject	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.

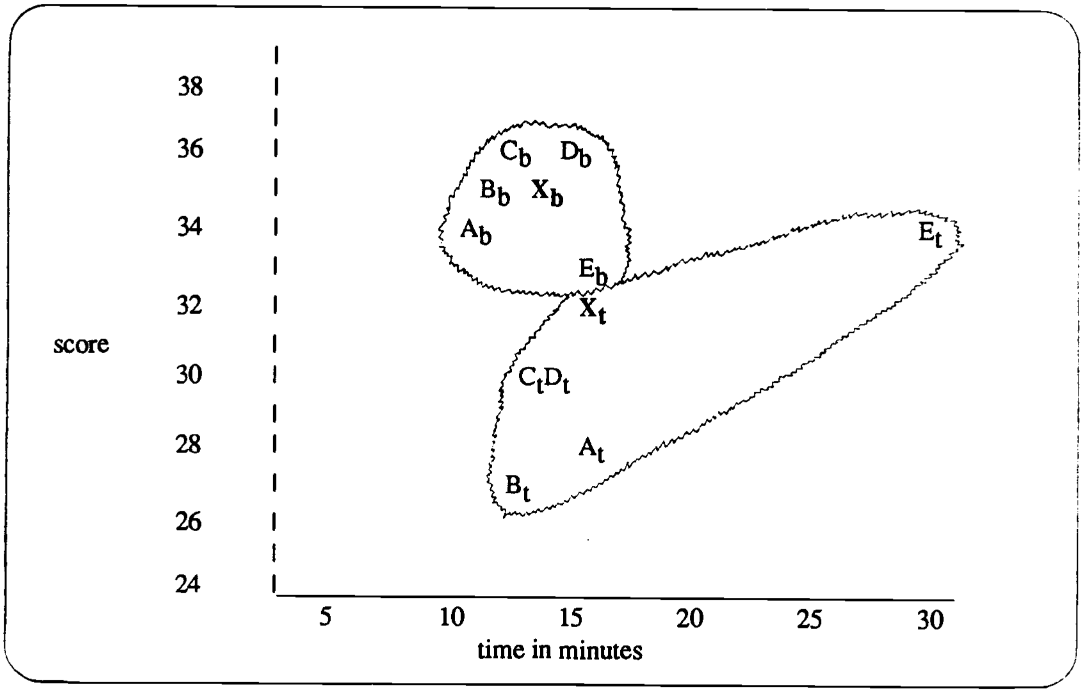


Figure 7.8. Scores and Times from Experiment 4 Call Graph Exercise

7.4.5. Responses to the open-ended questions.

Task: For this part of the experiment subjects were asked open-ended questions about how they would work with large programs. The first question asked how they would add a Display_Colors option to the xwininfo.c program; the other questions pertained to how they would work given their normal office environment, and with a large-screen multiwindowed terminal. Subjects were permitted to use the listing to assist in their answer. All answers were encouraged with positive feedback and/or requests for elaboration. The sole purpose of the task was to gain insight about how programmers go about understanding and working with programs larger than 1000 lines of code.

Results: All subjects described an adequate solution to the Display_Colors maintenance exercise. The descriptions to their solutions all centered around how they would operate at a computer terminal. That is, given an overall understanding of the program, they would all have proceeded with the maintenance task copying, cutting, pasting, and changing sections of code while running and testing the program.

When studying programs of this size and complexity in their office, all programmers indicated that they would simultaneously look at the code and watch it run. When asked how, those having multiwindow displays responded that they would set up two windows, one for the execution and one for the code view. Programmers without multiwindow displays said they'd run off a listing and a cross reference map (if it was more than 1000 lines of code) and trace through the printed listing while watching the program run. All subjects indicated they turn to printed listings when: (1) a multiwindowed environment is not available, (2) the program exceeds a few thousand lines of code, and/or (3) the code was convoluted, complex, and/or obscure.

What is particularly interesting is that all programmers expressed an occasional need to "sleep with," "get close to," or "stare at" printed listings. These are terms they used to describe the intense, detailed, and long term study of programs greater than a few thousand lines of code. All subjects indicated that on-line code reading was inadequate for large complicated systems.

There was only one noticeable difference between groups on how they answered the open-ended questions. This difference concerned how they'd set up windows in a multiwindow display system while viewing code. Programmers working with the book listing all said they'd put the index in one window and use other windows for tracing and running the program. Programmers using the traditional listing described essentially the same window displays except the index window was absent.

7.4.6. Conclusions from Experiment 4.

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

Programmers using the book listing all agreed it was a better way of listing code than they had seen before. All programmers agreed that implementing different parts of a listing via a multiwindow display was desirable. Although programmers disagreed as to what should be placed in the windows, it's worth noting that all programmers working with the book listing used the table of contents and the index in every part of the 4 part experimental task.

7.5. Conclusions from Experiments 1 through 4.

These four experiments -- one macro-typographic, two micro-typographic, and the empirical study with real programmers -- show that the book paradigm is a natural form for formatting source code that is better than traditional methods. Specifically, we have shown that: (1) computer science students perform maintenance tasks better when using our book format of macro-typographic style, (2) computer science students understand C and Pascal code segments better when written according to our book format for micro-typographic style, and (3) professional programmers work with book format listings quicker and more accurately than with traditional listings. We emphasize that these improvements come without any introduction or instructions on how to use the book listings.

Chapter 8

Conclusions

8.1. Review of results.

We have presented a different classification and analysis of the factors comprising programming style. Results from existing style research are inconsistent, indefinite, and contradictory. By reviewing the existing problems in programming style research and identifying the goals of that research, we were able to identify an appropriate research paradigm. Our paradigm led to the development of a style taxonomy which enabled us to identify areas of inconsistency, unsubstantiated rules, and misconceptions about programming style. By categorizing and isolating classes of style factors -- typographic, control structure, and information structure -- we were better able to assess the influence of specific style factors and determine their utility. The identification of macro and micro concerns within each category further refined the taxonomic breakdown and provided a means to identify principles and establish implementation techniques. Our paradigm for style research, combined with our style taxonomy, provides a very powerful and useful research vehicle for programming style analysis.

Using this research vehicle, we then concentrated on the analysis of macro- and micro-typographic style. Typographic style factors have no impact on code efficiency or information content and flow through the program; but they do affect how programmers view and work with code. Our preliminary experiments demonstrated the value of typographic style analysis and showed that typographic characteristics impact programmer comprehension. Specifically, we implemented a typographic style analyzer useful in consistency checking and authorship analysis; and we demonstrated the relationship between program comprehension and the spatial arrangement of nested IF structures. In doing so, we also demonstrated the inadequacy of existing code formatters and code formatting techniques.

After analyzing theories of programming comprehension and maintenance activity -- in relation to our analysis of typographic style -- we formulated several principles of typographic style that are consistent and compatible with those theories. The primary objectives of good typographic style are to reflect the underlying structure of the code (both control and information structures) by providing visual clues, and to provide programmers with many ways to view the code. Once these principles were identified, implementation techniques could be devised. Our book model of source code formatting is one mechanism for implementing these objectives. The book paradigm provides both macro- and micro-typographic clues that assist programmers during comprehension. Furthermore, it provides programmers with multiple avenues or access paths to get "into" the code.

We then defined and tested the book paradigm for typographic style. Our controlled experiments and empirical studies with the book model for formatting source code 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, (3) professional programmers can benefit from the book model, and (4) the book format listing is a "natural" implementation of typographic style.

8.2. Future work.

Our work has several implications on future style research. Results from programming style studies should be re-examined with a critical eye towards methodology and theoretic basis. Over-generalized conclusions have led to inappropriate applications which, in turn, have led to published criticisms [Kear86, Shen83]. Folklore and published style guidelines should be re- examined in light of our analyses on the affects of typographic style on comprehension.

Principles of good programming style can be identified. However, style rules and guidelines can only be developed after studying the affects of specific style factors on comprehension. The affects of control flow and information flow style remain to be studied. The combination of typographic and structural rearrangement (e.g., the way Kernighan and Plauger arranged their third nested IF) should prove to be a powerful mechanism with which to improve code maintainability.

Our research has several implications on programming 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 structural clues.

2. A "book maker" program could be implemented to turn existing code listings into book formatted listings. This type of reverse-engineering could be used to assist in the maintenance of existing software.

3. Language directed editors could be designed to incorporate "intelligent" code reformatting 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. Further, the implementation could maintain a logical versus physical separation of the code display and source file. That is, some characteristics may actually be incorporated into the source file (e.g., spacing), while others might be generated and displayed on demand only (e.g., indices).

4. A multiwindow hypertext-like maintenance tool could be designed to allow programmers to have simultaneous views into the code being studied. This line of research is being pursued by several groups (see [Marc88, Shne86]), but all are accessing and displaying information outside the source code listing and, thereby, are creating 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. Frequently, the only reliable documentation on a system is the source code. Hence, a book format hypertext maintenance tool could display multiple views and permit a variety of access paths without needing additional (external) knowledge structures.

5. A programming style analyzer that takes into account several classes of stylistic characteristics could be a powerful tool for consistency checking, standards enforcement, maintainability assessment, authorship analysis, and plagiarism

detection. A tool that automatically checks for typographic style consistency would aid the maintainer and could also check for adherence to style standards.

And finally, our research has implications on language design and development. As Gannon and Horning suggest in their paper on language design and reliability, "A language should contain constructs that encourage the clearest possible statement of the programmer's intentions" [Gann75]. That is, language constructs should be compatible with the way programmers view code both physically and mentally. The tabular spatial arrangement in our nested IF experiments is an example. The series of nested, indented IF's obscures the simple 1:1 relationship between condition and action. The structure becomes more apparent when expressed in CASE-like form, but most CASE structures are restricted to one value and type, so programmers are forced to used nested IFs. A more flexible CASE-like language construct, one that makes the 1:1 relationship obvious without superfluous tokens, would be clearer and easier to read. In general, language constructs and tokens that interfere with comprehension need to be eliminated and replaced with structures that facilitate comprehension and maintenance.

Bibliography

[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.

[Alsy86] Alsys, Inc., *Alsys PC AT Ada Compiler User's Guide*, Alsys Inc., Waltham MA, 1986.

[Arth85] L. J. Arthur, *Measuring Programmer Productivity and Software Quality*, John Wiley & Sons, New York NY, 1985.

[Bake79] A. Baker & S. Zweben, "The Use of Software Science in Evaluating Modularity Concepts," *IEEE Transactions on Software Engineering*, SE-5(2), Mar. 1979, pp. 110-120.

[Basi84] V. Basili & B. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, vol. 27(1), Jan. 1984, pp. 42-52.

[Bast87] F. Bastani & S. Iyengar, "The Effects of Data Structures on the Logical Complexity of Programs," *Communications of the ACM*, vol. 30(3), Mar. 1987, pp. 250-259.

[Bate81] R. M. Bates, "A Pascal Prettyprinter with a Different Purpose," *ACM SIGPLAN Notices*, vol. 16(3), Mar. 1981, pp. 10-17.

[Bend87] S. Bendifallah & W. Scacchi, "Understanding Software Maintenance Work," *IEEE Transactions on Software Engineering*, SE- 13(3), Mar. 1987, pp. 311-323.

[Ben86a] J. Bentley & D. Knuth, "Literate Programming," *Communications of the ACM*, vol. 29(5), May 1986, pp. 364-369.

[Ben86a] J. Bentley, D. Knuth & D. McIlroy, "A Literate Program," *Communications of the ACM*, vol. 29(6), June 1986, pp. 471-483.

[Berg84] H. L. Berghel & D. L. Sallach, "Measurements of Program Similarity in Identical Task Environments," ACM SIGPLAN Notices, vol. 19(8), Aug. 1984, pp. 65-76.

[Berg85] J. Bergeretti & B. Carre, "Information-Flow and Data- Flow Analysis of While-Programs," *ACM Transactions on Programming Languages and Systems*, vol. 7(1), Jan. 1985, pp. 37-61.

[Bern84] G. Berns, "Assessing Software Maintainability," *Communications of the ACM*, vol. 27(1), Jan. 1984, pp. 14-23.

[Berr85] R. E. Berry & B. A. E. Meekings, "A Style Analysis of C Programs," *Communications of the ACM*, vol. 28(1), Jan. 1985, pp. 80-88.

[Birk59] N. P. Birk & G. B. Birk, *Understanding and Using English*, Odyssey Press, New York NY, 1959.

[Broo80] R. E. Brooks, "Studying Programmer Behavior Experimentally: The Problems of Proper Methodology," *Communications of the ACM*, vol. 23(4), Apr. 1980, pp. 207-213.

[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.

[Card85] D. Card, G. Page, F. McGarry, "Criteria for Software Modularization," *Proceedings of the 8th International Conference on Software Engineering*, (London, England, Aug. 28-30, 1985), IEEE Computer Society Press, Washington D.C., 1985, pp. 372-377.

[Chas73] W. Chase & H. Simon, "Perception in Chess," *Cognitive Psychology*, vol. 4, 1973, pp. 55-81.

[ChiF81] M. Chi, P. Feltovich, & R. Glaser, "Categorization and Representation of Physics Problems by Experts and Novices," *Cognitive Science*, vol. 5, 1981, pp. 121-152.

[Cook84] C. Cook, W. Bregar, & D. Foote, "A Preliminary Investigation of the Use of the Cloze Procedure as a Measure of Program Understanding," *Information Processing and Management*, vol. 20(1), 1984, pp.199-208.

[Coop85] D. Cooper & M. Clancy, *Oh! Pascal!*, W. W. Norton & Company, New York NY, 1985.

[Crid78] J. E. Crider, "Structured Formatting of Pascal Programs," *ACM SIGPLAN Notices*, vol. 13(11), Nov. 1978, pp. 15- 22.

[Davi83] J. S. Davis, "Investigation of Chunks in Complexity Measurement," *Computer Science and Statistics: The Interface*, (J. E. Gentle, editor), North-Holland Publishing, 1983.

[Davi84] J. S. Davis, "Chunks: A Basis for Complexity Measurement," *Information Processing & Management*, vol. 20(1), 1984, pp. 119-127.

[DeGr65] A. De Groot, *Thought and Choice in Chess*, The Hague: Mouton, 1965.

[Dixo81] W. J. Dixon, *BMDP Statistical Software 1981*, University of California Press, Berkeley CA, 1981.

[Doer85] C. Doerflinger & V. Basili, "Monitoring Software Development Through Dynamic Variables," *IEEE Transactions on Software Engineering*, SE-11(9), Sept. 1985, pp. 978-985.

[Duns85] H. E. Dunsmore, "The Effects of Comments, Mnemonic Names, and Modularity: Some University Experiment Results," *Proceedings of the Second Symposium on Empirical Foundations of Information and Software Science*, 1985.

[Evan84] M. Evangelist, "Program Complexity and Programming Style," *Proceedings of the International Conference on Data Engineering* (Los Angeles CA, Apr. 24-27), IEEE CS Press, Silver Springs MD, 1984, pp. 534-541.

[Fair85] R. E. Fairley, *Software Engineering Concepts*, McGraw- Hill, New York NY, 1985.

[Gann75] J. D. Gannon & J. J. Horning, "Language Design for Programming Reliability," *IEEE Transactions on Software Engineering*, SE-1(2), June 1975, pp. 179-191.

[Gilm84] D. J. Gilmore & T. R. Green, "Comprehension and Recall of Miniature Programs," *International Journal of Man-Machine Studies*, vol. 21, 1984, pp. 31-48.

[Goul74] J. Gould & P. Drongonowski, "An Exploratory Study of Computer Program Debugging," *Human Factors*, vol. 16, 1974, pp. 258-277.

[Grem84] L. Gremillion, "Determinants of Program Repair Maintenance Requirements," *Communications of the ACM*, vol. 27(8), Aug. 1984, pp. 826-832.

[Grog79] P. Grogono, "On Layout, Identifiers and Semicolons in Pascal Programs," *ACM SIGPLAN Notices*, vol. 14(4), Apr. 1979, pp. 35-40.

[Hans85] J. Hansen & B. Sands, "Some Design Considerations for a C Source Code Pretty Printer," *ACM SIGSMALL/PC Notes*, vol. 11(2), May 1985, pp. 16-22.

[Harr86] W. Harrison & C. Cook, "A Note on the Berry-Meekings Style Metric," *Communications of the ACM*, vol. 29(2), Feb. 1986, pp. 123-133.

[Hutc85] D. Hutchens & V. Basili, "System Structure Analysis: Clustering with Data Bindings," *IEEE Transactions on Software Engineering*, SE-11(8), Aug. 1985, pp. 749-757.

[Kear86] J. Kearney, R. Sedlmeyer, W. Thompson, M. Gray, & M. Adler, "Software Complexity Measurement," *Communications of the ACM*, vol. 29(11), Nov. 1986, pp. 1044-1050.

[Kern74] B. W. Kernighan & P. J. Plauger, *The Elements of Programming Style*, McGraw-Hill, New York NY, 1974.

[Kern78] B. W. Kernighan & D. Ritchie, *The C Programming Language*, Prentice-Hall Publishing, Englewood Cliffs NJ, 1978.

[Knut84] D. Knuth, "Literate Programming," *Computer Journal*, vol. 27(2), May 1984, pp. 97-111.

[Ledg75] H. F. Ledgard, *Programming Proverbs*, Hayden Books, Rochelle Park NJ, 1975.

[Ledg78] H. F. Ledgard & L. J. Chmura, *FORTRAN With Style: Programming Proverbs*, Hayden Books, Rochelle Park NJ, 1978.

[Ledg87] H. F. Ledgard & J. Tauer, *Professional Software Vol. II: Programming Practice*, Addison-Wesley, Reading MA, 1986.

[Leto86] S. Letovsky, "Cognitive Processes in Program Comprehension," *Empirical Studies of Programmers*, (E. Soloway & S. Iyengar, editors), Ablex Publishing, Norwood NJ, 1986, pp. 58- 79.

[Litt86] D. Littman, J. Pinto, S. Letovsky & E. Soloway, "Mental Models and Software Maintenance," *Empirical Studies of Programmers*, (E. Soloway & S. Iyengar, editors), Ablex Publishing, Norwood NJ, 1986, pp. 80-98.

[Logi87] Logitech, Inc., *Logitech Modula 2/86 Point Editor*, Logitech Inc., Redwood City CA, 1987.

[Marc81] D. Marca, "Some Pascal Style Guidelines," *ACM SIGPLAN Notices*, vol. 16(4), Apr. 1981, pp. 70-80.

[Marc88] G. Marchionini & B. Shneiderman, "Finding Facts vs. Browsing Knowledge in Hypertext Systems," *IEEE Computer*, vol. 21(1), Jan. 1988, pp. 70-80.

[Mark84] C. Markel, *Technical Writing Style: Situations and Strategies*, Martin's Press, New York NY, 1984.

[Mart83] J. Martin & C. McClure, *Software Maintenance: The Problem and Its Solutions*, Prentice-Hall, Englewood Cliffs NJ, 1983.

[Maye82] R. Mayer, "Different Problem Solving Strategies for Algebra Word and Equation Problems," *Journal of Experimental Psychology*, vol. 8(5), 1982, pp. 448-462.

[McKe81] K. McKeithen, J. Reitman, H. Rueter, & S. Hirtle, "Knowledge Organization and Skill Differences in Computer Programmers," *Cognitive Psychology*, vol. 13, 1981, pp. 307-325.

[Meek83] B. Meekings, "Style Analysis of Pascal Programs," *ACM SIGPLAN Notices*, vol. 18(9), Sept. 1983, pp. 45-54.

[Merr74] G. & C. Merriam Co., *The Merriam-Webster Dictionary*, Pocket Books, New York NY, 1974.

[Miar83] R. Miara, J. Musselman, J. Navarro, & B. Shneiderman, "Program Indentation and Comprehensibility," *Communications of the ACM*, vol. 26(11), Nov. 1983, pp. 861-867.

[Mohe81] T. Moher & M. Schneider, "Methods for Improving Controlled Experimentation in Software Engineering," *Proceedings of the 5th International Conference on Software Engineering*, IEEE Computer Society Press, Los Angeles CA, 1981, pp. 224-233.

[Nanj88] M. Nanja, *An Investigation of the On-line Debugging Process of Expert and Novice Student Programmers*, Ph.D. Dissertation, Oregon State University Computer Science Dept., Jan. 1988.

[Norc78] A. F. Norcio, "Memory Organization and Computer Program Logic," presented at the annual meeting of the Human Factors Society, Detroit MI, 1978.

[Norc82] A. F. Norcio, "Human Memory Process for Comprehending Computer Programs," Applied Sciences Dept. note # AS-1-82, U.S. Naval Academy, Annapolis MD, 1982.

[Noru85] M. Norusis, *SPSS-X: Advanced Statistics Guide*, McGraw- Hill, New York NY, 1985.

[Olso87] G. Olson, S. Shepard, & E. Soloway (editors), *Empirical Studies of Programmers: Second Workshop*, Ablex Publishing, Norwood NJ, 1987.

[Oman86] P. W. Oman & D. Willson, "Paradigm for K-8 Computer Curriculum Design," *T.H.E. Journal*, vol. 14(2), Sept. 1986, pp. 82-88.

[Oman89] P. Oman, C. Cook, & M. Nanja, "Effects of Programming Experience in Debugging Semantic Errors", to appear in *The Journal of Systems and Software*, Jan. 1989.

[OttL77] L. Ott, *An Introduction to Statistical Methods of Data Analysis*, Duxbury Press, North Scituate MA, 1977.

[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.

[Penn87] N. Pennington, "Comprehension Strategies in Programming," *Empirical Studies of Programmers: Second Workshop*, (G. Olson, S. Shepard, & E. Soloway, editors), Ablex Publishing, Norwood NJ, 1987, pp. 100-113.

[Pete77] J. L. Peterson, "On the Formatting of Pascal Programs," *ACM SIGPLAN Notices*, vol. 12(12).

[Pres82] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York NY, 1982.

[Redi86] K. A. Redish & W. F. Smyth, "Program Style Analysis: A Natural By-Product of Program Compilation," *Communications of the ACM*, vol. 29(2), Feb. 1986, pp. 126-133.

[Rees82] M. J. Rees, "Automatic Assessment Aids for Pascal Programs," *ACM SIGPLAN Notices*, vol. 17(10), Oct. 1982, pp. 33- 42.

[Romb87] H. D. Rombach, "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Transactions on Software Engineering*, SE-13(3), Mar. 1987, pp. 344-354.

[Rose83] D. Rosenthal, in correspondence from the members, *ACM SIGPLAN Notices*, vol. 18(3), Mar. 1983, pp. 4-5.

[Schn81] G. Schneider & S. Buell, *Advanced Programming and Problem Solving With Pascal*, John Wiley & Sons, New York NY, 1981.

[Scho82] A. Schoenfeld & D. Herrmann, "Problem Perception and Knowledge Structure in Expert and Novice Mathematical Problem Solvers," *Journal of Experimental Psychology*, vol. 8(5), 1982, pp. 484-494.

[Shei81] B. A. Sheil, "The Psychological Study of Programming," *Computing Surveys*, vol. 13(1), Mar. 1981, pp. 101-120.

[Shen83] V. Y. Shen, S. D. Conte, & H. E. Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support," *IEEE Transactions on Software Engineering*, SE-9(2), Mar. 1983, pp. 155-165.

[Shep79] S. Sheppard, B. Curtis, P Milleman, & T. Love, "Modern Coding Practices and Programmer Performance," *Computer*, vol. 12(12), Dec. 1979, pp. 41-49.

[Shne86] B. Shneiderman, P. Shafer, R. Simon, & L. Weldon, "Display Strategies for Program Browsing: Concepts and Experiment," *IEEE Software*, vol. 3(3), May 1986, pp. 7-15.

[Solo82] E. Soloway, K. Ehrlich, & J. Bonar, "Tapping Into Tacit Programming Knowledge," *ACM SIGCHI 1982 Conference Proceedings*, 1982, pp. 52-57.

[Solo84] E. Soloway, & K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering*, SE-10(5), Sept. 1984, pp. 595-609.

[Solo86] E. Soloway & S. Iyengar (editors), *Empirical Studies of Programmers*, Ablex Publishing, Norwood NJ, 1986.

[Stru59] W. Strunk & E. B. White, *The Elements of Style*, MacMillan, New York NY, 1959.

[Tass78] D. V. Tassel, *Program Style, Design, Efficiency, Debugging, and Testing*, Prentice-Hall, Englewood Cliffs NJ, 1978.

[Thin84] Think Technologies, Inc., *Macintosh Pascal Reference Manual* (ver. 1), Think Technologies, Inc., Lexington MA, 1984.

[Thin86] Think Technologies, Inc., *Lightspeed Pascal: User's Guide and Reference Manual* (ver. 1), Think Technologies, Inc., Lexington MA, 1986.

[Vess83] I. Vessey & R. Weber, "Some Factors Affecting Program Repair Maintenance: An Empirical Study," *Communications of the ACM*, vol. 26(2), Feb. 1983, pp. 128-134.

[Vess84] I. Vessey & R. Weber, "Research on Structured Programming: An Empiricist's Evaluation," *IEEE Transactions on Software Engineering*, SE-10(4), Jul. 1984, pp. 397-407.

[Vess85] I. Vessey, "Expertise in Debugging Computer Programs: A Process Analysis," *International Journal of Man-Machine Studies*, vol. 23, 1985, pp. 459-494.

[Wei74a] L. Weissman, *A Methodology for Studying the Psychological Complexity of Computer Programs*, Ph.D. thesis, University of Toronto, 1974.

[Wei74b] L. Weissman, "Psychological Complexity of Computer Programs: An Experimental Methodology," *ACM SIGPLAN Notices*, vol. 9, Jun. 1974, pp. 136-138.

[Wied86] S. Wiedenbeck, "Processes in Program Comprehension," *Empirical Studies of Programmers*, (E. Soloway & S. Iyengar, editors), Ablex Publishing, Norwood NJ, 1986, pp. 48-57.

[Wint88] K. A. Winter, *A Prototype Intelligent Prettyprinter*, Oregon State University Computer Science Dept., Master's thesis, 1988.

[Wood81] S. Woodfield, H. Dunsmore, & V. Shen, "The Effect of Modularization and Comments on Program Comprehension," *Proceedings of the Fifth International Conference on Software Engineering* (San Diego CA, Mar. 9-12), IEEE Computer Science Press, Los Alamitos CA, 1981, pp. 215-223.

[Your75] E. Yourdon, *Techniques of Program Structure and Design*, Prentice-Hall, Englewood Cliffs NJ, 1975.

Appendices

Appendix A.

Composite Listing of Style Rules

0. General Programming Practices.

A. Design techniques & considerations:

- Define the problem completely [Ledg75].
- Think first, program later [Ledg75].
- Resist the urge to start coding right away [Schn81].
- Consider how to achieve generality from the earliest stages of program design and specification [Schn81].
- Simplify arithmetic and logical expressions before committing to code [Pres82].
- Write first in an easy-to-program pseudo-language; then translate into whatever language you have to use [Kern74].
- Include comments from the very beginning of the coding phase [Schn81].
- Get the syntax correct now, not later [Ledg75].
- Think: "Could I understand this if I was not the person who coded it?" [Pres82].
- Don't be afraid to start over [Ledg75].

B. Language selection and restrictions:

- Reread the (language) manual [Ledg75].
- Consider another language [Ledg75].
- Use a standard version of a language [Schn81].
- Use only ANSI standard features [Pres82].
- Avoid implementation-dependent features [Ledg75].

C. Testing and debugging practices:

- Build in debugging techniques [Ledg75].
- Use debugging compilers [Kern74].
- Test programs at their boundary values [Kern74].
- Watch out for off-by-one errors [Kern74].
- Don't stop at one bug [Kern74].
- Check some answers by hand [Kern74].
- Hand-check the program before running it [Ledg75].
- Prevent run-time errors by checking for them before any risky operation [Schn81].

D. Maintenance concerns:

- Don't patch bad code -- rewrite it [Kern74].
- Don't comment bad code -- rewrite it [Kern74].
- Be prepare to rewrite code that deserves a comment [Ledg87].
- Don't use long, involved comments to document obscure, complex code; rewrite the code [Fair85].
- Always be sure that comments and code agree with on another, and with the requirements and design specifications [Fair85].
- Do not let comments get out of date with the code [Schn81].

Appendix A: Composite Listing of Style Rules (continued)

1. Typographic Style.

1.1. Macro-typographic style.

A. Overall program formatting:

- o Format a program to help the reader understand it [Kern74].
- o Prettyprint [Ledg75].
- o Program lines should fit on 8.5x11 paper [Ledg87].
- o Use comments [Ledg75].

B. Global and intermodule commenting:

- o Provide good documentation [Ledg75].
- o Document your data layout [Kern74].
- o Provide standard documentation prologues for each subprogram and/or compilation unit [Fair85].
- o Proloque comments should appear at the beginning of every module; to include <list> [Pres82].
- o Use prologue comments [Schn81].
- o Use header and marker comments [Ledg87].
- o Minimize the need for embedded comments by using standard proloques, structured programming constructs, good coding style, descriptive names, and self-documenting features of the language [Fair85].
- o Use problem domain terminology in the comments [Fair85].
- o Be correct; an incorrect or misleading comment is worse than no comment at all [Pres82].

C. Module separation:

- o Separate modules with page breaks or at least 3 lines [Ledg87].

D. Identifier and label naming:

- o Pick good mnemonic names for all procedures and functions [Schn81].
- o Use nouns (or noun phrases) for variables, verbs (or verb phrases) for procedures [Ledg87].
- o Use names that accurately reflect their semantic roles [Ledg75].
- o Always ask what a name suggests, not whether the concept suggests the name [Ledg87].
- o Type (name) should be what it stands for; not just a particular instantiation [Ledg87].
- o The algorithm (vs the declaration) should read well [Ledg87].
- o Meaning (of names) is the game! [Ledg87].

E. Special symbols, case, fonts and type styles:

- o Use standard notational conventions where applicable [Ledg78].

1.2. Micro-typographic style.

A. Statement formatting:

- o Do not place more than one control statement per line [Schn81].
- o Every statment on a separate line [Ledg87].
- o Compound logical expressions spanning more than one line should have each condition on new line [Ledg87].
- o CONST, TYPE, and VAR sections should begin with those reserved words on separate lines [Ledg87].

Appendix A: Composite Listing of Style Rules (continued)

B. Vertical spacing:

- Separate program constructs by at least 1 line [Ledg87].
- Paragraph a program with either comments or blank lines [Schn81].

C. Horizontal spacing (indentation & alignment):

- Use indentation, parentheses, blank spaces, blank lines and borders around comment blocks to enhance readability [Fair85].
- Use spacing and/or readability symbols to clarify statement content [Pres82].
- Ident to highlight nesting depth of a group of control statements [Schn81].
- Indent multiline statements so that structurally related clauses are aligned [Schn81].
- If statements should be indented as in his (3) examples [Ledg87].
- Declarations should be "comb" structures [Ledg87].
- While and For loops with compound bodies should be in "comb" form [Ledg87].
- Relational operations and := should have spaces on either side [Ledg87].
- Put a space after every comma [Ledg87].

D. Intramodule commenting (content & form):

- Attach comments to blocks of code that perform major data manipulation, simulate structured control constructs using GOTO statements, and perform exception handling [Fair85].
- Comment each line in a VAR and CONST declaration [Schn81].
- Use comments to help match begin-end pairs [Schn81].
- Comment any statement whose intent is not immediately obvious [Schn81].
- Do not simply reiterate the obvious intent of a statement [Schn81].
- Don't just echo the code with comments -- make every comment count [Kern74].
- Do not write cryptic comments [Schn81].
- Avoid in-line comments [Ledg87].
- Describe blocks of code, rather than commenting every line [Pres82].
- Comment for content, not for dazzle [Ledg87].
- Use a style (of commenting) that does not interfere with program reading [Ledg87].
- Use blank lines, borders, and indentation to highlight comments [Fair85].
- Use blank lines or indentation so that comments can be readily distinguished from code [Pres82].
- Place comments to the far right to document changes and revisions [Fair85].
- When in doubt, leave the comment out [Ledg87].
- Try to make the code say it all [Ledg87].

E. Identifier and label naming:

- Pick good mnemonic names for all variables [Schn81].
- Use variable names that mean something [Kern74].
- Variables should be named for what they stand for [Ledg87].
- Use standard prefixes and suffixes for all related variables [Schn81].
- Put accuracy (of names) first [Ledg87].
- Do not make up cryptic or unclear abbreviations for variables [Schn81].
- Choose variable names that won't be confused [Kern74].
- Use psychologically distant names for different entities [Ledg78].
- Do not limit the length of names solely to save key strokes [Schn81].
- Do not abbreviate names for unusual items and concepts [Ledg87].
- Head for brevity (in names), especially full single words [Ledg87].
- Use context to shorten names [Ledg87].
- Do not use "cute" names that do not have mnemonic value [Schn81].
- Assign names to scalar constants where it will help clarity and readability [Schn81].
- Name mysterious constants [Ledg87].

Appendix A: Composite Listing of Style Rules (continued)

- Variable name (not type) deserves priority - because variables occur more frequently than types [Ledg87].
- Use statement labels that mean something [Kern74].
- Try to make the code speak for itself, effortlessly [Ledg87].

F. Special symbols and case:

- Parenthesize to avoid ambiguity [Kern74].
- Use parentheses to clarify logical or arithmetic expressions [Pres82].

2. Control Structure Syle.

2.1. Macro-control structure style.

A. Method of modular decomposition:

- Use the top-down approach (of functional decomposition) [Ledg75].
- Construct the program in logical units [Ledg75].
- Use procedures [Ledg75].
- Modularize; use subroutines [Kern74].
- Write and test a big program in small pieces [Kern74].
- Never halt a procedure [Schn81].

B. Module nesting (definition & call):

- Carefully examine routines having fewer than 5 or more than 25 executable statements [Fair85].

C. Module coupling, reusability and encapsulation:

- Use library functions [Kern74].
- Before you begin to design and code a procedure, first see if it already exists in a program library on your computer [Schn81].
- Replace repetitive expressions by calls to a common function [Kern74].
- Isolate machine dependencies in a few routines [Fair85].
- Don't strain to re-use code; reorganize instead [Kern74].
- Do not use a function when you need a subroutine [Ledg78].
- Do not use a subroutine when you need a function [Ledg78].

2.2. Micro-control structure style.

A. Code complexity (simplicity, efficiency, clarity):

- Write clearly -- don't sacrifice clarity for "efficiency" [Kern74].
- Never sacrifice clarity of expression for cleverness of expression [Schn81].
- Never sacrifice clarity of expression for minor reductions in machine execution time [Schn81].
- Don't sacrifice clarity for small gains in "efficiency [Kern74]."
- Always strive for simplicity and clarity [Schn81].
- Don't be too clever [Fair85].
- Avoid tricks [Ledg75].
- Avoid confusing programming tricks [Schn81].
- Say what you mean, simply and directly [Kern74].
- Make it right before you make it faster [Kern74].

Appendix A: Composite Listing of Style Rules (continued)

- Make it fail-safe before you make it faster [Kern74].
- Make it clear before you make it faster [Kern74].
- Keep it simple to make it faster [Kern74].
- Don't diddle code to make it faster -- find a better algorithm [Kern74].
- Instrument your programs. Measure before making "efficiency" changes [Kern74].
- Let the machine do the dirty work [Kern74].
- Let your compiler do the simple optimizations [Kern74].
- Don't suboptimize [Fair85].
- Use "fast" arithmetic operations [Pres82].

B. Use of structured constructs (loops, conditionals, cases):

- Learn to use and master the 6 basic control structures in Pascal: begin-end, if-then-else, case-end, while-do, repeat-end, and for-do -- so that you can implement any program unit as a properly nested set of blocks [Schn81].
- Use a few standard, agreed-upon control constructs [Fair85].
- Don't use conditional branches as a substitute for a logical expression [Kern74].
- If a logical condition is hard to understand, try transforming it [Kern74].
- Avoid the use of complicated conditional tests [Pres82].
- Use IF...ELSE IF...ELSE IF...ELSE... to implement multi-way branches [Kern74].
- Avoid null THEN statements [Fair85].
- Avoid THEN-IF statements [Fair85].

C. Use of unconditional non-structured branching:

- Avoid the Fortran arithmetic IF [Kern74].
- Avoid unnecessary branches [Kern74].
- Use GOTOs only to implement a fundamental structure [Kern74].
- Avoid GOTOs completely if you can keep the program readable [Kern74].
- Do not use the GOTO to jump backwards [Schn81].
- Do not use the GOTO to implement one of a number of ways to exit a loop [Schn81].
- Do not use the GOTO too often [Schn81].
- Avoid unecessary GOTOs [Ledg75].
- Consider the GOTO to handle errors or other abnormalities in logic [Schn81].
- Consider using the GOTO to handle unusual circumstances such as long forward branches to the end of a program unit or multilevel forward branches out of a complex nested structure [Schn81].
- Use GOTOs in a disciplined way [Fair85].
- Be careful when a loop exits to the same place from side and bottom [Kern74].

D. Control flow and exception handling:

- Make sure your code "does nothing" gracefully [Kern74].
- Make sure special cases are truly special [Kern74].
- Check to see that you have handled all the null cases properly [Schn81].
- Take care to branch the right way on equality [Kern74].
- Use integer arithmetic and boolean expressions whenever possible [Pres82].
- Eliminate tests on "negative" conditions [Pres82].
- 10.0 times 0.1 is hardly ever 1.0 [Kern74].
- Don't compare floating point numbers solely for equality [Kern74].
- Never assume exact equality of real values [Schn81].

Appendix A: Composite Listing of Style Rules (continued)

E. Nesting and span of control structures:

- While, repeat, for if, case, and with statement nesting should not exceed 5 levels [Ledg87].
- Don't nest too deeply [Fair85].
- Avoid heavy nesting of loops or conditionals [Pres82].
- Carefully evaluate nested loops to determine if statements or expressions can be moved outside [Pres82].

3. Information Structure Style.

3.1. Macro-information structure style.

A. Global data structures:

- Choose a data representation that makes the program simple [Kern74].
- Introduce user-defined data types to model entities in the problem domain [Fair85].
- When possible, avoid the use of multidimensional arrays [Pres82].
- Use data arrays to avoid repetitive control sequences [Kern74].
- When possible, avoid the use of pointers and complex lists [Pres82].
- Use recursive procedures for recursively-defined data structures [Kern74].
- Hide data structures behind access functions [Fair85].
- Localize and identify unavoidable machine-dependent information [Schn81].
- Avoid machine dependent constants [Schn81].
- Avoid specific collating sequences [Schn81].

B. Input structures (processing & validation):

- Keep input format(s) simple [Pres82].
- Use uniform input formats [Kern74].
- Select formats on the basis of ease of preparation and legibility, not ease of programming [Schn81].
- Keep input format(s) uniform when a programming language has stringent formatting requirements [Pres82].
- Use free-form input when possible [Kern74].
- Always terminate data through eof or a signal card, never user-supplied count fields [Schn81].
- Use end-of-data indicators, rather than requiring a user to specify "number of items" [Pres82].
- Terminate input by end-of-file or marker, not by count [Kern74].
- Always identify the input you are requesting, either with a prompt or screen template [Schn81].
- Label interactive input requests, specifying available choices or bounding values [Pres82].
- Have users supply input in the form most natural for them, not the program [Schn81].
- Make input easy to prepare and output self-explanatory [Kern74].
- Avoid programs that trap users in an infinite loop demanding correctly structured input, without offering assistance in preparing it [Schn81].
- Use a "help" mode to aid users who are confused about how to prepare input [Schn81].
- Always validate input for legality [Schn81].
- Always validate input for plausibility [Schn81].
- Validate all input data [Pres82].
- Check the plausibility of important combinations of input items [Pres82].
- Make sure input doesn't violate the limits of the program [Kern74].
- Make input easy to proofread [Kern74].
- Use defaults to reduce the amount of input data required for generalized programs [Schn81].
- Use self-identifying input; allow defaults and echo both on output [Kern74].
- Use self-identifying input to modify defaults [Schn81].
- Identify bad input; recover if possible [Kern74].
- Never stop (program execution due to bad input) if something useful can be done [Schn81].

Appendix A: Composite Listing of Style Rules (continued)

- If an error is detected, try to capture enough information to identify the exact cause of the error [Schn81].
- Produce a meaningful error message directly related to the specific error. Whenever possible, keep going in order to get as much additional information and/or results as you can [Schn81].
- Your goal should be to write a program that is protected against all improper data [Schn81].

C. Intermodule communication:

- Protect input parameters using call-by-value [Schn81].
- Do not alter formal parameters [Ledg78].
- Do not alter global variables [Ledg78].
- Avoid global variables and procedures with side effects [Schn81].
- Avoid side effects [Ledg75].
- Avoid obscure side effects [Fair85].
- Use a function only for its returned value [Ledg78].
- Programs should not contain any global variables (except for globally defined variables needed to implement the concept of own) [Ledg87].
- Where appropriate use signal flags to return the status of a computation to the calling program [Schn81].
- Make key data values parameters to procedures, not local variables [Schn81].

D. Output mechanisms:

- Always echo print the input [Schn81].
- Get the program correct before trying to produce good output [Ledg75].
- When the program is correct, produce good output [Ledg75].
- Avoid producing output within a procedure (unless the sole purpose of the procedure is output) [Schn81].
- Label all output and design all reports [Pres82].
- Above all, spend sufficient time in producing "finished" output that is elegant, legible, and directly usable by intended end-users [Schn81].

3.2. Micro-information structure style.

A. Temporary structures (complexity & variety):

- Do not be afraid to use local variables [Ledg78].
- Avoid temporary variables [Kern74].

B. Initialization techniques:

- Make sure all variables are initialized before use [Kern74].
- Initialize constants with data statements of initial attributes; initialize variables with executable code [Kern74].
- Never assume the computer assumes anything (initializes variables) [Ledg75].

C. Useage restrictions and span of variables:

- Make all temporary variables local to the procedure where they are used [Schn81].
- Use intermediate variables properly [Ledg75].
- Don't use an identifier for multiple purposes [Fair85].
- Don't mix data types [Pres82].
- Make key data values variables, not constants [Schn81].
- Leave loop variables alone [Ledg75].
- Do not recompute constants within a loop [Ledg75].

Appendix B.

Nested IF Experiments Sample Test Packet

THIS TEST IS DESIGNED TO MEASURE YOUR PROGRAM COMPREHENSION.

IT WILL NOT EFFECT YOUR GRADE IN THIS, OR ANY OTHER, COURSE.

IT WILL NOT EFFECT YOUR STANDING IN COMPUTER SCIENCE IN ANY WAY.

---> DO NOT TURN THE PAGE UNTIL TOLD TO DO SO <---

The following page contains a listing of some Pascal code. After the listing are some questions we would like you to answer. Please answer the questions in order because questions 1 through 5 are timed. When you have completed the first five questions, question 6 asks you to write down the elapsed time from the clock at the front of the room. Once you have recorded your time DO NOT GO BACK TO QUESTIONS 1 THROUGH 5. Please be as accurate as possible with your time and then go on to question 7.

You will have no more than 5 minutes to complete the first 5 questions.

STOP when you have completed the test and turn it face down on your desk. Please remain seated and quiet until everyone has finished.

WHEN YOUR INSTRUCTOR SAYS "GO" TURN THE PAGE AND START THE TEST.

Appendix B. Nested IF Experiments Sample (continued)

```
****************************************
*                                      *
*   IF A > B                           *
*     THEN S := 1                      *
*     ELSE IF A = B                    *
*       THEN IF C > D                  *
*         THEN S := 2                  *
*         ELSE S := 3                  *
*       ELSE IF C > D                  *
*         THEN S := 4                  *
*         ELSE IF C = D                *
*           THEN S := 5                *
*           ELSE S := 6;               *
*                                      *
****************************************
```

Questions:

1. Given A = 6, B = 7, C = 3, and D = 4, what value would be assigned to S ? 6

2. Given A = 7, B = 10, C = 1, and D = 0, what value would be assigned to S ? 4

3. Given A = 10, B = 10, and C = D, what value would be assigned to S ?
3

4. Given B = 6, C = 6, and D = 2, what can you say about A if S is assigned the value 4 ? A<B

5. Under what conditions will S be assigned the value 6 ?
A<B and C<D

6. Please look at the clock and write down the elapsed time:

MINUTES:______________ SECONDS:________________

7. How would you rate this method of indentation for readability?
1......2......3......4......5
very poor o.k. good very
poor good

8. How would you rate this structure for clarity and readability?
1......2......3......4......5
very poor o.k. good very
poor good

STOP -- Turn the test face down, remain seated and quiet.

Appendix C.

Experiment 1: Macro-typographic Experiment Instructions

NAME ________________________

THIS TEST IS DESIGNED TO MEASURE YOUR PROGRAMMING EXPERIENCE.

IT WILL NOT REDUCE YOUR GRADE IN THIS COURSE.

---> DO NOT TURN THE PAGE UNTIL TOLD TO DO SO <---

Try to do your best in the time available -- but don't let this test bother you -- it won't lower your grade it can only increase your grade. Everyone working on this test will receive a 20% bonus coupon that can be used to increase one of your programming assignments by 20% of your earned points.

The following is a listing of a text editor written in Pascal. It is a real, working, text editor, except that I've taken out some of the code. The editor should permit free-form command entry, but I've removed the code that skips blanks in the editor command input lines. Specifically, I took out all the calls (there were 4 calls) and definition of this procedure:

PROCEDURE skip_blanks(VAR line : line_def);

Your job is to rewrite the skip_blanks procedure (it's only 7 lines long) and re-insert it's calls in the right places. To do this, you must understand the program, it's data structures, and it's execution order. (For simplicity, I've also removed a lot of other code that is unrelated to this problem.)

When you have completed the problem (or given up) please write down the time from the clock on the wall. This is just to compare the quality of your solution with the time it took you to do it. Your time will have no affect on your score.

Please be as accurate as possible with your time and then turn in your solution (turn in everything) and leave quietly. I will score the exercise over the weekend and turn everything back to you.

WHEN YOUR INSTRUCTOR SAYS "GO", PLEASE START THE TEST.

When finished, please look at the clock and write down the TIME:___________

Appendix D.

Experiment 2: Pascal Micro-typographic Experiment Sample Test Packet

THIS EXPERIMENT IS DESIGNED TO STUDY PROGRAMMING STYLE.

IT WILL NOT EFFECT YOUR GRADE IN THIS, OR ANY OTHER, COURSE.

IT WILL NOT EFFECT YOUR STANDING IN COMPUTER SCIENCE IN ANY WAY.

---> DO NOT TURN THE PAGE UNTIL TOLD TO DO SO <---

The following two pages contain listings of two Pascal procedures taken from a commercially available toolbox. On the last page there are some questions pertaining to the Pascal routines. Please try to answer as many as you can, with reasonable accuracy and speed. Questions 1 through 10 are timed.

When you have completed the first 10 questions, there is a place for you to write down the elapsed time from the clock at the front of the room. Once you have recorded your time DO NOT GO BACK TO QUESTIONS 1 THROUGH 10. Please be as accurate as possible with your time and then go on to the last question.

You will have no more than 10 minutes to complete the timed questions.

STOP when you have completed all the questions and turn the test materials face down on your desk. Please remain seated and quiet until everyone has finished.

Turn in only the last page (you may tear it off while working on it). You may keep the Pascal code listings.

WHEN YOUR INSTRUCTOR SAYS "GO", PLEASE START THE TEST.

Appendix D. Pascal Micro-typographic Experiment Sample (continued)

```
PROCEDURE Find;

{ Find is used to find, edit and delete customer records          }

{ Global constants --  CtrlZ, Tab, Enter                          }
{ Global variables --  NameIndexFile, DataFile                    }

{ User defined calls --  InputStr, KeyFromName, SearchKey,
                  GetRec, Ask, Process_Edit,
                  Process_Delete                                  }

VAR

  L, Index, Key        : Integer;
  EndChar, Answer  : Char;
  FirstNm, LastNm  : AnyStr;
  Customer             : CustRec;

BEGIN

  FirstNm := '';
  LastNm  := '';
  L := 1;

  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));

  Key := KeyFromName( LastNm, FirstNm );
  SearchKey( NameIndexFile, Key, Index );
  GetRec( DataFile, Index, Customer );

  Ask( ' E)dit, D)elete, Q)uit ', ['E','D','Q'], Answer );
  Case Answer of
    'E' : Process_Edit    ( Customer, Index );
    'D' : Process_Delete( Customer, Index );
  end;

END {Find};
```

Appendix D. Pascal Micro-typographic Experiment Sample (continued)

```
PROCEDURE InputStr( Var InStr          : AnyStr;
                        MaxLen, X, Y   : Integer;
                        EndingChars    : SetofChars;
                    Var EndChar        : Char );

{ InputStr permits entry and inline editing of character strings          }
{    Global constants -- Backspace, Leftarrow, Ritearrow, Lefttab, Ritetab }
{    Library routines  -- GotoXY, Length                                   }
{    User defined calls -- Insert, Delete, Beep                            }

VAR
    Col    : Integer;
    InChar : Char;

BEGIN

   Col := 0;

   Repeat

      GotoXY( X+Col, Y );
      Read( Kbd, InChar );

      Case InChar of

         'A'..'z'    : If Col < MaxLen then begin
                          Col := Col + 1;
                          Insert( InChar, InStr, Col);
                          end
                        else Beep;

         Backspace : If Col > 0 then begin
                          Delete( InStr, Col, 1 );
                          Col := Col - 1;
                          end
                       else Beep;

         Leftarrow  : If Col > 0              then Col:=Col-1  else Beep;
         Ritearrow  : If Col < Length(InStr)  then Col:=Col+1  else Beep;

         Lefttab    : Col := 0;
         Ritetab    : Col := Length( InStr );

      else if not (InChar in EndingChars) then Beep;
      end {of case};

   until InChar in EndingChars;

   EndChar := InChar;
END {InputStr};
```

Appendix D. Pascal Micro-typographic Experiment Sample (continued)

1. When Find calls InputStr, how many characters are allowed in first
   names? 15 In last names? 30

2. When does Find stop calling InputStr? (list conditions)
   (Endchar = Ctrl Z) or ((Endchar = Enter) and (L = 1))

3. Customer records are stored in a (a) sequential file or [✓ (b)] indexed file

4. What two procedures does InputStr call to build InStr?
   Insert and Delete

5. When does InputStr stop reading characters? (list conditions)
   When Inchar is in Endingchar

6. What happens in InputStr if a left-arrow key is struck before
   entering any other characters? Beep

7. The variable Col in InputStr is used to
   [✓ a)] keep track of the character position
   b) count the number of characters input
   c) count the number of editing operations
   d) compute the maximum string length

8. The variable L in Find is used to
   a) count the number of characters in names
   b) count the number of error returns
   [✓ c)] switch between firstname and lastname inputs
   d) compute the search key

9. If Ask works like InputStr, then it should repeat until
   a) L = 1
   [✓ b)] Answer in ['E','D','Q']
   c) EndChar in [CtrlZ,Tab,Enter]
   d) InChar = CtrlZ

10. Customer records are actually changed in which two procedures?
    Process-Edit and Process-Delete

— — — — — — — — — —

11. Please look at the clock and write down the elapsed time:

    MINUTES:____________ SECONDS:____________

12. How would you rate this code for clarity and readability?
    1......2......3......4......5
    very poor o.k. good very
    poor good

STOP -- Turn the test face down, remain seated and quiet.

Appendix E.

Experiment 3: C Micro-typographic Experiment Sample Test Packet

THIS EXPERIMENT IS DESIGNED TO STUDY PROGRAMMING STYLE.

IT WILL NOT REDUCE YOUR GRADE IN THIS COURSE.

---> DO NOT TURN THE PAGE UNTIL TOLD TO DO SO <---

Your participation in this experiment is entirely voluntary -- it won't lower your grade it can only increase your grade. Everyone working on this test will receive a 5% bonus coupon that can be used to increase one of your exam grades by 5% of your earned points.

The following two pages contain a listing of a C program taken out of Kernighan and Ritchie's book "The C Programming Language." Following the program listing there are some questions pertaining to the C program. Please try to answer as many as you can, with reasonable accuracy and speed. Questions 1 through 9 are timed.

When you have completed the first 9 questions, there is a place for you to write down the time from the clock at the front of the room. Once you have recorded your time DO NOT GO BACK TO QUESTIONS 1 THROUGH 9. Please be as accurate as possible with your time and then go on to the last question.

You will have only 10 minutes to complete the timed questions.

STOP when you have completed all the questions and turn the test materials face down on your desk. Please remain seated and quiet until everyone has finished.

Turn in only the last page (you may tear it off while working on it). You may keep the C code listings. Be sure your name is on the test page.

WHEN YOUR INSTRUCTOR SAYS "GO", PLEASE START THE TEST.

Appendix E. C Micro-typographic Experiment Sample (continued)

```
/* ---------------------------------------------------------- */

#define MaxOp    20  /* max size of operand, operator */
#define Number   '0' /* signal that number found */
#define TooBig   '9' /* signal that string is too big */

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


main()   /* reverse Polish desk calculator */

{ int   Type;
  char  S[MaxOp];
  double Op2, AtoF(), Pop(), Push();

  while ( (Type=GetOp(S,MaxOp)) != EOF )

     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;

       case '=':    printf( "\t%f\n", Push(Pop()) );              break;

       case 'c':    Clear();                                      break;

       case TooBig: printf( "%.20s ... is too long\n", S );       break;

       default:     printf( "unknown command %c\n", Type ); break;

    }/* end switch */

  /* end while */

}/* end main */
```

Appendix E. C Micro-typographic Experiment Sample (continued)

```
/* ------------------------------------------------------------------ */

#define MaxVal   100     /* maximum depth of Val stack */

int     Sp = 0;           /* stack pointer */
double  Val[MaxVal];      /* value stack */

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



double Push(F)            /* push F onto a value stack */

   double F;
{
   if ( Sp < MaxVal )  return( Val[Sp++]=F );

   else {
      printf( "error: stack full\n" );
      Clear();
      return( 0 );
   }/* end if-else */

}/* end Push() */



double Pop()   /* Pop top value from stack */
{
   if ( Sp > 0 )  return( Val[--Sp] );

   else {
      printf( "error: stack empty\n" );
      Clear();
      return( 0 );
   }/* end if-else */

}/* end Pop() */



Clear()   /* Clear stack */

{  Sp = 0;  }
```

Appendix E. C Micro-typographic Experiment Sample (continued)

Name ____________________________

1. Which two functions are called but not defined in the reverse
Polish desk calculator program ? Getop and AtoF

2. What input to the desk calculator causes the stack to be
cleared ? "c"

3. What is the maximum number of stack entries ? 100 (maxval)

4. The result of an arithmetic operation is
a) printed to standard output
b) popped from the top of the stack
[✓ c)] pushed onto the top of the stack
d) converted to string form

5. The op2 variable is used for
a) flushing the stack.
[✓ b)] temporarily holding the top operand.
c) implementing the unary minus operation.
d) checking the validity of operands.

6. What is the stack pointer value when the stack is empty ? Ø

7. What happens to the top entry on the stack when an equal sign
is entered into the desk calculator ? pop, print, and push

8. What does the atof function do ? converts alpha to floating

9. What happens when a parentheses is entered into the desk
calculator ? prints "unknown command"

=== === === === === === === === === === === === === === === ===

Please look at the clock and write down the time as accurately as possible.
TIME: ________________

10. How would you rate this code for clarity and readability?
1......2......3......4......5
very poor o.k. good very
poor good

STOP -- Be sure your name is above. Turn the test face down, remain seated and quiet.

Appendix F.

Experiment 4: Empirical Studies Experiment Sample Test Packet

Name ______________________________

Yrs. experience ___________________

Yrs. C exp.________________

We are trying to determine how professional programmers go about understanding a large program. There are 4 parts to this exercise, each part takes about 30 minutes and will be explained in detail when we get there. You will be paid $40 for your help.

Do you have any general questions before we start part 1 ?

Part 1. Imagine this scenario:

The programmer responsible for the X_Windows_Info system has just quit. You are now responsible for maintaining that program. Here is a listing of the xwininfo.c program and its include files dsimple.h and dsimple.c. The program was originally written at MIT and is now supported by HP as part of their Unix system.

The programmer that quit will be here in 30 minutes. At that time you can ask him any questions you want about the program. He is leaving the company so this will be your one and only opportunity to ask him questions. For the next 30 minutes I want you to study this program and think of questions you will want to ask him.

We are interested in what you think of as you look at this program; so as you look at the program please THINK OUT LOUD. That is, as you study the program, explain what you're doing and why. I will occasionally ask you questions about what you've done. I'll be recording everything you say so you can simply verbalize all your questions and comments. At the end of the 30 minute study period I will answer as many questions as I can and then we'll move on to part 2.

Do you have any questions before you begin?

Appendix F. Empirical Studies Experiment Sample (continued)

Part 2. I'm going to ask you some general questions about the program. Just answer verbally. You can use the listing to help with your answers.

1. What is the "usage" routine for ?

prints help message info about command line arguments if -help is requested or there is an error on command line

2. Why does Select_Window_Args call Window_With_Name ?

Select_Window_Args parses some of the command line args, it calls Window_with_name to ① parse the command line for the -name argument, ② find that window, and ③ return a "w" structure to Select_Window_Args. Hence, "to get a window" based on command line args.

3. The get_error routine is only 10 lines and is just called once by Display_Window_Id. Why is it a separate routine ?

Display_Window_Id calls an X library function and passes that function the get_error routine as an argument. Hence, it must be separate.

4. The xwininfo.c program gets and displays information about windows. There are two ways users can select the window they want information on. What are the two ways ?

a. via command line (-id or -name) options

b. via mouse selection (clicking on a window)

5. What does the Lookup routine actually lookup ?

Each Display routine establishes a local table of alpha strings and numeric indices. The lookup routine takes the index (code and the table as arguments and returns the corresponding string.

6. What command line parameters to xwininfo.c will cause "all" window information to be displayed ?

a) The -all option

b) listing -tree, -stats, -bits, -events, -wm, and -size all on the command line

7. What command line parameters to xwininfo.c will cause only "stats" window information to be displayed ?

a. - stats option

b. listing no info parameter (-stats is the default option)

Appendix F. Empirical Studies Experiment Sample (continued)

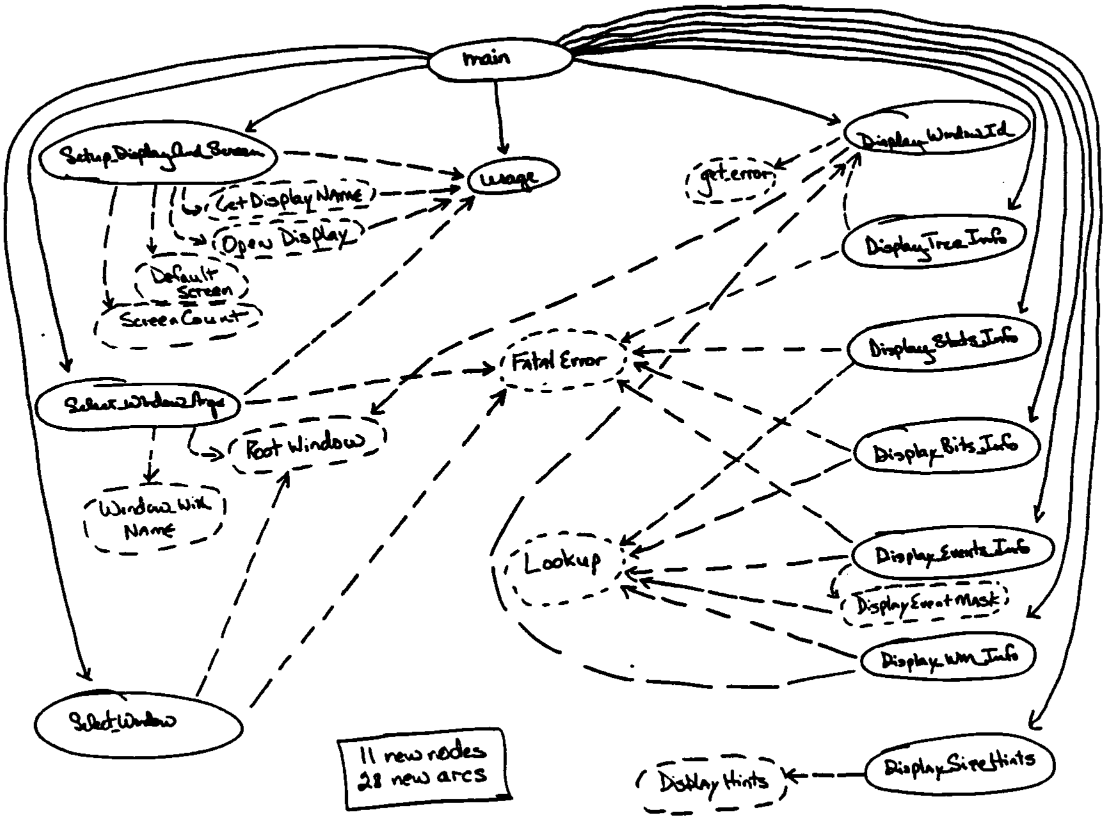
Part 3.

Here is the beginning of a "uses graph," also known as a call chart, for the xwininfo.c program. We will trace through the top part together so I'm sure you understand what I did to get it this far.

Now I'd like you to finish the uses graph. You can skip over all routines that begin with X, because they're not included here. Also, there's some other routines that aren't defined -- I'll point them out when you get to them.

This is a timed and scored exercise. That is, I'll be measuring both the speed and accuracy of your work, so work as fast as you can without sacrificing accuracy.

Do you have any questions before you start?



Appendix F. Empirical Studies Experiment Sample (continued)

Part 4.

I'm going to give you a hypothetical maintenance situation. I want you to descibe how you'd go about doing the task.

1. Users want information about window colors. How would you add a Display_Colors option?

2. If you had to understand another program of this size and complexity and you where in your office, how would you go about studying the program ?

3. If you had a multiple windows, large screen monitor, and had to work with a large program, what would you put in the windows ? How/what would you look at to get an overall feel for what a program does ?

4. This is what we call the "book" paradigm of source code listing.

   a. Do you think listing programs in this manner would be helpful ?

   b. Do you think a multi-windowed electronic display of a book listing would be helpful ? If so, what should go in each window?