Guidelines for using style to improve computer program comprehension have often been proposed without empirical testing. This thesis reports on the results of three controlled experiments that investigated ways program style may be used to aid comprehension of source code listings.

Experiments 1 and 2 were conducted using advanced computer science students as subjects and short Pascal programs. Results showed that student programmers used meaningful identifier names as important sources of information during comprehension of short programs.

A review of the literature showed the need for the thesis' proposed methodology for designing controlled experiments on program comprehension that produce results which generalize well to situations involving professional programmers working on real world tasks. This methodology was used to design Experiment 3.
Text comprehension researchers have investigated the use of signaling, or the placement of non-content information, in a text in order to emphasize certain ideas and/or clarify the organization. Experiment 3 investigated the role of signaling in another domain, that of computer program source code listings.

The experiment had professional programmers study a 913-line C program. Three types of signals were investigated: preview statements, headings, and typographic cueing. The major results were

(a) meaningful module names served as headings in the source code listing and helped professional programmers understand and locate information in the program;
(b) header comments, when written as preview statements, helped professional programmers understand and locate information in the program;
(c) typographic cueing, designed to provide emphasis and segmentation cues, helped programmers understand the program; and
(d) the effects of meaningful names, header comments, and typographic cueing were additive. No significant interactions of effects were observed.

Based on these results, guidelines are proposed for ways programmers may use comments and module names in source code to act as signals that aid future readers. In addition, guidelines are suggested for adding typographic signaling to provide emphasis to the comments and names.
Using Signaling to Aid
Computer Program Comprehension

by

Edward M. Gellenbeck

A THESIS
submitted to
Oregon State University

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

Completed May 10, 1991
Commencement June 1992
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 May 10, 1991
Typed by Ed Gellenbeck
Acknowledgements

I wish to express my appreciation to each member of my doctoral committee for their contribution to this research. In particular, to Curtis Cook, my major professor, for making me stick to the topic and see it through. To Donna Cruse for being a friend and advisor. To Tom Dietterich, Ted Lewis, and James Coakley for their comments and suggestions.

I also wish to thank UES-USAF for their support under the Laboratory Graduate Fellowship Program during three years of my doctoral study. In addition, I thank Dick Schori for arranging this support and serving as a mentor during my years at Oregon State University.

Finally, I wish to thank my parents for their encouragement and support. To Jean Lofy, who makes it all worthwhile. And to God.
# Table Of Contents

1. Introduction ................................................................................................................. 1
   1.1 Motivation for the Study......................................................................................... 2
   1.2 Definition of Key Terms......................................................................................... 4
   1.3 Scope of the Research............................................................................................ 6
   1.4 Objectives of the Research..................................................................................... 8
   1.5 Fundamental Assumptions..................................................................................... 9
   1.6 Organization of the Thesis.................................................................................... 10

2. Using Signaling to Aid Text Comprehension: A Review of the Literature ............... 12
   2.1 Introduction........................................................................................................... 12
   2.2 Verbal Signals....................................................................................................... 13
   2.3 Typographic Signals............................................................................................. 20
   2.4 Conclusions........................................................................................................... 25

3. Beacons and Signaling in Computer Programs: A Review of the Literature .......... 27
   3.1 Cognitive Models of Program Comprehension..................................................... 27
   3.2 Beacons in Computer Programs............................................................................ 35
   3.3 Signaling in Computer Programs......................................................................... 40
   3.4 Conclusions........................................................................................................... 46

4. A Framework for Program Comprehension Research............................................. 48
   4.1 Problems with Methodology.................................................................................. 48
   4.2 A Framework for Program Comprehension Research......................................... 59
4.3 Using the Framework in Controlled Studies .................. 69
4.4 Conclusions ........................................ 76

5. Experiments 1 and 2: Using Meaningful Identifier Names
   To Aid Comprehension .................................. 78
   5.1 Experiment 1 ..................................... 78
   5.2 Experiment 2 ..................................... 92
   5.3 Conclusions ....................................... 98

   6.1 Introduction ..................................... 100
   6.2 Method .......................................... 104
   6.3 Results ......................................... 117
   6.4 Discussion ....................................... 133

7. Suggestions for Using Signaling to Aid Comprehension .......... 138
   7.1 Using Signals with Hard-Copy Listings ................. 138
   7.2 Using Signaling to Aid On-Line Browsing ............ 142

8. Conclusions and Future Work .................................. 145
   8.1 Review of Results ................................ 145
   8.2 Limitations of the Study .......................... 146
   8.3 Conclusions ..................................... 147
   8.4 Future Research ................................... 148

Bibliography .............................................. 150
Appendices

A. Binary Search Procedures, Experiment 1 ....................... 160
B. Sort Procedures, Experiment 1 .................................. 161
C. Background Questionnaire, Experiment 1 ..................... 162
D. Binary Search Procedures, Experiment 2 ..................... 163
E. Background Questionnaire, Experiment 2 .................... 164
F. Informed Consent Form, Experiment 3 ....................... 165
G. Background Questionnaire, Experiment 3 .................... 166
H. Preview Statements, Experiment 3 .......................... 167
I. Module Names, Experiment 3 ................................. 170
J. Hypothesis Questions, Experiment 3 ......................... 171
K. Location Questions, Experiment 3 .......................... 174
L. Incorrect Responses to Hypothesis Questions, Experiment 3 .... 176
M. Confidence Ratings for Hypothesis Questions, Experiment 3 ...... 177
N. Response Times to Hypothesis Questions, Experiment 3 ........ 179
O. Incorrect Answers to Location Questions, Experiment 3 ........ 181
P. Types of Errors to Location Questions, Experiment 3 ........... 182
Q. Response Times to Location Questions, Experiment 3 .......... 183
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Beacon Example: Swap Operation in a Sorting Routine</td>
<td>36</td>
</tr>
<tr>
<td>4.1 The Tetrahedral Model Applied to Program Comprehension Experiments</td>
<td>70</td>
</tr>
<tr>
<td>6.1 Eight Conditions, Experiment 3</td>
<td>111</td>
</tr>
</tbody>
</table>
List of Tables

Table | Page
---|---
3.1 Sources of Beacons [from Brooks, 83] | 36
4.1 Summary of Subject Selection for Controlled Experiments on Program Comprehension | 50
4.2 Summary of Choice of Materials for Controlled Experiments on Program Comprehension | 55
4.3 A Framework for Program Comprehension | 60
5.1 Percentage of Subjects Correctly Identifying the Binary Search Procedure, Experiment 1 | 83
5.2 Chi Square Tests of Differences: Binary Search Procedures, Experiment 1 | 83
5.3 Mean Confidence Ratings: Binary Search Procedures, Experiment 1 | 84
5.4 Percentage of Subjects Correctly Identifying the Sort Procedure, Experiment 1 | 85
5.5 Chi Square Tests of Differences: Sort Procedures, Experiment 1 | 86
5.6 Mean Confidence Ratings: Sort Procedures, Experiment 1 | 87
5.7 Percentage of Subjects Correctly Identifying the Binary Search Procedure, Experiment 2 | 96
5.8 Chi Square Tests of Differences: Binary Search Procedures, Experiment 2 | 96
6.1 Subject Background Data, Experiment 3 | 107
6.2 Hypothesis Questions -- Number of Incorrect Responses
   by Condition................................................................. 118

6.3 Hypothesis Questions -- Number of Incorrect Responses
   by Question................................................................. 119

6.4 Hypothesis Questions -- Mean Confidence Ratings by Condition .... 120

6.5 Hypothesis Questions -- Mean Confidence Ratings by Question ...... 121

6.6 Hypothesis Questions -- Response Time Means by Condition ........... 122

6.7 Hypothesis Questions -- Ranking of Conditions by Response Times . 123

6.8 Hypothesis Questions -- Response Time Main Effects ..................... 123

6.9 Hypothesis Questions -- Response Time Means by Question .......... 125

6.10 Location Questions -- Number of Incorrect Responses Attempted
   by Condition................................................................. 127

6.11 Location Questions -- Number of Incorrect Responses
   by Question................................................................. 128

6.12 Location Questions -- Types of Errors by Condition ..................... 129

6.13 Location Questions -- Mean Location Times by Condition ............ 130

6.14 Location Questions -- Ranking of Conditions by Response Times... 130

6.15 Location Questions -- Location Time Main Effects ..................... 131

6.16 Location Questions -- Location Time Means by Questions .......... 132
Using Signaling to Aid Computer Program Comprehension

Chapter 1

Introduction

Few people would argue the importance of program comprehension in computer programming. Obviously, development programmers need to understand the programs they write; maintenance programmers need to understand the programs they modify; students need to understand the program examples they study at school. What is not obvious are the factors which make one program easier to understand than another.

This research investigated using signaling as a way to make programs easier to read and understand. Signaling refers to the addition of non-content information in order to emphasize certain ideas and/or clarify the organization of a text [Meyer, 75]. While the use of signaling with textbooks is widespread, making the texts more accessible and comprehensible to their readers, almost no research
has looked at ways to use signaling with computer program source code listings in order to make them more accessible and comprehensible to their readers.

Chapter 1 consists of six sections. The first discusses the motivation for this study and its contribution to the computer science field. The second defines key terms. The third section presents the research objectives; the fourth explains the scope of the research. Fifth, the underlying assumptions to the research are made explicit. Finally, the chapter concludes with an overview of the content of the remainder of the thesis.

1.1 Motivation for the Study

The long-term goal of this research is to reduce software costs through improved programmer productivity by making programs easier to understand. Since program comprehension is a subtask of so many programming activities, making programs easier to understand should speed up many programming tasks and improve software quality.

A number of studies have demonstrated the importance of program comprehension. Fjeldstad and Hamlen [83] found that maintenance programmers devote as much time to comprehending the original program as they do to coding an enhancement. In addition, they reported that maintenance programmers depended primarily on study of the source code for comprehension and that the programmers cited understanding the intent and style of the code as the hardest part to making a change.

Littman, Pinto, Letovsky, and Soloway [86] found a strong relationship between program comprehension and performance on an enhancement task. Nanja
and Cook [87] showed that the key difference between expert and novice programmers during debugging was the experts’ superior understanding of the program being debugged.

There are a number of approaches to making programs easier to understand. Many of these approaches are radical. Proponents create new languages, new programming paradigms, and knowledge-based reasoning systems. While order-of-magnitude improvements may result from these approaches, they do not directly address the problem of how to make the huge body of structured, procedural code in use today more comprehensible to the people working with it.

Our research follows a more conservative approach to making programs easier to understand: that is, using typographic style to improve the comprehensibility of the source code. Oman and Cook [90b] defined typographic style as the set of characteristics concerned with the formatting and commenting of source code that do not impact the execution of the program.

Typographic style can be used to make existing programs, as well as future programs, easier to understand. This is important because almost all program maintenance tasks deal with modifying existing programs. By targeting our research to aid the enormous task of maintaining existing programs, even modest improvements can yield significant benefits.

Our goal is to develop guidelines for typographic style based on empirical studies. At times, programmers stylize their programs based on their own intuitive ideas of what makes a program understandable. More often, they stylize their programs based on company standards or with an automatic formatting program. We want to provide an empirical base to help guide the typographic style decisions
made by programmers, creators of company standards, and designers of the automatic formatting programs.

This research contributes directly to computer program visualization, an emerging field within computer science. Baecker and Marcus [90] define computer program visualization as the "use of computer graphics to enhance the art of presenting and communicating programs and thereby to facilitate the understanding and effective use of computer programs by people." In addition, this research contributes to the fields of software engineering, human-computer interaction, empirical studies of programmers, and computer science education. Due to its interdisciplinary nature, these results should also be of interest to text comprehension researchers.

1.2 Definition of Key Terms

This section defines the key terms used in this research. A more thorough discussion of these terms appears in chapters 2 and 3.

Most people have an intuitive notion of what program comprehension means, but these notions differ among individuals. While the goal of program comprehension is to understand an unfamiliar program, the level of understanding will vary. Sometimes a comprehender develops a superficial understanding of what a program does; at other times, a comprehender may develop an in-depth understanding of how a program works.

This research focuses on situations where programmers gain their understanding by reading source code. Examples of such situations include (a) re-engineering tasks performed by maintenance programmers on programs written by
development programmers, (b) code reviews where programmers read and critique code written by their colleagues, and (c) learning situations where students learn to program by reading code written by their textbooks' authors.

When viewed this way, program comprehension resembles the situation studied by text comprehension researchers. That is, the program comprehender is the reader and the source code is the text. Fortunately, text comprehension is a well-studied field of psychology, and we used results from research in text comprehension to help direct our investigation.

Our research assumes a top-down, hypotheses-driven model of program comprehension, such as the one proposed by Brooks [83]. According to his model, readers comprehend programs by forming hypotheses and selectively searching the source code to confirm the hypotheses and to develop new ones.

An important question is: what are the types of information programmers search for in source code? Brooks suggested that programmers searched for beacons, or sets of key features that typically indicate the presence of a particular structure or operation. For example, the swap operation is a beacon because it typically indicates a sort operation.

The notion of signaling comes from text comprehension research. A comparison between a college textbook and a popular novel will immediately yield one big difference: textbooks make extensive use of signaling to aid their readers in selectively sampling the information. Examples of signaling include the use of (a) typographic cueing, such as larger fonts and boldfacing, to show level of importance and (b) preview statements to inform the reader of the topic being discussed and to clarify the organization of the text, and (c) headings to help the reader access the material.
The signaling most often seen in source code listings is spatial layout. Either the programmer or an automatic formatting program, such as a prettyprinter or a syntax-directed editor, adds a consistent indentation and alignment style to help readers grasp the program’s control flow, nesting, and module structure.

1.3 Scope of the Research

In this dissertation, we investigate using signaling as a way to aid program comprehension. To reduce the scope of the investigation, we restrict the program comprehension environment to hard-copy source code listings. However, our results should aid in the design of on-line software browsing systems as well.

Hard-copy listings are still widely used by professional programmers. Oman and Cook [90a] reported on a poll of professional programmers in which they found that hard-copy listings were still being used even though the programmers had state-of-the-art workstations in their offices.

While the use of signaling is widespread in textbooks, it is still a largely untapped resource for making programs more understandable. High resolution output devices, most notably the laser printer, permit typographic cueing in source code listings that rivals the sophisticated typesetting systems used with textbooks.

We feel there is an urgent need for basic empirical research on the effectiveness of signaling to aid computer program comprehension. To provide a narrower focus, we selected preview statements, headings, and typographic cueing as the types of signaling to investigate through controlled experimentation.

Surprisingly, given the importance of beacons in Brooks’ top-down model of program comprehension, they remain under-investigated and hard to identify.
Only Wiedenbeck [86, 89] has published research on beacons, and her work has focused on the swap operation in a sort routine.

We began our investigation with the idea that using typographic cueing to draw attention to beacons in source code should aid comprehension. However, after several unsuccessful attempts to identify code idioms that serve as beacons besides that swap operation, we concluded that the swap operation may be an anomaly and not representative of the types of information used by programmers during comprehension, especially with large programs.

Instead, we identified meaningful identifier names and comments as key sources of information used by programmers during comprehension and investigated their role in making programs easier to understand. Using meaningful identifier names and commenting to aid comprehension falls under the category of typographic style decisions since neither affects the execution of a program. A top-down model of comprehension predicts that using signaling to draw attention to meaningful identifier names and comments should aid comprehension.

The research most closely related to ours is Oman and Cook [90a, 90b] and Baecker and Marcus [90]. The major difference between our work and these two is in scope. By focusing on the use of signaling to aid program comprehension and by using results from research in text comprehension to help direct our research, our results augment the ongoing work of both Oman and Cook and Baecker and Marcus.

This goal of this research is inspired by Knuth’s [84] call for literate programming. Knuth contends that one should be able to read and enjoy a program’s source code the same way one does a fine book. Literate programming will be made possible through new ways of writing, documenting, and presenting
programs. This research contributes to literate programming by treating source code as a text and suggesting ways to make it more readable.

1.4 Objectives of the Research

This section discusses the research objectives. Our primary goal is to develop ways to make source code listings easier to read and understand. In particular, we focus on ways to use signaling in source code. We investigated three types of signaling, preview statements, headings and typographic cueing, with the following objectives:

1. Show that preview statements significantly aid comprehension.
2. Show that headings significantly aid comprehension.
3. Show that typographic cueing significantly aids comprehension.
4. Investigate the interactions of preview statements, headings, and typographic cueing on program comprehension.
5. Develop typographical style guidelines based on the results from objectives #1-4 in order to come up with ways of presenting programs that are easier to read, access, and understand.

In addition, after conducting several pilot experiments that included large programs as part of the experimental material, we saw the need to apply ideas from text comprehension research and cognitive psychology to

6. Develop a methodology for controlled experimentation on program comprehension that improves the external validity of the experiments.
1.5 Fundamental Assumptions

This section describes five assumptions underlying this research. We present these assumptions with minimal justification, since we do not expect controversy. In the following chapters we offer empirical evidence in support of each of these assumptions.

Assumption 1: Improving program comprehension results in superior performance on a variety of programming tasks. We consider program comprehension a basic programming skill. It plays a crucial role in successful software re-engineering, software re-use, software evaluation, and programmer education.

Assumption 2: A top-down model of program comprehension, such as Brooks' [83], is an appropriate theory for modeling comprehension of large programs. We assume that readers form hypotheses about the program and search the source code for information to confirm their hypotheses. Therefore, a top-down theory predicts that using typographic cueing to draw attention to key information in the source code will aid comprehension.

Assumption 3: Schema theory can be used to elaborate Brooks' top-down model of program comprehension. Schemas refer to cognitive structures that specify the general properties of objects or events while leaving out the nonessential details [Stillings, et al., 87]. We assume that programmers make use of schemas to select, organize, and integrate source code during comprehension.

Assumption 4: Typographic style changes can make programs easier to understand. While results from empirical studies on the role of typographic style on program comprehension have been mixed, both Oman and Cook [90b] and
Baecker and Marcus [90] showed that typographic style can be used to make programs more understandable.

**Assumption 5:** Results from properly designed controlled experiments on typographic style should generalize to guidelines that help make real world programs more understandable.

### 1.6 Organization of the Thesis

The remainder of this thesis is organized into three major sections. First, in Chapters 2, 3, and 4 we review research related to our research. Next, in Chapters 5 and 6 we present the results of two empirical studies we conducted. Finally, in Chapters 7 and 8 we present our major conclusions and offer guidelines for using signaling to aid program understanding.

While there are a number of studies that bear on our research, there are only a few that directly relate. We divided our literature review section by discipline. In Chapter 2, we review text comprehension literature; in Chapter 3, we review program comprehension literature. Chapter 4 examines the methodological problems common to many controlled experiments on program comprehension and proposes a way of dealing with these problems.

In Chapter 5, we present the results of two experiments we conducted that showed meaningful identifier names aid comprehension. Experiment 1 showed that both meaningful procedure and variable names aided high-level comprehension of short Pascal procedures. Experiment 2 investigated only variable names and replicated the results of Experiment 1.
In Chapter 6, we present the results of an experiment we conducted that showed that signaling can aid professional programmers with understanding and locating information in large programs. Three types of signaling were investigated: preview statements, headings, and typographic cueing.

Chapter 7 offers guidelings for making programs easier to understand based on the results from our experiments.

Finally, in Chapter 8, we conclude the thesis by restating our major conclusions, pointing out the limitations of our study, and commenting on topics for future study.
Chapter 2

Using Signaling to Aid Text Comprehension:

A Review of the Literature

In the next three chapters, we review literature related to our research. Chapter 2 surveys research from the text comprehension literature that investigated ways to use signaling to make textbooks more understandable. We present this information prior to our review of program comprehension literature in order to define signaling and provide evidence for its effectiveness. Next, in Chapter 3 we focus on the theory and evidence from program comprehension literature that directly bears on our research questions and experimental hypotheses. We conclude our literature review section with Chapter 4, a review of the methodology used with controlled experimentation in program comprehension and suggestions on ways to improve this methodology.

2.1 Introduction

In Chapter 2, we review research on the use of signaling to aid readers of English text. Signaling refers to the placement of non-content information, or signals, in a text in order to emphasize certain ideas and/or clarify the organization
Researchers have proposed that signals help readers call up appropriate mental schemas, form hierarchical frameworks in which to store information, decide what information is important, and check the correctness of their integration and storage of information in memory [Spyridakis and Standal, 87].

There is a large body of practical research on various types of signaling techniques and their effectiveness. Surprisingly, given the similarities between text and computer program comprehension, almost none of these techniques have been investigated or used with computer program source code listings.

The remainder of this chapter is divided into three sections. First, we present an overview of research on verbal signaling. Verbal signaling uses words, phrases and sentences to cue ideas and/or structure. Most of the empirical research on signaling has investigated verbal signals. We present this material in order to review both (a) the major research results and (b) the lessons learned about designing controlled experiments that investigate the effectiveness of signaling. In the second section we focus on typographic signaling. Typographic signaling involves the use of physical cues, such as indentation and italics, to aid text comprehension. Finally, we end by citing some open areas of research and suggest ways our research is of interest to text comprehension researchers.

2.2 Verbal Signals

Research on signaling has most often investigated verbal signaling. Indeed, many researchers have restricted the definition of signaling to refer to the addition of non-content words and phrases that emphasize the conceptual structure
or organization of a passage [e.g. Loman and Mayer, 83; Spyridakis and Standal, 87].

In this section, we first review the types of verbal signals researchers have identified and investigated. Following that, we summarize the results of empirical studies on verbal signaling and discuss reasons why many of these studies did not find significant differences between signaled versus non-signaled versions of the same texts.

2.2.1 Types of Verbal Signals

Meyer [75] identified four types of verbal signals: (a) structural cues that convey information about the structure of relationships among ideas in the text, (b) preview statements that reveal information occurring later in the text, (c) summary statements that restate important ideas, and (d) pointer words that provide emphasis. Discussion on each of these four types follows.

**Structural Cues.** Structural cues consist of words and phrases that specify the structure of relationships among ideas in the text. The text's structure is important because it establishes both the logical connections among ideas as well as the hierarchy or level of importance of ideas relative to each other [Meyer and Rice, 89]. Many studies have demonstrated that understanding the structure of a text is a distinguishing characteristic between good and poor readers.

Meyer, Yound and Bartlett [89] identified six basic text structure relationships which they called plans. They suggested that authors use structural cues as signals to help their readers recognize these plans in the text. These plans included:
(a) The description plan that presents attributes, specifics, or setting information about a topic. Authors can use words like for example and such as to signal the description plan.

(b) The sequence plan that groups ideas together based on order or time. Authors can use words like first and next to signal the sequence plan.

(c) The causation plan that presents causal relationships between ideas. Authors can use words like as a result and because to signal the causation plan.

(d) The problem/solution plan that organizes ideas into two parts: a problem part and a solution part. The problem part poses the problem or raises a question. The solution part responds to the problem or answers the question. Authors can use words like problem and question to signal the problem parts and words like solution and answer to signal the solution parts.

(e) The comparison plan that connects ideas on the basis of differences and similarities. Authors can use words like but and in contrast to signal the comparison plan.

(f) The listing plan that groups ideas together. The listing plan can occur with any of the other five plans. Authors can use words like in addition and also to signal the listing plan.

Meyer, Brandt, and Bluth [80] claimed that when readers clearly understand the structural relationships among ideas in the text, they use a reading strategy involving selective attention. A strategy of this type encourages the reader to construct a hierarchical, well-organized internal representation of the material. In contrast, poor comprehenders fail to understand the structural relationships
among ideas and follow a default/list strategy. With this strategy, readers have no focus and view each idea as a separate entity. Readers’ mental representations based on a default/list strategy make no attempt to interrelate ideas. Instead, comprehension becomes a rote memorization of facts.

Mayer, Dyck and Cook [84] suggest that headings and sub-headings may also serve as structural cues. They argue that headings and sub-headings specify the hierarchical structure of a text and help readers identify its major content units. We feel that headings offer two types of signals to readers. One of these types is structural, as Mayer, et al. suggests. Using typographic signaling to offset the headings greatly increases their value as structural cues. The other type of signaling that headings provide is the use of the content of the headings as preview statements. The following section discusses the use of preview statements as a signaling technique.

**Preview Statements.** Preview statements are words and phrases that reveal information appearing later in the text. These statements do not add new content material, rather they repeat information abstracted from the text which follows [Britton, 86]. Preview statements often take the form of an introductory or topic sentence that announces the material which follows. The contents of titles, chapter headings, and sub-headings can also function as preview statements if readers take the time to read them.

Kozminsky [77] summarized research on the use of titles and headings before exposure to the text. In general, the use of a title or heading prior produced better recall compared to the control condition. She concluded that titles and section headings function as pointers into memory and that readers use them to activate schemas with which to add the new information. If, however, the material
is completely new to the reader, then readers use titles and headings as labels or anchor points in memory around which to organize the new material.

Glover et al. [88] claimed that preview statements aided comprehension in two ways. First, like Kozminsky [77], they felt that preview statements served to activate relevant information in memory. With the relevant memory active, subjects found it easier to understand and integrate the new information. Second, they claimed that the preview statements served to direct readers' attention during reading. Glover, et al. used a secondary task to show that readers devoted more attention to signaled than non-signaled information.

**Summary Statements.** Like preview statements, summary statements repeat material contained in the text. As their name implies, readers encounter summary statements after they have seen the same ideas in the text. While the use of preview statements has been a frequent topic for experimentation, few researchers have investigated summary statements as a signaling technique.

Glover, et al. [88] provided evidence that summary statements serve as signals for aiding text comprehension. They concluded that summary statements serve to mark ideas in the text as important. In addition, summary statements serve to call up the summarized ideas in memory and aid integration of the new ideas with the summarized ones.

**Pointer Words.** Pointer words show subordination of some ideas to others. Authors use pointer words to identify ideas that they consider central to the topic and to de-emphasize other ideas that they consider peripheral. Examples of pointer words include *foremost* and *unfortunately.*
Pointer words may be used to achieve two types of emphasis: normal and differential [Meyer, 81]. Normal emphasis reflects a correspondence between the hierarchical content structure of the text and its level of importance. With normal emphasis, authors highlight high-level information in the content structure of the text, such as the major theme and topics. In contrast, differential emphasis highlights information lower in the content structure of the text. Authors use differential emphasis to draw attention to specific ideas which, while lower in the hierarchical content structure of text, they feel are particularly important for comprehension.

2.2.2 Summary of Empirical Research on Verbal Signaling

Even though the use of verbal signaling is widespread and considered a requirement for effective writing, results from empirical studies on their effectiveness are inconclusive and occasionally contradictory.

Bonnie J. F. Meyer is one of the most active researchers investigating verbal signaling. In [Meyer, 81], she summarized the research on verbal signaling:

Signals highlighting normal emphasis may have little influence on students with high and low skill levels, but substantial effects on middle range students and adults no longer in school. The research with college students indicates that these highly skilled students are affected substantially by signaling reflecting differential emphasis. This type of signaling appears to increase the recall of the signaled information and depress the recall of nonsignaled information, especially when nonsignaled information is at equivalent or lower levels in the content structure.

In other words, the effectiveness of signaling depends both on the characteristics of the subject population and on the type of emphasis of the signaling.
Loman and Mayer [83] demonstrated that signaling can produce a qualitative effect on comprehension. They suggested that previous mixed results from research investigating signaling may be due to the fact that signaling does not increase the total amount of information recalled. Rather, signaling influences the type of ideas recalled and the readers' mental organization of the material. Loman and Mayer showed that signaling produced better recall of conceptual information and led to the generation of high quality problem solutions. Without the signals, subjects were better at recalling information from the beginning and end of a passage and produced lower quality problem solutions. Overall total recall of the text by the experimental and control groups did not differ. Loman and Mayer concluded that signaling helped to direct readers towards conceptual information in the text and that signaling encouraged readers to build internal representations containing causal links that transfer to new problem situations.

Britton, Glynn, Meyer, and Penland [82] showed that signaling can reduce the cognitive processing required to read technical text. They claimed that structural cues, by cueing the structural relationships among ideas, helped readers construct internal representations of the text content. Without the structural cues, readers must infer the relationships among ideas in the text. They concluded that this inferring activity used cognitive capacity and was responsible for the increase in cognitive processing they observed with the control group.

Glover, et al. [88] showed that material signaled by preview statements received more cognitive processing compared to the same material without the preview statements. In addition, they showed material prefaced with summary statements received more cognitive processing compared to the same material without the summary statements. Their results do not contradict Britton, et al.
Rather, they conclude that different types of verbal signals affect cognitive processing in different ways.

Spyridakis and Standal [87] showed that the effectiveness of signaling depends, at least in part, on passage length and difficulty. In their study, signaling proved most effective when the text was most appropriately challenging for their subjects, that is, neither too easy nor too difficult.

Spyridakis [89a] reviewed 25 recent studies on signaling. Results across experiments were inconsistent and often contradictory. She concluded that the mixed results may be attributable to methodological problems. She suggested the use of longer and more difficult passages. In addition, looking at the types of ideas recalled better measured the effects of signaling on comprehension than just looking at total recall.

2.3 Typographic Signals

In this section, we review research on the use of typographic signals to aid text comprehension. We define typographic signals as physical cues added to text to aid readers during comprehension. Like verbal signals, typographic signals do not add new content material. Instead, they tell readers something about the structure of the text and/or draw attention to some ideas over others.

Typographic signaling has not been as well-investigated as verbal signaling. We present our literature review in two sections. First, we review literature on the types of typographic signals researchers have identified and investigated. Following that, we review the literature on empirical studies that investigated the effectiveness of typographic signaling for aiding text comprehension.
2.3.1 Types of Typographic Signals

We have categorized the types of typographic signaling into two groups: (a) those that cue text structure by physically distinguish meaningful components of a text and (b) those that provide differential emphasis to material in the text.

Typographic signaling that cues text structure can aid readers during inspection of a text's structure, navigation through the text, and location of information. The most common example of typographic signaling to cue text structure is the use of indentation to signal paragraphs. A new paragraph tells the reader that one subarea of the topic has ended, and a new one is beginning [Britton, 86]. While almost all types of prose use some type of typographic signaling to cue new paragraphs, our interest is with types of typographic signaling primarily encountered in long expository text passages such as textbooks.

Waller [77] contends that even when written in a foreign language, one can easily distinguish between textbooks and novels solely on the basis of appearance. Textbooks make extensive use of typographic signaling to show their structure while novels most often just use verbal signals. At the macro-level, the use of typography to distinguish the table of contents, index, glossaries, and prose components of a textbook is essential. Imagine a table of contents or index laid out as continuous prose, it would be almost completely unusable for its intended purpose. Instead, designers of textbooks have adopted typographic conventions so, for example, readers can recognize a table of contents from just its appearance.

Limiting the scope of our discussion to the continuous prose component of a textbook, Waller [80] reports on four ways that designers of textbooks can use typographic signaling to cue text structure:
(a) Delineating the beginning and end of sections in the text. For example, designers may typographically signal the start of new chapters by beginning each on a new page, labeling it with the next chapter number, and beginning the text half-way down the page. Vertical spacing, ruled lines, and indentation conventions are other examples of typographic signaling techniques that designers can use to delineate section of the text.

(b) Indicating the insertion or juxtaposition of a short segment of text into a longer one in such a way as to preserve the continuity of the longer segment. Examples include the use of footnotes, boxed inserts, marginal notes, and indentation.

(c) Identifying the components of sets or series. For example, numbering the elements of a list.

(d) Showing a change in the mode or tone of discourse. For example, using a larger type size to differentiate the section headings from the text that follows.

The other category of typographic signaling is signaling that cues differential emphasis. Most of the research on this type of typographic signaling have investigated underlining as the signaling technique. The emphasis on underlining may be due to the ease with which underlining can be added to typewritten text. The use of color, italics, boldface, and all capital letters to provide emphasis has received considerably less attention then underlining. The assumption exists that each of these typographic signaling techniques can serve to provide emphasis provided the reader is aware that it is being used as a signaling device.
2.3.2 Summary of Empirical Research on Typographic Signaling

Researchers have frequently investigated the use of typographic signaling as a way to cue emphasis (or de-emphasis) to certain material in a text. The main conclusion is that typographic signaling to cue emphasis improves recall of the signaled material without affecting total recall. However, like the results from research on verbal signals, the results from controlled experimentation on the effectiveness of typographic signaling to provide emphasis have been inconclusive and occasionally contradictory.

Most people intuitively think underlining key ideas in a text helps comprehension. However, results from empirical studies have been mixed. Hartley, Bartlett, and Branthwaite [80] reviewed 19 controlled experiments that investigated the effect of underlining to provide emphasis. Six of the studies found positive effects, 12 of the studies found no effects, and one study found a negative effect for underlining. Their own experiment showed underlining helped sixth grade students recall underlined words and did not affect recall of the other information in the text. They concluded that underlining only works when the reader is aware that underlining indicates that an idea is important.

The size and difficulty of the passage may also affect the usefulness of typographic signaling. Crouse and Idstein [82] showed that underlining important information helped comprehension of a 6000-word passage. In contrast, underlining did not help when the passage was only 210 words in length.

Hershberger and Terry [65] showed that signaling core material by using red type increased learning of core concepts without decreasing total amount learned. Foster [77] showed that using all capital letters to signal core material
improved recall of the core material; Foster and Coles [77] extended that result to using boldface type to signal core material. Pratt, Krane, and Kendall [81] showed that italics can provide emphasis to words and phrases in an ambiguous passage. Fowler and Barker [74] showed that special highlighter marking pens, while effective for cueing target items, were no more effective than underlining.

Interestingly, researchers have shown that too many cues can be confusing to the reader. Hershberger and Terry [65] used color and type to distinguish five categories of importance. Their results showed that this "more sophisticated" level of cueing did not increase learning of either the core or enrichment content. Glynn and Di Vesta [79] showed that when two sets of conflicting signals were present, typographic signaling became less effective.

Unfortunately, there have been very few studies that investigated typographic signaling of text structure to aid text comprehension. Designers most often base typographic signaling decisions on what looks effective and what seems to have worked well in the past [Hartley, 86]. We review three studies that investigated typographic signaling to cue text structure.

Britton [86] compared ten United States Army instructional texts. Each text had an original and a revised version. An outside consultant prepared the revised texts independent of Britton's study. The revised versions were considered examples of the "correct" way to present textual material for increased comprehensibility. Britton identified five types of signaling used in the revised versions: (a) levels of subordination signaled by indentation, (b) titles and subheadings, (c) numbering or lettering of paragraphs to indicate the elements of list, (d) different typefaces to show levels of importance, and (e) paragraphing to point out changes in subtopics. He noted that verbal signaling was much less
commonly used in the revisions than typographic signaling. Britton concluded that signaling, especially indentation to show subordination, improved retention of the material.

Frase and Schwartz [79] reported on a study that showed that professionals considered numbering the components of a list and placing them in a vertical alignment as an effective way to revise technical documents. They conducted two experiments that looked at the effect of spatial and segmentation cues on comprehension. First, they broke the paragraphs of a complex technical passage up into meaningful components. Then, they used indentation to highlight the structure of each paragraph's components. Their results showed that segmentation and indentation both produced faster response times for adults verifying information in the passages. However, if the text was meaningfully segmented and each segment appeared on a separate line, then the use of indentation did not significantly affect response time.

Lorch and Chen [86] showed that number signals directed college readers to the sentences they cued and led to better understanding of the signaled information. They also found that the presence of number signals caused their readers to slow down their reading of the signaled information and that readers' order of recall corresponded more closely to the text's organization when the versions read contained number signals.

2.4 Conclusions

In this final section, we cite open areas for research on signaling and suggest ways our research is of interest to text comprehension researchers.
Most of the empirical studies have investigated verbal signaling of relatively short passages. The types and role of signaling in longer passages is not well-understood. Signaling may prove to be an effective cue for readers evaluating material for in-depth study. In particular, signaling may prove to be very effective in helping readers locate material, a sub-task for many real-world comprehension situations.

Identifying different types of typographic signaling and their effectiveness has not received as much attention as that of verbal signaling. We feel that many of the benefits of verbal signaling can be achieved through typographic signaling. For example, listing structures may be effectively signaled by using bullets and vertical alignment instead of the verbal cues of first, next, and finally. Another example is boldfacing which might be more effective for cueing emphasis than pointer words such as most importantly.

Our research is designed to investigate the role of signaling in another domain, that of computer program source code. We feel this is an interesting domain because of the limitations imposed by the programming language syntax and the inherent complexity of the material. In the next chapter, we review research that suggests that some of the signaling techniques developed for aiding readers of textbooks would be equally effective for aiding readers of source code.
Chapter 3

Beacons and Signaling in Computer Programs:

A Review of the Literature

In this chapter, we review the computer science literature related to our research. Chapter 3 is divided into four sections. First, we present an overview of research on cognitive models of program comprehension in order to provide our readers with the context or "big picture" surrounding our research. Next, we discuss in some detail the theory and evidence that most directly bears on our research. Specifically, section 3.2 reviews the literature on the role of beacons in computer comprehension, notably Wiedenbeck [86a, 89]. Section 3.3 reviews the research on using signaling to aid program understanding, notably Oman and Cook [90b] and Baecker and Marcus [90]. Finally, we conclude by citing some open areas for research and connect the specifics of our research with these open areas.

3.1 Cognitive Models of Program Comprehension

Cognitive models of program comprehension are important for predicting and explaining the effects of style on program comprehension. They should provide the theoretical basis for developing style guidelines. Unfortunately, this is
not typically done. Oman [88] concluded that most proposed style guidelines are based on intuition and experience, but without any empirical testing. In this section, we examine the literature on cognitive models of program comprehension that guided our research.

Shneiderman and Mayer [79] list two key components to a cognitive model of program comprehension: the cognitive structures and the cognitive processes. The cognitive structures refer to the hierarchical structures and other types of organizations programmers have or come to have in their long-term memory. The cognitive processes refer to the mental operations involved in using and adding to the cognitive structures in memory.

This section is divided into three parts. First, we discuss the literature on cognitive structures used during program comprehension. Following that, we discuss the literature on cognitive processes used during program comprehension. Finally, we conclude by discussing how the cognitive model research relates to using signaling to aid computer program comprehension.

**3.1.1 Cognitive Structures.**

A programmer uses various types of cognitive structures (or knowledge) during program comprehension. Often the term schemas is used to refer to the general knowledge structures that guide a reader during comprehension. Schemas specify general properties of an object, event, or situation and ignore irrelevant details [Stillings, et al., 87]. Readers use schemas to select and organize text into a meaningful, integrated representation [Mayer, 83].

Researchers has identified two types of schemas involved during text comprehension: content schemas and textual schemas [Armbruster, 86]. Content
schemas store knowledge about specific objects, events, and situations. Readers use their content schemas to recognize and integrate the plot, characters, and setting of the material they are reading. Textual schemas represent knowledge about the conventions of discourse and convey information about the text's structure. Readers use their textual schemas to grasp the structure of the material they are reading.

Content and textual schemas are also used during computer program comprehension. Content schemas represent the generic data structures and operations a programmer has stored in their long-term memory. Content schemas are called plans in computer science literature. Textual schemas represent knowledge about programming language syntax, basic structured programming units such as sequence, iteration and conditional, and conventions of programming. Our review of research on cognitive structures invoked during program comprehension will be organized around these two types of schemas -- content and textural.

**Content Schemas.** Content schemas, or plans, are stereotypical action sequences used in programming [Soloway and Ehrlich, 84]. Examples include summing a series of numbers or swapping two data elements. Programmers learn hundreds, perhaps thousands, of these plans through experience.

Content schemas help programmers reduce a program text into smaller units for understanding. First, they partition the text into plan units based on what the code does. Beacons may provide clues to this partitioning. Next, programmers combine these groups based on global plans. Eventually, programmers form one hierarchical structure that represents the entire program's text.
Pennington [87b] claimed a programmer’s mental representation based on plan knowledge would emphasize function and data flow information about the program. Detailed operations and control flow information about the program would be less accessible.

There exists evidence to support the importance of content schemas during program comprehension. Soloway and Ehrlich [84] showed that programmers used plans to complete missing elements of source code. Adelson [81, 84] concluded that expert programmers organize their mental representations according to plan knowledge. Rist [86] identified several global plans. He showed that programmers used these plans for high-level program understanding.

Content schemas are often thought of as a hierarchical list of attributes or slots. When viewed in this way, beacons represent the dominant attribute in a content schemas’ list of attributes. Recognizing plans can be likened to a process of matching beacons in source code with their corresponding content schemas’ dominant attribute.

**Textual Schemas.** Textual schemas in programming can take on three forms: (a) the programming language syntactic rules, (b) the basic structured programming units such as sequence, iteration and conditional, and (c) the rules concerning the conventions of programming.

Pennington [87b] examined the role of schemas during program comprehension. She referred to the knowledge about the basic structured programming units as text structure knowledge. She concluded that text structure knowledge dominated her experts’ mental representation of computer programs.

She claimed text structure knowledge helped programmers reduce a program text into simpler terms for understanding. According to Pennington,
programmers partition source code into units based on the basic control structures. They use the syntactic rules of the programming language to provide clues to boundaries of these control structure units.

There exists ample evidence that text structure knowledge plays an important role in program comprehension. McKeithen, Reitman, Rueter, and Hirtle [81] conducted two experiments that showed that expert programmers organized their recall of programming concepts based on text structure knowledge. Sheppard, Curtis, Milliman, and Love [79] showed that ill-structured programs were harder to comprehend than well-structured programs. Green [77] showed that the style of the control statements in a program influenced comprehension.

Soloway and Ehrlich [84] studied the influence of rules of programming conventions on comprehension. An example of such a rule is choosing variable names that describe their function in the program. These rules can number in the hundreds or thousands and are learned through experience. They showed that both expert and novice programmers understand programs better when the programs adhere to these rules.

### 3.1.2 Cognitive Processes.

Wiedenbeck [86b] discussed two different cognitive models of program comprehension: bottom-up and top-down. These models differ primarily in their type of cognitive processing. The bottom-up model builds understanding from reading the source code. The top-down model builds comprehension from high-level hypotheses about the program’s functions.
**Bottom-up Processing.** According to bottom-up models of program comprehension, programmers begin by studying the program code and interpreting its meaning. The programmer then pieces together and elaborates these units of understanding to form ever larger units. These units of understanding represent chunks stored in the long-term memory of the programmer. Eventually, the programmer understands the entire program.

Typically, bottom-up processing does not require programmers to work from the lowest level details to gain an overall understanding of the program. They may achieve a high-level understanding of the program even if low-level details are not fully understood. Likewise, a programmer may understand the low-level details without comprehending the high-level actions of the program.

Shneiderman and Mayer [79] proposed the first bottom-up model of program comprehension. According to their model, programmers use their syntactic knowledge of the programming language to form an internal, multilevel semantic representation of the program. Once formed, this semantic representation is resistant to forgetting and accessible to a variety of transformations. On the other hand, the programmer quickly forgets the verbatim program text.

Basili and Mills [82] also viewed comprehension as a bottom-up process. They feel comprehension begins with the reduction of the program to its basic control units. Programmers proceed in a step-by-step manner. First, they work at understanding the basic units and chunking their understanding. Next, they combine these chunks to form new units and chunk their understanding of the new units. Comprehension proceeds in this manner, forming ever larger chunks. Eventually the programmers reach a complete understanding of the program.
**Top-down Processing.** According to top-down models of program comprehension, programmers develop hypotheses about the program even before study of the program text. Initially, programmers develop vague, general hypotheses about the function and major data structures of the program. The programmers base these hypotheses on any clues available. In most cases, just the program name is enough to start the hypotheses rolling. Only when the domain is completely unfamiliar will the programmer delay generating hypotheses until after study of the program text.

Brooks' [83] comprehension model is a top-down, interactive process that requires programmers to form hypotheses, test them, refine them when the tests succeed, and modify them when the tests fail. According to Brooks' model, programmers form an initial hypothesis about the program's function. This leads to subsidiary hypotheses about the program's inputs, outputs, data structures, and operations. These subsidiary hypotheses cause the programmer to expect certain objects or operations in the program text.

Programmers test hypotheses by selectively searching the program text for expected beacons. Finding the beacon strengthens the hypothesis. If the programmer fails to locate the beacon, the hypotheses will need modification.

According to Brooks, a programmer completely understands a program when all the hypotheses are bound to segments of the program. In addition, there should exist no unbound parts remaining in the program.
3.1.3 The Role of Beacons and Signals in the Cognitive Models

Comprehension of large computer programs is a very difficult task. Most hard tasks require people to use a variety of cognitive structures and processes during problem solving. We feel that programmers make use of both content and textual schemas and bottom-up and top-down processing.

Large programs require some top-down processing, if only to cut out parts of the program from an in-depth, bottom-up study. Oman and Cook [90b] observed pruning of code based on module name and/or location in an experiment they ran using experienced programmers and a comprehension task. We feel these programmers made hypotheses about what these parts did and decided that studying them would not contribute to their task at hand.

Soloway and his colleagues have done considerable research on programmers' use of plans during programming; however, the role of beacons in top-down processing has not been well-studied. In the following section, we review the literature on the role of beacons in program comprehension.

When top-down comprehension proves ineffective, Brooks [83] concedes that readers resort to bottom-up processing. In this case, textual schemas, because they are more generic than content schemas, become increasingly important. Researchers in English text comprehension have shown that signaling helps readers activate appropriate textual schemas. In section 3.3 we review the literature on using signaling to cue textual schemas and aid comprehension of computer programs.
3.2 Beacons in Computer Programs

Brooks [83] introduced the notion of beacons, i.e. key features that typically indicate the occurrence of certain data structures or operations. He felt programmers scanned source code and documentation for beacons. Programmers use beacons both for confirming hypotheses previously developed and to suggest new hypotheses.

The term beacon is ill-defined. Brooks gave only one example of a beacon: the three-line swap operation is a beacon for a sorting operation. Figure 3.1 illustrates this beacon inside a sorting routine (the beacon lines are highlighted with boldface type). In addition, he provided a list of sources of beacons (see Table 3.1). Our research has focused only on beacons internal to the program text.

Specific data structures or operations may have multiple beacons and the same beacon may suggest multiple data structures or operations. Brooks suggests that the types of beacons useful during comprehension will vary during the stages of comprehension. During the initial stages, hypotheses will be global and less detailed, hence the most useful beacons will indicate the global structure of the program. As the hypotheses become more detailed, lower level beacons such as variable names and code idioms will be most useful.

Associated with beacons is a confidence level or diagnostic strength. Strongly diagnostic beacons indicate the presence of certain structures or operations with a very high probability; weakly diagnostic beacons may need to occur in combination with other beacons to suggest certain data structures or operations. The swap operation qualifies as a highly diagnostic beacon because it very strongly suggests a sorting operation.
Figure 3.1

Beacon Example: Swap Operation in Sorting Routine

Procedure X (var a : arraytype; n : integer);
    var i, j, incr : integer;
    temp : char;
begin
    incr := n div 2;
    while incr > 0 do begin
        i := incr;
        repeat
            i := i + 1;
            j := i - incr;
            while j > 0 do
                if a[j] > a[j + incr] then begin
                    temp := a[j];
                    a[j] := a[j + incr];
                    a[j + incr] := temp;
                    j := j - incr
                end
            else
                j := 0;
        until i = n;
        incr := incr div 2
    end
end;

Table 3.1

<table>
<thead>
<tr>
<th>Internal to the program text</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Prologue comments</td>
<td>1. User’s manuals</td>
</tr>
<tr>
<td>2. Identifier names</td>
<td>2. Program logic manuals</td>
</tr>
<tr>
<td>3. Variable declarations</td>
<td>3. Flowcharts</td>
</tr>
<tr>
<td>5. Layout</td>
<td>5. Published descriptions</td>
</tr>
<tr>
<td>6. Module structure</td>
<td>of algorithms, etc.</td>
</tr>
<tr>
<td>7. I/O statements</td>
<td></td>
</tr>
</tbody>
</table>
Clearly, as Table 3.1 shows, Brooks did not confine his notion of beacons to code idioms like the swap operation. However, beacons have become more associated with code idioms than with any of the other sources of information. Therefore, we feel a more useful definition of beacons limits them to the code idioms which strongly indicate the presence of a certain operation or data structure. In addition, there are lots of other sources of information that programmers use during top-down study of a program. However, calling all these other sources of information beacons tends to confuse and overload the definition.

Our research investigated two other sources Brooks cited as beacons: identifier names and in-line comments. However, rather than call identifier names and comments beacons, we refer to code idioms as beacons and regard identifier names and in-line comments as other sources of information about a program.

### 3.2.1 Empirical Evidence for Beacons

Little experimental evidence exists regarding beacons. Wiedenbeck [86], using a sorting routine, demonstrated that experienced programmers recalled beacon lines much better than the non-beacon lines. She concluded that her results support the idea that beacons do exist inside source code and serve as focal points for comprehension by experienced programmers.

Her first experiment asked novice, intermediate, and expert programmers to memorize the sort program for later recall. Recall of beacon versus non-beacon lines was not significantly different for her novices and intermediate subjects but was significantly different for her expert subjects. Her experts recalled the beacon lines much better than the non-beacon lines.
Wiedenbeck’s experimental design ruled out potential sources for the differences in recall. Both intermediate and expert subjects recalled non-beacon lines with about the same frequency. If there was something inherent in the test program’s layout or naming convention that favored the recall of the beacon lines, intermediate subjects should also recall the beacon lines more often than non-beacon lines. In actuality, both intermediate and novice subjects recalled non-beacon lines more often than beacon lines. Only the expert subjects recalled the beacon lines more often than non-beacon lines.

Thus, the notion that beacons play an important role in comprehension was only weakly supported by this experiment. Novice, intermediate, and expert subjects did not differ significantly on the identification of the test program as a sort. Indeed, 77% of her novices identified the program as a sort while none of them correctly recalled all the beacon lines.

Her second experiment asked expert programmers to study the sort program for comprehension. After 3 minutes of study, she presented 7 lines from the program and asked her subjects to write the two lines that immediately follow each of the 7 lines. One of the 7 lines presented was the beginning of the swap operation. She found that her experts were significantly better at recalling the lines in the swap operation than any of the other lines. Her subjects also ranked the missing swap lines as the lines they best remembered.

Wiedenbeck [89] completed another set of experiments that demonstrated a causal connection between the presence of the swap beacon and identification of a procedure as a sort. She conducted four experiments that showed the swap beacon is used extensively for comprehension, especially by experienced programmers, and that the swap beacon is so strong it may lead to false comprehension.
Experiment 1 compared comprehension of a standard shellsort procedure with a variation of the shellsort that did not contain the swap beacon. Her subjects performed significantly better at identifying the function of the procedure when the version contained the swap operation. Both procedures were closely matched in length and looping complexity; however, the no-swap version used an odd and inefficient way to get around the swap operation. As Wiedenbeck pointed out, her subjects’ familiarity with the standard shellsort combined with the obvious inefficiency in the no-swap version may have confounded her results.

Experiment 2 was designed to replicate Experiment 1 using less familiar sorting algorithms. An odd-even transposition sort was used as the swap version; a distribution sort was used as the no-swap version. Both procedures were matched in length. There were differences between the procedures in the complexity of the looping structures and the data variables. These differences in complexity may have confounded her results.

Both Experiments 1 and 2 showed that programmers correctly identified a sort procedure significantly more often when the procedure contained the swap beacon. Some of her other results from experiments 1 and 2 were mixed, perhaps because of the confounding factors. Her results did not show whether programmers were more confident about understanding a sort procedure when the swap beacon is present. In addition, her results did not show that programmers were able to recall beacon lines better than non-beacon lines.

Wiedenbeck’s Experiments 3 and 4 demonstrated that the association between the swap beacon and a sort procedure was so strong that it could produce errors in comprehension. Her subjects were mislead into identifying a binary search procedure and two defective sorting procedures as sorts when these procedures contained the swap beacon. She concluded that during the initial study
of a program, a strongly diagnostic beacon, like the swap beacon, was relied on heavily for comprehension. Other features, whether they be missing features that would have normally accompanied the beacon or contradictory features that suggested other operations, were likely to be overlooked.

Wiedenbeck’s experiments imposed a bottom-up comprehension process on her subjects because the programs she presented for comprehension lacked meaningful names, comments, or any other clues with which to form high-level hypotheses about the function of the program. Hence, her results seem to provide limited information about top-down comprehension theory.

Wiedenbeck’s research was centered around the swap beacon in sorting procedures. She found the swap operation to be an exceptionally strong diagnostic beacon and mentioned but did not investigate other beacons or weaker diagnostic beacons that are not uniquely associated with a particular program structure or operation. She suggested future research is needed to identify beacon-like structures, the impact of the context around a beacon, and beacons across a range of programs.

Our research differs from Wiedenbeck’s in two significant ways. First, Wiedenbeck examined the role of code idioms as beacons; we examined the roles identifier names and commenting play as information sources during comprehension. Second, all of Wiedenbeck’s research was done using very short (10-30 line) programs. We used both short and long programs.

### 3.3 Signaling in Computer Programs

The section reviews the literature on the use of signaling to aid program comprehension. While we found references to typographic signaling of source
code, as far as we can tell, we are the first to examine the use of verbal signaling to aid program comprehension.

This section is divided into three parts. First, we review empirical studies that studied the effect of typographic signals on program comprehension. Next, we review Oman and Cook's [90b] book paradigm for formatting source code. Last, we review Baecker and Marcus [90] work on a program visualization tool for C source code.

### 3.3.1 Empirical Studies on Typographic Signals

The typographic signals most commonly seen in source code are (a) indentation and alignment to show nesting levels, (b) blank lines and page breaks to show segmentation, and (c) typefaces and fonts to differentiate between the types of words and statements in the source code.

**Indentation and Alignment.** A consistent indentation and alignment scheme is by far the most common typographic signal used in source code. Programmers either manually or with a prettyprinter program systematically indent source code based on the syntactic structure of the program.

While programmers generally agree that indentation helps readability, results from controlled studies have been mixed. Shneiderman and McKay [75], Weissman [74], and Kesler, *et al.* [84] did not find significant differences in performance between subjects receiving indented and non-indented programs. In contrast, Miara, Musselman, Navaro and Shneiderman [83] and Gilmore and Green [84] showed that indentation can significantly aid comprehension.
Blank Lines and Page Breaks. Ledgard [87] argued that two adjacent program units, such as a procedure or large comment block, should be separated by a page break or at least 3 blank lines. Two adjacent constructs, such as a for loop or single-purpose code fragment, should be separated by at least one blank line. In both cases, vertical layout is used to cue the conceptual units, or chunks, that make up the code.

Unlike indentation, controlled studies investigating the affect of blank lines and page breaks on comprehension are rare. Miara, et al. [83] report on two unpublished studies done by at the University of Maryland in 1980 which included blank lines as one of the independent variables being manipulated. In one of these experiments, blank lines significantly hurt comprehension for undergraduate students. In a follow-up study, blank lines had no significant effect on comprehension for experienced professional programmers.

Typefaces and Fonts. The use of typefaces and fonts to differentiate the types of words and statements in source code is usually limited to boldfacing the reserved words of the programming language. We know of no empirical studies that examined the effect on comprehension of boldfacing the reserved words.

The vgrind utility available with many UNIX systems is an example of a systems that uses typefaces as signals. Vgrind uses italics to signal comments, boldface to signal keywords, and places function names in italics along the right margin of the source code listing.

Oman and Cook’s [90b] book format paradigm and Baecker and Marcus’ [90] visual compiler for the C language are examples of more sophisticated systems which use typefaces and fonts as signals.
3.3.2 Oman and Cook’s Book Format Paradigm

Oman and Cook [90b] presented a set of principles for formatting source code based on program comprehension theory. Their book format paradigm consists of typographic style guidelines that implement their formatting principles. They presented the results of four experiments that demonstrated the book format paradigm significantly improved comprehension of source code.

One of their principles for formatting source code emphasized highlighting beacons. They claimed that code can be made more understandable by highlighting beacons which (a) indicate intermodule control flow and communication, (b) delimit sections such as constants, data declarations and code body, and (c) indicate changes in control flow. As an example, the Const and Var sections in a Pascal program could be delimited by using boldface (or all capitalized letters) for the reserved words, placing them on separate lines, and preceding them with a blank line [Oman, 88]. However, they did not address the problem of how to identify beacons in the source code.

The book format paradigm also makes use of indentation, alignment, blank lines and page breaks to provide spatial cues which indicate grouping and separation. They suggested that:

- Blank lines can separate chunks; alignment and embedded spacing (which includes indentation) can provide the 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 types can be used to highlight important constructs within sentences (in some languages) [Oman and Cook, 90b].

They tested their guidelines using 94-lines of Pascal code and intermediate computer science students as subjects. Highlighted in the code were (a) section headings which were highlighted by using all capital letters for procedure, var,
*begin*, and *end* reserved words and (b) procedure calls which were highlighted by boldfacing the procedure names. Subjects who received the code formatted according to their book format paradigm performed significantly better on a comprehension test than subjects who received a version formatted in a traditional style.

They cannot conclude that using typographic cueing to highlight parts of the code makes it more understandable. Other differences between the two versions of the code may have produced the differences they observed. The book format paradigm version was formatted with control constructs separated by blank lines, statements written as sentences whenever possible, and related clauses aligned together. Any of these other style changes may have made the code easier to understand.

They replicated these results using a short C program and advanced computer science students as subjects. In this experiment, procedure names in the book format paradigm version were highlighted using boldface type. No reserved words were highlighted. As in the first experiment, subjects who received the code formatted according to the book format paradigm performed significantly better on a comprehension test than subjects who received a version formatted in a traditional style. However, the versions also differed in indentation, embedded spacing and alignment, any of which may have been responsible for making the program easier to understand.

The book format paradigm also suggests including a table of contents, an index, chapter divisions, and paginating the source code listing. They performed one experiment that showed that providing a table of contents with a 1,000-line program significantly improved performance on a maintenance task.
Their final experiment used code formatted according to all the book format paradigm guidelines and professional C programmers as subjects. All programmers using the book format paradigm version performed better on a comprehension task than a closely matched programmer using a traditional formatted listing.

The main contribution of Oman and Cook's research is evidence that typographic style can influence comprehension and performance on a maintenance task. Since their book format paradigm includes a number of typographic style guidelines, it is still unclear about the role each of their guidelines plays in program comprehension. There may exist one or two of their guidelines that produce most of the improvement in comprehension while their remaining guidelines may only marginally aid comprehension.

**3.3.3 Baecker and Marcus Visualization Tool for C**

Baecker and Marcus [90] have developed guidelines for formatting C source code based on graphic design principles. In addition, they have implemented SEE, a visual compiler for the C language that automatically formats C source code according to many of their guidelines. To back up their claim that their formatting makes reading, understanding, and using computer programs easier, they presented the results from an experiment they conducted using code formatted both with SEE and with a conventional format.

Their guidelines make extensive use of signaling. For example, (a) three fonts, four typefaces, and seven type sizes are used to differentiate elements in the code; (b) light grey background areas are used to surround and highlight comments; and (c) ruled lines are used as visual separators between file and
function names and the code and comments that define them. They claim that are literally thousands of ways in which their new format differs from a traditional version.

They report on an experiment they conducted that tested their SEE formatting conventions against traditional formatting conventions. For experimental materials they used two short C programs, each formatted both ways. Undergraduate computer science students served as subjects and their score on a comprehension quiz was used as the dependent measure. Only one of the programs was significantly easier for their subjects to comprehend when it was formatted in the SEE format. With the other program, subjects scored higher on the traditionally formatted version. Baecker and Marcus concluded that SEE was an improvement but recommended more controlled experimentation.

### 3.4 Conclusions

In this final section, we first cite open areas for research in the use signaling to aid program comprehension and connect the specifics of our research with these open areas.

Identifying and highlighting beacons in source code is a fundamental principle in Oman and Cook’s book format paradigm. Other than Wiedenbeck’s work on the swap operation, which serves as a beacon to a sorting operation, research on identifying beacons in source code does not exist. Oman and Cook suggest some reserved words and module names may serve are beacons and these are the words they highlighted in their examples of the book format paradigm. However, because of confounding factors, they cannot conclude that highlighting these reserved words and module names improved comprehension.
Our research is designed to show that meaningful identifier names and in-line comments are important sources of information during comprehension. This would verify another important class in Brooks' table of beacons. We feel that as programs become larger, programmers use meaningful identifier names and in-line comments to aid comprehension much more than they use code idioms.

The use of signals in source code is becoming possible because of the increased availability of high-resolution output devices such as laser printers. We feel sophisticated program visualization tools such as the prototype versions of Oman and Cook's book paradigm formatter and Baecker and Marcus' SEE compiler will become increasingly available and an important research topic. Up until now, research has focused on answering the question: Does signaling in source code aid program comprehension? Both the intuitive answer and the empirically verified answer is: yes!

The time is ripe to begin studying the effect of specific types of signals on comprehension. We feel a good point to start is with the research question: does typographic signaling of module names and comments make programs easier to understand? Answering questions of this type require carefully designed controlled experimentation in which the confounding factors are eliminated or minimized.

Finally, research on the effectiveness of verbal signaling from expository text research should be extended to source code listings. We feel that properly chosen module names and in-line comments may function as preview statements. Our research is designed to investigate preview statements in source code as an aid to comprehension.
Chapter 4

A Framework for Program Comprehension Research

The methodology used in many controlled experiments on program comprehension is often criticized [Brooks, 80; Sheil, 81; Moher and Schneider, 82]. Either the subjects are too inexperienced, the programs are too small, or the tasks are irrelevant to real world programming. This chapter first examines the methodological problems characteristic of controlled studies of program comprehension. Next, it describes a framework for examining program comprehension. This framework is based on the notion that comprehension does not occur within a vacuum; rather it emerges from the interplay of a programmer’s motivation, strategy, and results. Finally, the chapter concludes with suggestions on how researchers may use this framework to create realistic scenarios for experimental studies.

4.1 Problems with Methodology

Brooks’ [80] criticisms of methodology focused on three areas of concern: (a) choice of subjects, (b) choice of materials, and (c) choice of dependent variable. Clearly all three areas are linked. It makes no sense to pair novice programmers up with a 10,000-line program and ask them to recall as many lines
of the program as possible. A description of common problems for each of these three areas follows.

**4.1.1 Choice of Subjects**

When choosing subjects for use in an experiment, a researcher must satisfy two objectives: (a) selecting subjects who truly reflect the overall characteristics the population of interest and (b) including enough subjects in the sample to reflect the diversity among members of the target population.

Table 4.1 summarizes the population of interest for 32 controlled experiments on program comprehension which asked subjects to view source code listings. Quite often, the population of interest was not explicitly stated in the published report. In these cases, if the published report's research goals and conclusions related to professional programmers, then professional programmers were assumed to represent the target population.

Frequently, these controlled experiments have looked at the differences between expert and novice programmers. Expert subjects have usually been selected from advanced computer science students, novice subjects from beginning computer science students. Ideally, the results obtained from the expert subjects should generalize to professional programmers with considerable experience. However, in many cases, this generalization is questionable given the level of expertise of the subjects.

Programmers, professional programmers, expert, and novice programmers are all vague terms. This lack of specificity of the target population makes generalization of experimental results to a real world population difficult. If
### Table 4.1
Summary of Subject Selection for Controlled Experiments on Program Comprehension

<table>
<thead>
<tr>
<th>Researcher</th>
<th>Target Population</th>
<th>Sample Population</th>
<th># Subjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adelson, 81</td>
<td>expert programmers</td>
<td>teaching fellows</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>novice programmers</td>
<td>undergrads with 1 programming course</td>
<td>5</td>
</tr>
<tr>
<td>Adelson, 84</td>
<td>expert programmers</td>
<td>teaching fellows</td>
<td>18</td>
</tr>
<tr>
<td>Experiment 1</td>
<td>novice programmers</td>
<td>undergrads with 1 programming course</td>
<td>18</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>expert programmers</td>
<td>teaching fellows</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>novice programmers</td>
<td>undergrads with one programming course</td>
<td>24</td>
</tr>
<tr>
<td>Bateson, et. al., 87</td>
<td>expert programmers</td>
<td>college students with more than 3 programming courses</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>novice programmers</td>
<td>college students with less than 3 programming courses</td>
<td>20</td>
</tr>
<tr>
<td>Crosby and Stelovsky, 90</td>
<td>expert programmers</td>
<td>8 graduate students, 1 PhD</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>novice programmers</td>
<td>undergrads with 2 programming courses</td>
<td>10</td>
</tr>
<tr>
<td>Gilmore and Green, 1984</td>
<td>programmers</td>
<td>college students (all non-programmers)</td>
<td>40</td>
</tr>
<tr>
<td>Gould, 75</td>
<td>professional programmers (debuggers)</td>
<td>professional programmers (at least 4 years experience)</td>
<td>10</td>
</tr>
<tr>
<td>Harrison and Cook, 86</td>
<td>professional programmers</td>
<td>undergrads</td>
<td>148</td>
</tr>
<tr>
<td>McKeithen, et al., 1981</td>
<td>expert programmers</td>
<td>university teachers</td>
<td>6</td>
</tr>
<tr>
<td>Experiment 1</td>
<td>intermediate programmers</td>
<td>undergrads, 1 ALGOL course</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>novice programmers</td>
<td>undergrads, starting first ALGOL course</td>
<td>24</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>expert programmers</td>
<td>same as before</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>intermediate programmers</td>
<td>same as before</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>novice programmers</td>
<td>same as before</td>
<td>8</td>
</tr>
</tbody>
</table>
Table 4.1 (Cont.)

Summary of Subject Selection for Controlled Experiments on Program Comprehension

<table>
<thead>
<tr>
<th>Researcher</th>
<th>Target Population</th>
<th>Sample Population</th>
<th># Subjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miara, et al., 83</td>
<td>expert programmers</td>
<td>3 or more years programming in school more and/or than 2 years professional experience less than 3 years programming in school and less than 2 years professional experience</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>novice programmers</td>
<td></td>
<td>54</td>
</tr>
<tr>
<td>Oman and Cook, 90b</td>
<td>professional programmers (enhancement)</td>
<td>senior/grad CS students</td>
<td>53</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>professional programmers</td>
<td>all had over 6 years prof. experience</td>
<td>12</td>
</tr>
<tr>
<td>Pennington, 87b</td>
<td>professional programmers</td>
<td>all had over 3 years prof. experience</td>
<td>80</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>professional programmers</td>
<td>same as before</td>
<td>40</td>
</tr>
<tr>
<td>Rist, 1986</td>
<td>expert programmers</td>
<td>third year CS grad students enrolled in intro. prog. course</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>novice programmers</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>Schmidt, 1986</td>
<td>programmers</td>
<td>undergrad and graduate students</td>
<td>18</td>
</tr>
<tr>
<td>Sheppard, et al., 1979</td>
<td>professional programmers</td>
<td>professional programmers</td>
<td>36</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>professional programmers</td>
<td>professional programmers</td>
<td>36</td>
</tr>
<tr>
<td>Experiment 3</td>
<td>professional programmers</td>
<td>professional programmers</td>
<td>54</td>
</tr>
<tr>
<td>Shneiderman, 77</td>
<td>programmers</td>
<td>college students enrolled in introductory programming course</td>
<td>62</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>programmers</td>
<td>college students enrolled in introductory programming course</td>
<td>48</td>
</tr>
</tbody>
</table>
Table 4.1 (Cont.)

Summary of Subject Selection for Controlled Experiments on Program Comprehension

<table>
<thead>
<tr>
<th>Researcher</th>
<th>Target Population</th>
<th>Sample Population</th>
<th># Subjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shneiderman, 82</td>
<td>professional programmers</td>
<td>undergraduate CS students</td>
<td>57</td>
</tr>
<tr>
<td>Experiment 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Experiment 2</td>
<td></td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>Soloway and Ehrlich, 84</td>
<td>advanced programmers</td>
<td>at least 3 programming courses</td>
<td>45</td>
</tr>
<tr>
<td>Experiment 1</td>
<td>novice programmers</td>
<td>less than 3 programming courses</td>
<td>94</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>professional programmers</td>
<td>professional programmers</td>
<td>41</td>
</tr>
<tr>
<td>Vessey, 1987</td>
<td>professional programmers</td>
<td>professional programmers</td>
<td>16</td>
</tr>
<tr>
<td>Experiment 1</td>
<td>expert programmers</td>
<td>professional programmers</td>
<td>38</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>novice programmers</td>
<td>students enrolled in programming course</td>
<td>42</td>
</tr>
<tr>
<td>Weiser, 82</td>
<td>professional programmers</td>
<td>graduate teaching assistants and computing center staff</td>
<td>21</td>
</tr>
<tr>
<td>Wiedenbeck, 1986</td>
<td>experienced programmers</td>
<td>grad students and faculty</td>
<td>12</td>
</tr>
<tr>
<td>Experiment 1</td>
<td>intermediate programmers</td>
<td>junior level students</td>
<td>12</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>novice programmers</td>
<td>enrolled in first programming course</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>experienced programmers</td>
<td>grad students and faculty</td>
<td>12</td>
</tr>
<tr>
<td>Woodfied, et al., 81</td>
<td>professional programmers</td>
<td>grad and upper-level students</td>
<td>48</td>
</tr>
</tbody>
</table>
the researchers had been more explicit and selective in stating their target populations, then their conclusions could be more critically evaluated.

Even though the population of interest in most comprehension studies has been professional programmers, researchers have frequently used sample populations consisting of college students enrolled in computer programming classes (see Table 4.1). A likely reason for this is convenience. A survey of the empirical research in software maintenance published between the years 1978 and 1987 showed that academic researchers conducted more than 82% of the published controlled studies [Hale and Haworth, 88]. Since these academic researchers are at universities, the students in their programming classes were readily available for use as subjects. Indeed, Hale and Haworth found that 57% of the published articles used only students as the sample population. There is no reason to believe that results produced from experimental studies using student programmers generalize to conclusions applicable to professional programmers.

The second objective when selecting a sample is including enough subjects in the sample to reflect the diversity among members of the target population. In general, the bigger the sample the better. If the sample size is too small, statistical tests are less likely to produce significant results and/or a few outliers or a non-normal distribution may invalidate the statistical tests chosen.

Professional programmers are a fairly diverse group [Moher and Schneider, 82]. Even if professional programmers were more homogeneous in experience and training, studies have shown that programmer productivity can differ by as much as 25 to 1 across experienced programmers with equivalent backgrounds [Sackman, 70]. Given this diversity, large sample sizes are desirable.

Most comprehension experiments have used modest sample sizes (see Table 4.1). Often, they have also reduced the variability among subjects by
restricting the sample population to members of the same programming class or college. Therefore, many research conclusions must by scrutinized due to a small sample population which is not truly representative of both the overall characteristics and the diversity of the target population.

In summary, the methodology of many controlled experiments in program comprehension may be criticized for generalizing the results produced from a small group of college students to professional programmers. Researchers' conclusions would be stronger if they carefully considered what target population they wanted to generalize their results to and then selected subjects which are representative of this target population.

4.1.2 Choice of Materials

When choosing materials for use in an experiment, researchers must select materials which produce a result within the limitations of the experimental setting while at the same time being representative of the real world problem under investigation. Table 4.2 summarizes the materials used in the same 32 controlled experiments on program comprehension which asked subjects to view source code listings.

Table 4.2 shows that most comprehension studies have used programs of about 100 lines or less. This is often due to the limitations imposed by the experimental setting. These limitations include

(a) time -- longer programs require longer times for the subjects to complete the experimental tasks,

(b) subject selection -- materials suitable to both novice and expert subjects are limited to those suitable to novices,
Table 4.2

Summary of Choice of Materials for Controlled Experiments on Program Comprehension

<table>
<thead>
<tr>
<th>Researcher</th>
<th>Materials</th>
<th>Experimental Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adelson, 81</td>
<td>16 lines of PPL code</td>
<td>verbatim recall</td>
</tr>
<tr>
<td>Adelson, 84 Experiment 1</td>
<td>eight PPL programs (example shown is 11-lines) and two types of flowcharts</td>
<td>comprehension quiz score and time</td>
</tr>
<tr>
<td>Adelson, 84 Experiment 2</td>
<td>eight PPL programs (slightly longer than Experiment 1) and two types of flowcharts</td>
<td>Same as Experiment 1</td>
</tr>
<tr>
<td>Bateson, et. al., 87 Syntactic test errors</td>
<td>four FORTRAN programs (20 to 21-lines each)</td>
<td>verbatim recall (minor syntax allowed)</td>
</tr>
<tr>
<td>Crosby and Stelovsky, 90</td>
<td>10-lines Pascal code</td>
<td>cloze procedure eye fixation</td>
</tr>
<tr>
<td>Gilmore and Greenm 1984</td>
<td>15-line BASIC-like program</td>
<td>comprehension quiz score and time</td>
</tr>
<tr>
<td>Gould, 75</td>
<td>four FORTRAN programs (84, 139, 124, and 98-lines each)</td>
<td>time to debug # of bugs found # subject errors</td>
</tr>
<tr>
<td>Harrison and Cook, 86</td>
<td>two Pascal programs 130 and 100-lines each</td>
<td>comprehension quiz score</td>
</tr>
<tr>
<td>McKeithen, et al., 1981 Experiment 1</td>
<td>31-line ALGOL program</td>
<td>verbatim recall recall order</td>
</tr>
<tr>
<td>McKeithen, et al., 1981 Experiment 2</td>
<td>same as Experiment 1 but in random order</td>
<td></td>
</tr>
<tr>
<td>Miara, et al., 83</td>
<td>102-line Pascal program</td>
<td>comprehension quiz score subjective rating of difficulty</td>
</tr>
</tbody>
</table>
### Table 4.2 (Cont.)

Summary of Material Selection for Controlled Experiments on Program Comprehension

<table>
<thead>
<tr>
<th>Researcher</th>
<th>Materials</th>
<th>Experimental Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oman and Cook, 90b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Experiment 1</td>
<td>1000-line Pascal program</td>
<td>enhancement time and correctness</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>1057-line C program</td>
<td>comprehension quiz score and time construct call graph time and correctness</td>
</tr>
<tr>
<td>Pennington, 8b7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Experiment 1</td>
<td>eight program segments (15-lines each, FORTRAN and COBOL)</td>
<td>comprehension quiz score recognition time</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>200-line program (FORTRAN and COBOL)</td>
<td>comprehension quiz score</td>
</tr>
<tr>
<td>Rist, 1986</td>
<td>twelve Pascal programs (22 to 42-line of code each)</td>
<td>cluster analysis of &quot;chunking&quot; by subjects</td>
</tr>
<tr>
<td>Schmidt, 1986</td>
<td>32-line PL/1 program</td>
<td>reading time</td>
</tr>
<tr>
<td></td>
<td>32-lines PL/1 statements (random)</td>
<td>verbatim recall recognition (accuracy)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>comprehension quiz score</td>
</tr>
<tr>
<td></td>
<td></td>
<td>verbatim recall</td>
</tr>
<tr>
<td>Sheppard, et al., 1979</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Experiment 1</td>
<td>nine FORTRAN programs (26 to 57-lines each)</td>
<td>functional recall</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>three FORTRAN programs (39 to 56-lines each)</td>
<td>modification time and accuracy</td>
</tr>
<tr>
<td>Experiment 3</td>
<td>three FORTRAN programs (25 to 225 lines each)</td>
<td>time to find bug</td>
</tr>
<tr>
<td>Shneiderman, 77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Experiment 1</td>
<td>26-line FORTRAN program</td>
<td>modification task (grade)</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>67-line COBOL program</td>
<td>verbatim recall (# lines attempted, # lines correct)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>comprehension quiz score</td>
</tr>
<tr>
<td></td>
<td></td>
<td>recall (# lines attempted, # lines verbatim correct, # lines functionally correct)</td>
</tr>
</tbody>
</table>
Table 4.2 (Cont.)

Summary of Material Selection for Controlled Experiments on Program Comprehension

<table>
<thead>
<tr>
<th>Researcher</th>
<th>Materials</th>
<th>Experimental Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shneiderman, 82</td>
<td>223-line Pascal program and pseudocode or data structure diagram</td>
<td>comprehension quiz score</td>
</tr>
<tr>
<td>Experiment 1</td>
<td>223-line Pascal program and program design aid</td>
<td>comprehension quiz score</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>223-line Pascal program and program design aid</td>
<td>comprehension quiz score</td>
</tr>
<tr>
<td>Soloway and Ehrlich, 84</td>
<td>eight Pascal programs (13 to 17-lines each)</td>
<td>cloze procedure</td>
</tr>
<tr>
<td>Experiment 1</td>
<td>six ALGOL programs (13 to 17-lines each)</td>
<td>verbatim recall</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>six ALGOL programs (13 to 17-lines each)</td>
<td>verbatim recall</td>
</tr>
<tr>
<td>Vessey, 1987</td>
<td>67-line COBOL program</td>
<td>functional recall</td>
</tr>
<tr>
<td>Experiment 1</td>
<td>same as Experiment 1 (four versions)</td>
<td>functional recall</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>same as Experiment 1 (four versions)</td>
<td>functional recall</td>
</tr>
<tr>
<td>Weiser, 82</td>
<td>three ALGOL programs (75 to 150-lines each)</td>
<td>time to locate bug recognition (accuracy)</td>
</tr>
<tr>
<td>Wiedenbeck, 1986</td>
<td>23-line Pascal program</td>
<td>verbatim recall</td>
</tr>
<tr>
<td>Experiment 1</td>
<td>23-line Pascal program</td>
<td>verbatim recall</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>23-line Pascal program</td>
<td>verbatim recall</td>
</tr>
<tr>
<td>Woodfied, et al., 81</td>
<td>111-line FORTRAN program four versions</td>
<td>comprehension quiz score</td>
</tr>
</tbody>
</table>
(c) task selection -- many tasks such as recall are possible only on small programs, and
(d) equivalence among conditions -- when different programs are used, showing that 2 or more large programs are comparable in all significant respects except the independent variable(s) being manipulated is a very difficult task [Brooks, 80].

Again, there is no reason to believe that results from the small programs generalize to larger programs. In fact, the effort required to write and maintain programs seems to increases exponentially with the size of the program [Brooks, 83].

4.1.3 Choice of Dependent Variable

Finally, the dependent variables (DV$s$) used in controlled studies of program comprehension have been criticized as having little relevance to real world programming tasks. Table 4.2 summarizes the DV$s$ used in the 32 controlled experiments on program comprehension.

Over one-third of the studies used recall as the dependent variable. While Shneiderman [77] showed that recall correlated well with comprehension quiz scores, Vessey [87] showed that recall was not a good indicator of debugging performance. Moreover, recall does not directly correspond to any real world programming task nor is it applicable as a DV with larger programs.

The studies used the subjects' score on a comprehension quiz as the dependent variable over one-third of the time. Comprehension is a recognized subtask of most programming activities. Nanja and Cook [87] showed a correlation between comprehension and superior performance on a debugging task.
However, construction and grading of an unbiased and representative comprehension quiz becomes very hard when the program is large.

In summary, recall and comprehension quiz scores, while easy to administer and grade with small programs, do not make good DVs when the programs used in the experiment exceed 30 lines of code.

### 4.1.4 Conclusion

External validity measures how well results from an experiment generalize to new situations. Experiments have low external validity if the circumstances of the experiment bear little relationship to the real world problem they are investigating [Saslow, 82]. Many of the controlled studies on program comprehension suffer from low external validity. Their external validity may be strengthened by researchers being more selective in the target population, choosing more representative and larger samples, having the subjects work with larger programs, and selecting dependent variables which more directly resemble real world tasks.

### 4.2 A Framework for Program Comprehension Research

#### 4.2.1 Introduction

This section proposes a framework for discussing the program comprehension process (see Table 4.3). This framework is similar to one proposed by Waller [77] for text comprehension. The goal of establishing this framework is to define some of the circumstances surrounding specific programming tasks which
have program comprehension as a sub-component. This allows researchers to be more precise about the real world problems under investigation and leads to the design of controlled experiments with high external validity.

Table 4.3
A Framework for Program Comprehension

<table>
<thead>
<tr>
<th>Reader/Goal</th>
<th>Strategy</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re-engineering task</td>
<td></td>
<td>Change the Program</td>
</tr>
<tr>
<td>corrective</td>
<td>Objective</td>
<td>add code</td>
</tr>
<tr>
<td>adaptive</td>
<td>high-level understanding</td>
<td>remove code</td>
</tr>
<tr>
<td>perfective</td>
<td>location</td>
<td>revise the code</td>
</tr>
<tr>
<td>Re-use of code</td>
<td>in-depth understanding</td>
<td>restructure the code</td>
</tr>
<tr>
<td>Education</td>
<td>critique</td>
<td>modify documentation</td>
</tr>
<tr>
<td>language</td>
<td>confirm hypothesis</td>
<td></td>
</tr>
<tr>
<td>syntax/semantics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>algorithm/data structure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program style</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Anticipation</td>
<td>Reading style</td>
<td>Re-use of Code</td>
</tr>
<tr>
<td>Administration</td>
<td>browse</td>
<td>Gain Knowledge</td>
</tr>
<tr>
<td>Evaluation</td>
<td>skim/preview</td>
<td>language</td>
</tr>
<tr>
<td>quality assurance</td>
<td>search</td>
<td>syntax/semantics</td>
</tr>
<tr>
<td>programmer expertise</td>
<td>intense study</td>
<td>algorithm/ data structure</td>
</tr>
<tr>
<td></td>
<td>review</td>
<td>program style</td>
</tr>
<tr>
<td></td>
<td>Environment</td>
<td>assess quality</td>
</tr>
<tr>
<td></td>
<td>hardcopy listings</td>
<td>Confirm/refute hypotheses</td>
</tr>
<tr>
<td></td>
<td>on-line listings</td>
<td></td>
</tr>
<tr>
<td></td>
<td>support tools</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cross reference listing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>flow/structure charts</td>
<td></td>
</tr>
<tr>
<td></td>
<td>data flow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>debuggers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>human experts</td>
<td></td>
</tr>
<tr>
<td></td>
<td>other documentation</td>
<td></td>
</tr>
</tbody>
</table>
Given this framework, factors which influence comprehension can now be said to be helpful in certain situations and harmful in others. This coincides with the notion there is no one ideal program style for improving comprehension; rather the appropriate style depends, at least in part, on the motivations and tasks surrounding the comprehension process.

The notion behind the framework is simple. Goals motivate someone to understand a program; they choose a strategy; and hopefully, they achieve some outcome. Program comprehension will not continue satisfactorily if there is a breakdown or change in any one of these three areas.

Rather than being static and sequential in nature, the goals, strategy, and outcome of the comprehension process are dynamic and interactive. For example, a programmer assigned a corrective re-engineering task might adopt a strategy of quickly skimming the program to develop an overall understanding of it and then searching the program to find the most likely location of the error. The desired outcome is to revise the code and correct the error. However, if the program is written in an unfamiliar style, skimming it may prove impossible. First, the programmer will need to educate him/herself on the style through a combination of browsing and intense study of the source code. Only when the programmer successfully learns the programming style can a return to the strategies desired for the corrective re-engineering task occur.

This section discusses some common goals, strategies, and outcomes which accompany program comprehension. Our objective is not to present a comprehensive framework which describes the situations surrounding every program comprehension activity. Rather, by being more explicit about common comprehension situations, the framework may serve to guide the design of
controlled experiments which investigate program comprehension in real world situations.

4.2.2 Goals

The first column in the framework for program comprehension focuses on identifying readers of source code and their goals. The most common task associated with program comprehension is maintenance programmers performing some re-engineering task. However, there are other readers of source code beside maintenance programmers. They include

(a) Development programmers reading code in order to re-use some component of it,
(b) Programmers reading code to learn a new language, algorithm, or programming style,
(c) Programmers reading code to familiarize themselves with it because they anticipate doing something with it in the future,
(d) Clerks reading code to help label and archive the software,
(e) Professional programmers reading code to help them evaluate it during code walk-throughs and inspections,
(f) Program testers reading code to identify boundary conditions for error testing, and
(g) Managers and teachers reading code to help them evaluate the competence of the code’s author.

Discussion on each of these comprehension goals follows. These goals include most of the situations surrounding program comprehension. There are
undoubtedly other goals associated with program comprehension. However, their frequency of occurrence is relatively low.

**Re-engineering task.** The need to perform some re-engineering task is probably the most common motivating factor for programmers picking up an unfamiliar program and attempting to understand it. Both Fjeldstad and Hamlen [79] and Lientz and Swanson [81] found that the data processing organizations they surveyed devoted over 50% of their resources to maintaining existing systems and that the maintenance programmers spent over 23% of their time just studying source code.

Re-engineering may be partitioned into three activities: corrective, adaptive, and perfective. Corrective re-engineering fixes defects in the program. Adaptive re-engineering modifies the program to meet the evolving hardware and software environment. Perfective re-engineering attempts to respond to user requests or to improve the quality of the software. Examples of perfective re-engineering include restructuring the code to make it more maintainable, creating and updating the documentation, and fine-tuning the system to speed up execution time. Lientz and Swanson [81] reported that the data processing organizations they surveyed devoted about 17% of the maintenance resources to corrective changes, 28% to adaptive changes, and 52% to perfective changes.

The characteristics of maintenance programmers run the gamut from computer science graduates fresh out of college to old-time professionals with little or no formal education in computer science. Rarely is re-engineering taught as part of the computer science curriculum.

**Re-use of code.** Some of the biggest gains in software productivity are possible through the reuse of existing software. In many cases, reusing software
begins as a comprehension task. The programmer must first understand the existing software before assessing its reusability.

Currently, reuse of software means reuse of the functionality of code through abstract interfaces and information hiding. Weiser 87] argued that source code is also a source of things to reuse. A programmer can reuse variable names, variable naming schemes, generic code, modular structure, algorithms, indentation styles, and performance or storage tricks from source code listings.

Re-use is most often associated with development programmers. However, managers also make re-use decisions. When a program’s specifications and environment have changed substantially, managers may decide that re-use of the existing code may be more trouble than simply starting over from scratch.

**Education.** One of the ways people learn to program is through reading and studying source code examples. Pick up any programming text and there will be short segments of code scattered throughout. The reader’s objective may be to learn a new programming language, a new algorithm, or a new programming style.

While most people normally associate students with reading code to learn from it, even experienced professional programmers can learn from browsing source code. Weiser 87] argued that access to source code is an essential ingredient of a good programming environment. Cedar, Interlisp-D, Zetalisp, and Smalltalk are examples of programming environments characterized by a high degree of easily accessible source code.

**Anticipation.** Programmers may want to study a program’s source code because they expect to do something with it in the future. In this case, they may read the code without a clear objective or task in mind. Rather, they just want to familiarize themselves with the program enough so if, in the future, an opportunity
arises to work with or use some idea from the program, they will have benefited from this study time.

**Administration.** Brooks 82] argues for the importance of a program clerk as a member of the software team. The program clerk acts as a librarian for the program's code and documentation. Program clerks will read code the same way librarians read incoming books. They want to familiarize themselves with the software and classify it correctly so they can locate and lend it out in the future. They may also perform some quality control checks by ensuring that the software conforms to company standards. Program clerks need not be proficient in programming. Instead they may rely exclusively on names and documentation.

**Evaluation.** Code walk-throughs and inspections play an important role in early defect removal and quality assurance of the software. During these walk-throughs and inspections, typically five or six members of the software development team review a section of code written by another programmer. Soloway, Pinto, Letovsky, Littman, and Lampert 88] found that most of the discussion during an IBM code inspection they analyzed centered on making the code and documentation easier to understand for future readers, making sure the code performed according to its specifications, and maintaining consistency with the details of the code and its corresponding documentation.

Program testers read code to help them identify the boundary conditions, branch conditions, and termination conditions to include in the test cases.

While managers seldom read code to help them evaluate a programmer's competence, they should read code to assess its maintainability, robustness, and overall quality rather than studying it to determine its functionality.
Finally, teachers read their student's code in order to assess its quality. Like managers, they should evaluate the code's maintainability, robustness, and overall quality as well as its correctness.

4.2.3 Strategy

The middle column in the framework for program comprehension focuses on the code reader's strategy. Readers select a strategy to accomplish their goals. The strategy involves refining their goals into objectives and then choosing a reading style which interacts with the environment in order to accomplish these objectives. Discussion of the objectives, reading styles, and environment follows.

Objective. Readers develop objectives based on their goals. For example, many re-engineering tasks decompose into three objectives: (a) first obtain a high-level understanding of the structure of the program, (b) next locate where something is done, and (c) then obtain an in-depth understanding of the pertinent small section of the program which will be changed.

Top-down comprehension models assert that a common objective during program comprehension is to confirm hypotheses about the program. According to the model, while confirming hypotheses may require in-depth understanding, more often, hypotheses are confirmed by locating beacons in the program.

Evaluation goals decompose into a variety of critiquing objectives. Code may be critiqued for correctness, clarity, maintainability, consistency, etc.. While some of these critiquing objectives require an in-depth understanding, others can be assessed from the surface features of the code.
**Reading Style.** Code readers select a reading style to accomplish their objectives. There are many ways of reading programs. They may be browsed, skimmed, previewed, searched, studied intensively, or reviewed. Rarely are they read like a novel, i.e. consecutively from the beginning to the end.

The reading style involves both perceptual and cerebral processing. Scanning the code for cues is as important as understanding the statements. Scanning the code relies heavily on the reader's expectations and the presence of beacons and signals in the code which serve as cues.

**Environment.** The available environment for program comprehension may include a hard-copy listing of the source code, an on-line listing of the source code, maintenance support tools, other people familiar with the code, language or environment, and external documentation for the program. Comprehenders make use of one or more of these in order to understand the program. The scope of this thesis is environments which consist solely of hard-copy listings.

### 4.2.4 Outcome

The third column in the framework for program comprehension focuses on the outcome resulting from the program comprehension. Possible outcomes include changing the program, re-using some part of the program, gaining knowledge, and receiving pleasure.

**Change the program.** Since re-engineering tasks rank as the most common motivator for studying programs, changing the program to accomplish the re-engineering task is a common outcome. The programmer may add code,
remove code, revise the code, restructure the code, and/or modify the documentation.

**Re-use of Code.** Programmers may be able to re-use some (or all) of the program as the result of their study. The re-use may take the form of a simple copy and paste operation, learning a way of doing something (algorithm or data structure) but which requires re-coding, or adopting some element of the program's style, naming convention, etc..

**Gain Knowledge.** Gaining knowledge almost always results from reading programs. The programmer may learn about a new algorithm, language, or programming style and store this new knowledge in memory for later retrieval. Often, programmers find it is easier to remember the general concept and the location in the source code of the details instead of remembering all of the details.

In addition, readers form opinions about the quality of the program and the competence of the programmer(s) who wrote it. In this case, they may not store any of the details of the program in memory. Rather, they store their assessment of the quality and competence.

Finally, programs are read to confirm hypotheses. Again in this case, readers may store little of the details of the program. Rather, they store only the confirmation or refutation of the hypotheses.

**Receive Pleasure.** Knuth 74] argues eloquently in his Turing Award lecture that writing a program can be like composing great poetry or music. At least for Knuth, reading "literate" programs can be a thrill.
4.3 Using the Framework in Controlled Studies

4.3.1 Introduction

The section describes how the framework for program comprehension may be useful in designing controlled experiments on program comprehension. We based our suggestions on two basic assumptions. First, research in program comprehension is context sensitive. Experimental results depend on the subjects, their orienting instructions, the materials, and the criterial tasks chosen for the experiment. Second, research in program comprehension should be designed with real world situations in mind.

Jenkins [76] proposed a tetrahedral model for experiments in memory and learning. His model emphasized the interdependencies between the nature of the subjects, their orienting instructions, the materials employed, and the criterial tasks they perform. He felt that interaction between these variables is the norm, not the exception. Therefore, researchers should frame their conclusions within the context of the experiment.

Figure 4.1 applies Jenkins' tetrahedral model to experiments on program comprehension. The framework for program comprehension is integrated with Jenkins' model to provide a real world orientation to the experimental design. By grounding subject selection, orienting tasks, materials, and criterial tasks in real world comprehension situations, researchers can design experiments which resemble, as close as possible, the conditions to which they wish to generalize. Discussion on each of the four vertices of the tetrahedral follows.
Figure 4.1
The Tetrahedral Model Applied to Program Comprehension Experiments

SUBJECTS
- maintenance programmers
- development programmers
- students
- managers
- clerks
- teachers
- testers

ORIENTING TASKS
- goals
- objectives
- reading styles

MATERIALS
- hardcopy listings
- on-line listings
- software tools
- human experts
- other documentation

CRITERIAL TASKS
- changes to program
- re-use of code
- gain knowledge
- evaluate
- confirm hypothesis
4.3.2 Subjects

Maintenance programmers, development programmers, students, managers, clerks, and teachers all read source code. Therefore they all represent target populations for experiments on program comprehension. Choosing subjects randomly from the target population may not select a representative sample or even be feasible. Therefore the task becomes choosing the most representative sample available.

Researchers must carefully consider the differences between the sample and target population when generalizing their results. By being more specific in the target population under investigation, researchers are better able to evaluate the differences between the target and sample populations and to estimate the degree to which these differences weaken their conclusions.

Researchers should chose subjects appropriate to the orienting tasks, materials, and criterial tasks used in their experiment. For example, they often choose students as subjects. The students may be appropriate subjects if they are instructed to read a small program in a new programming language or style and then to add or revise some code. This situation is similar to what they have already encountered in their programming classes.

However, these same students would be inappropriate as subjects if the orienting task were to prepare for a code inspection, or the materials included a 10,000-line program, or their criterial task was to assess the program's quality or perform a re-engineering task. Quite simply, most students do not have experience working in these situations so their performance will not generalize to target populations of professional programmers or managers.
Many interesting studies have been done looking at differences between novice and expert programmers. Figure 4.1 suggests that studies should also look at differences between the target populations. For example, it is quite possible that factors which improve comprehension for maintenance programmers may hurt comprehension for students, managers, or development programmers.

4.3.3 Orienting Tasks

Orienting tasks refers to the task the researcher assigns to the subjects at the start of the experiment. Orienting tasks such as "study this program for comprehension" are ineffective because they are too vague and open to interpretation. As a consequence, subjects perform this task in too many different ways.

Figure 4.1 suggests that researchers should select orienting tasks from the framework's goals, objectives, and reading styles. This will add realism to the experimental setting and induce a mental set among the subjects which is similar to the real world situation under investigation.

Pennington [87a] used the following orienting task:

"Programmers were instructed that they were to make a modification to a program normally maintained by another programmer. However the "other programmer" was going on vacation and the modification was urgent. The subjects' task was then to become familiar with the program and to make the changes to it. Furthermore the hypothetical other programmer had left the program with the subject to study and would return in 45 min to explain the modification task."

She reported that her subjects accepted this scenario as realistic and meaningful.

Oman and Cook [90b] used the same orienting task. The framework for program comprehension would label this scenario as anticipatory since the programmer
reads the code anticipating working with it in the future but without a clearly-defined task at hand.

Figure 4.1 suggests that reading styles and objectives may help frame the orienting task. Instructions to subjects to skim a program to develop a high-level understanding of its structure or to confirm a specific hypothesis will reduce variation among subjects due to their different interpretations of the instructions.

Orienting tasks may differ from the criterial task. For example, experimenters may instruct subjects to evaluate the quality of the code for the next 30 minutes in preparation for a code-inspection. At the end of the 30-minute study, subjects may take a comprehension quiz and their scores used as the dependent variable.

Interesting experiments may be designed which study the effects of orienting tasks on comprehension. For example, subjects instructed according to Pennington's anticipatory scenario may comprehend a program differently than subjects who are given the re-engineering task specifications before the 30-minute study. If this is the case, then results generated from experiments using Pennington's scenario may not generalize to re-engineering situations since maintenance programmers generally know the task specifications before their program study.

4.3.4 Materials

The materials used in experiments on program comprehension will have a profound effect on the generalizability of the results. For high external validity, materials used in the experiment should be representative of real world environments encountered in program comprehension situations. However, most
studies have used programs consisting of 100 lines or less. The real world situation most closely resembling this is students learning to program from textbook examples. Results produced from experiments using 100-line programs do not necessarily generalize to comprehension situations where longer programs are more typically involved.

The point is often made that the small programs used in experiments correspond to routines inside of larger programs. For example, if a re-engineering task is confined to a routine, then the argument is made that results obtained from using just the routine should generalize to the re-engineering task under investigation. However, in this case, the results generalize to an objective contained within the re-engineering task situation, i.e. in-depth understanding of a small section of the program. While this can yield useful information, it is important to realize that other key objectives required by the re-engineering task such as high-level understanding of the program’s structure and location of the parts of the program affected by the changes are missing from the experimental situation.

The small programs used in experiments designed to generalize to maintenance situations also differ from the real world environment due to the self-contained nature of small programs. When maintaining large systems, programmers work with small sections which interact with a much bigger system. Rather than attempt to understand the entire system, they make hypotheses based on names, documentation, and previous experience. We suggest that code used in experimental studies intended to generalize to maintenance situations not be self-contained.

Small programs still have a role in program comprehension experiments. In many real world situations, small programs represent the actual environment.
For example, students study small sections of code to learn new concepts and code walk-throughs generally review only a short section of code.

### 4.3.5 Criterial Tasks

Researchers use criterial tasks to produce the dependent variable which measures comprehension. The criterial tasks most commonly used are recall and comprehension quizzes. Recall and quizzes are useful for measuring knowledge gained: a real world outcome. Therefore, results using these criterial tasks may help predict behavior in situations where the readers’ goal was education and the desired outcome was to gain detailed knowledge about the program’s code. However, the results may not directly generalize to conclusions about re-engineering tasks or re-use of code.

Figure 4.1 suggests basing criterial tasks on the framework for program comprehension. The advantage is that results would directly generalize to real world outcomes. The disadvantage is that, in many cases, more than one thing is being measured. For example, if the criterial task is to add code to the program and the dependent variables are correctness of the code added and the time it took to add the code, both the programmer’s comprehension and code writing ability are being tested.

Researchers may also choose criterial tasks which measure attainment of comprehension objectives rather than goals. The advantage to measuring objectives attained rather than goals is that the objectives are more focused. The disadvantage is that generalizing results from objectives becomes more questionable. For example, using location as the criterial task and time as the
dependent variable, it may be possible to show that finding a routine is faster for the experimental condition than the control condition. However, the conclusion that the experimental condition makes re-engineering tasks quicker is questionable because the experimental condition may have introduced other factors which make changes to the program more difficult.

4.4 Conclusions

The framework for program comprehension proposed in this chapter is helpful in designing controlled experiments on program comprehension and in generalizing their results to real world comprehension situations.

The framework helps in designing experiments by adding precision to the situations surrounding real world program comprehension. This precision permits the design of controlled experiments with high external validity in the following ways:

(a) Researchers may base subject selection on the framework. This will lead to better defined, more homogeneous target populations. Traditionally, target populations for controlled studies have been unspecified or ill-defined.

(b) Researchers may select materials based on the framework. This will lead to materials that are focused and representative of the real world situation being investigated. For most comprehension studies, larger programs are more appropriate than programs of 100 lines or less.

(c) Researchers may select orienting tasks based on the framework that are realistic and familiar to the subjects.
(d) Researchers may develop criterial tasks based on the framework that correspond with the goals and objectives of the real world comprehension situation under investigation.

The framework for program comprehension helps generalize research results to conclusions about the real world. By adding precision to real world comprehension situations, researchers may evaluate the differences between the experimental setting and the real world situation under investigation. They can then estimate the degree to which these differences weaken the conclusions of their study.
Chapter 5

Experiments 1 and 2:

Using Meaningful Identifier Names to Aid Comprehension

In this chapter, we report on two controlled experiments we conducted investigating the role identifier names play in program comprehension. Since we view this work as an extension to the research on beacons conducted by Wiedenbeck [86, 89], we designed the experiments to correspond to her design.

This chapter is divided into three sections. The first section reports on Experiment 1. In this experiment, we investigated both procedure and variable names as sources of information during comprehension. The second section of this chapter reports on Experiment 2. We conducted Experiment 2 as a follow-up to our first experiment and investigated the role different variable names play during comprehension. We end this chapter with a conclusion section which restates our major findings.

5.1 Experiment 1

The purpose of Experiment 1 was twofold: (a) to determine if meaningful identifier names aid comprehension; and (b) more importantly, to compare the
relative diagnostic strength of meaningful names and code idiom beacons (like the swap operation). Comprehension was viewed as the identification of the high-level function of a procedure. The scope of the comprehension effort was limited, like Wiedenbeck's experiments, to short procedures. All our subjects were advanced programmers.

Brooks [83] suggested that meaningful identifier names were logical candidates for beacons during comprehension. Clearly, the right name indicates a particular structure and/or operation. The name pop, for example, suggests both a stack data structure and the operation that removes the first element. We expected programmers to rely very heavily on meaningful procedure names during comprehension, perhaps even more so than strong code idiom beacons like the swap operation. We expected programmers to rely also on meaningful variable names during comprehension, especially when the names highlight key code sections in the program.

5.1.1 Method

A between-subject, completely randomized experiment was run. Each subject was given the task of understanding one of four versions of two short Pascal programs.

Program 1 was a binary search procedure. The independent variables associated with this program were (a) type of procedure name and (b) type of variable name. In both cases, the names had two levels: meaningful and neutral. The dependent variables were (a) correct identification of the procedure's function and (b) subjects' confidence ratings of the understanding of the procedure's function.
Program 2 was a sort procedure. The independent variables associated with this program were (a) type of procedure name and (b) presence of the swap operation. The procedure name had two levels: neutral or misleading. The swap operation was either present or not present in the procedure. The dependent variables were (a) correct identification of the procedure’s function and (b) subjects’ confidence ratings of the understanding of the procedure’s function.

Subjects. Ninety-six students were recruited from a senior/graduate-level (52 subjects) and two graduate-level (44 subjects) computer science courses at Oregon State University to serve as subjects. The experiment was conducted during regular classroom hours. Participation was voluntary; five students opted not to participate or misunderstood the instructions and were not included in the data.

Half of the subjects had one or more years of professional programming experience; they knew, on the average, more than 5 programming languages; and had attended, on the average, 3.4 years of undergraduate and graduate-level computer science classes.

Materials. Source code listings in Pascal for all four versions of both the binary search and sort procedures were prepared (see Appendices A and B respectively). We chose Pascal as the source code language because it is the programming language most widely known by students at Oregon State University.

The four versions of the binary search procedure differed by two factors: (a) the type of procedure name (meaningful, neutral) and (b) the type of variable name (meaningful, neutral). Two versions used the meaningful procedure name search; the other two used the neutral procedure name x. We felt an important code section for the binary search procedure was the line that calculated the middle
index into the array being searched, $v := (t + s) \div 2$. To draw attention to this line, we substituted the variable name *middle* for $v$; the control condition used the variable name $v$.

The four versions of the sort procedure differed by two factors: (a) the type of procedure name (neutral, misleading) and (b) the presence of the swap beacon (present, not present). Two versions used the neutral procedure name $y$; the other two used the misleading procedure name *shuffle*. We chose the misleading name *shuffle*, because we felt the name suggested some sort of randomizing operation, which could be implemented with a swap operation. Two types of sort routines were used; an insertion sort and an odd-even transposition sort. We chose the insertion sort because it is a well known sort that does not contain a swap operation. We chose the odd-even transposition sort because it is less well known, contained a swap operation, and was used by Wiedenbeck [4]. Both sort procedures were matched in length and complexity.

Each subject was given a five-page packet of materials, stapled together. Page one consisted of background questions, which the subjects answered before the experiment began. Pages two and four were the binary search and sort procedures respectively; pages three and five were blank to be filled in by the subjects with their written descriptions of the procedures and their confidence ratings.

All instructions were given orally. Overhead transparencies that contained the same instructions were displayed along with the oral instructions.

**Procedure.** Packets were distributed to subjects in random order. Subjects were told that they would have one minute to study the Pascal procedure that appeared on page two. They were told to try to understand as well as possible
what the procedure did (its function). At the end of the minute, they were to fold the procedure over so they could no longer see it. Next, they were given two minutes to provide a written description of the procedure they had just studied. They were instructed to describe the procedure's function and to be as specific as possible. Following that they were asked to rate on a scale from 1 to 6 how confident they were about their description of the procedure's function, with 1 being *not confident* and 6 being *extremely confident*. Subjects were given one minute to complete the confidence rating task. The exact same set of instructions and procedure followed for the sort procedure. That is, first, subjects studied a version of the sort procedure for one minute; next, the subjects wrote a description of the procedure's function; lastly, the subjects rated their confidence of their written description.

### 5.1.2 Results

Table 5.1 gives the percentage of subjects who correctly identified the function for each of the binary search procedures. The subjects' descriptions were considered correct if the description contained the words *binary search*. Overall, 35% of the subjects identified their version as a binary search.

A Chi square test was used to analyze differences in comprehensibility between the procedure versions (Table 5.2). Subjects did not perform equally well on all versions (*p* = .027). Although versions 1, 2, and 3 did not differ significantly in difficulty (*p* = .542); version 4, which contained neither the meaningful variable name nor the meaningful procedure name, was significantly harder for the subjects to comprehend than the other three versions (*p* = .005).
Table 5.1
Percentage of Subjects Correctly Identifying the Binary Search Procedure, Experiment 1

<table>
<thead>
<tr>
<th>Procedure Name</th>
<th>Meaningful</th>
<th>Neutral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meaningful</td>
<td>52%</td>
<td>38%</td>
</tr>
<tr>
<td>Variable Name</td>
<td>n=23</td>
<td>n=26</td>
</tr>
<tr>
<td>Version 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neutral</td>
<td>38%</td>
<td>10%</td>
</tr>
<tr>
<td>Version 3</td>
<td>n=26</td>
<td>n=21</td>
</tr>
<tr>
<td>Version 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2
Chi Square Tests of Differences: Binary Search Procedures, Experiment 1

<table>
<thead>
<tr>
<th>Versions tested</th>
<th>$\chi^2$</th>
<th>df</th>
<th>N</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effect of Version</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 vs. 2 vs. 3 vs. 4</td>
<td>9.19</td>
<td>3</td>
<td>96</td>
<td>.027</td>
</tr>
<tr>
<td>1 vs. 2 vs. 3</td>
<td>1.23</td>
<td>2</td>
<td>75</td>
<td>.542</td>
</tr>
<tr>
<td>(1 + 2 + 3) vs. 4</td>
<td>7.87</td>
<td>1</td>
<td>96</td>
<td>.005</td>
</tr>
<tr>
<td>Effect of Meaningful Variable Name</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 vs. 4</td>
<td>5.12</td>
<td>1</td>
<td>47</td>
<td>.024</td>
</tr>
<tr>
<td>1 vs. 3</td>
<td>0.93</td>
<td>1</td>
<td>49</td>
<td>.336</td>
</tr>
<tr>
<td>(1 + 2) vs. (3 + 4)</td>
<td>3.93</td>
<td>1</td>
<td>96</td>
<td>.047</td>
</tr>
<tr>
<td>Effect of Meaningful Procedure Name</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 vs. 4</td>
<td>5.12</td>
<td>1</td>
<td>47</td>
<td>.024</td>
</tr>
<tr>
<td>1 vs. 2</td>
<td>0.93</td>
<td>1</td>
<td>49</td>
<td>.336</td>
</tr>
<tr>
<td>(1 + 3) vs. (2 + 4)</td>
<td>3.93</td>
<td>1</td>
<td>96</td>
<td>.047</td>
</tr>
</tbody>
</table>

The analysis of the data showed that the meaningful variable name significantly aided subjects' comprehension only for versions of the procedure that contained the neutral procedure name ($p=.024$). If the version contained the meaningful procedure name, then there is no evidence from this data to show that
the meaningful variable name significantly affected the subjects' comprehension 
\((p= .336)\).

The data also showed that the meaningful procedure name significantly 
aided comprehension only for versions of the procedure that did not contain the 
meaningful variable name \((p= .024)\). If the version contained the meaningful 
variable name, then there is no evidence from this data to show that the meaningful 
procedure name significantly affected the subjects' comprehension \((p= .336)\).

Table 5.3 gives the subjects' mean confidence ratings for each of the four 
vversions of the binary search procedure. Five of the subjects did not include a 
confidence rating for their description of the procedure's function and were not 
included in the data. There is no evidence from this data to show that the presence 
of either the meaningful procedure name or the meaningful variable name 
significantly affected the subjects' confidence rating, \(F(3,87)=1.62, p=.190\).

Table 5.3

Mean Confidence Ratings: Binary Search Procedures, 
Experiment 1

<table>
<thead>
<tr>
<th>Version</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rating</td>
<td>4.3</td>
<td>3.3</td>
<td>3.4</td>
<td>3.3</td>
</tr>
<tr>
<td>n</td>
<td>22</td>
<td>23</td>
<td>25</td>
<td>21</td>
</tr>
</tbody>
</table>
Sort. Table 5.4 gives the percentage of subjects correctly identifying the function for each of the sort procedures. The subjects' descriptions were considered correct if the description contained the word sort. Overall, 59% of the subjects identified their version as a sort.

Table 5.4

Percentage of Subjects Correctly Identifying the Sort Procedure, Experiment 1

<table>
<thead>
<tr>
<th>Procedure Name</th>
<th>Neutral</th>
<th>Misleading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swap Code Idiom</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Swap</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Swap Code Idiom</th>
<th>74%</th>
<th>68%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version 1</td>
<td>61%</td>
<td></td>
</tr>
<tr>
<td>n=23</td>
<td></td>
<td>n=25</td>
</tr>
<tr>
<td></td>
<td>36%</td>
<td></td>
</tr>
<tr>
<td>Version 4</td>
<td></td>
<td>n=25</td>
</tr>
</tbody>
</table>

A Chi square test was used to analyze differences between the procedure versions (Table 5.5). Subjects did not perform equally well on all versions ($p=.037$). Although versions 1, 2, and 3 did not differ significantly in difficulty ($p=.639$); version 4, which contained no swap beacon and had a misleading procedure name, was significantly harder to comprehend than the other three versions ($p=.006$).
Table 5.5

Chi Square Tests of Differences: Sort Procedures, Experiment 1

<table>
<thead>
<tr>
<th>Versions tested</th>
<th>$\chi^2$</th>
<th>df</th>
<th>N</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Effect of Version</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 vs. 2 vs. 3 vs. 4</td>
<td>8.45</td>
<td>3</td>
<td>96</td>
<td>.037</td>
</tr>
<tr>
<td>1 vs. 2 vs. 3</td>
<td>0.90</td>
<td>2</td>
<td>75</td>
<td>.639</td>
</tr>
<tr>
<td>(1 + 2 + 3) vs. 4</td>
<td>7.66</td>
<td>1</td>
<td>96</td>
<td>.006</td>
</tr>
<tr>
<td><strong>Effect of Swap Beacon</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 vs. 4</td>
<td>5.13</td>
<td>1</td>
<td>50</td>
<td>.024</td>
</tr>
<tr>
<td>1 vs. 3</td>
<td>0.89</td>
<td>1</td>
<td>46</td>
<td>.345</td>
</tr>
<tr>
<td>(1 + 2) vs. (3 + 4)</td>
<td>5.23</td>
<td>1</td>
<td>96</td>
<td>.020</td>
</tr>
<tr>
<td><strong>Effect of Misleading Procedure Name</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 vs. 4</td>
<td>2.97</td>
<td>1</td>
<td>48</td>
<td>.085</td>
</tr>
<tr>
<td>1 vs. 2</td>
<td>0.20</td>
<td>1</td>
<td>48</td>
<td>.653</td>
</tr>
<tr>
<td>(1 + 3) vs. (2 + 4)</td>
<td>2.35</td>
<td>1</td>
<td>96</td>
<td>.125</td>
</tr>
</tbody>
</table>

The analysis of the data showed that the swap beacon significantly aided comprehension only for versions that contained the misleading procedure name ($p = .024$). If the version contained the neutral procedure name, then there is no evidence from this data to show that the presence of the swap beacon significantly affected comprehension ($p = .345$).

The data produced no evidence to support the claim that the misleading procedure name significantly hurt comprehension. The misleading procedure name may have had some effect if the version did not contain the swap beacon ($p = .085$). However, if the swap beacon was present, then subjects performed about equally well on versions with either the neutral or misleading procedure name ($p = .653$).

Table 5.6 gives the mean of the subjects’ confidence ratings for each of the four versions of the swap procedure. Three of the subjects did not include a confidence rating for their comprehension of the sort procedure and were not included in the data. There is no evidence that the presence of the misleading
procedure name or the swap beacon significantly affected the subjects' confidence rating, $F(3,89)=0.14$, $p=.939$.

Table 5.6
Mean Confidence Ratings: Sort Procedures, Experiment 1

<table>
<thead>
<tr>
<th>Function</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rating</td>
<td>3.6</td>
<td>3.6</td>
<td>3.8</td>
<td>3.6</td>
</tr>
<tr>
<td>n</td>
<td>23</td>
<td>24</td>
<td>23</td>
<td>23</td>
</tr>
</tbody>
</table>

5.1.3 Discussion

The results showed that both meaningful procedure and variable names can aid comprehension. No conclusions can be drawn about differences in diagnostic strength of procedure names versus variable names; they appeared to carry about the same strength. However, a strong code idiom beacon, like the swap operation, was more influential than the procedure name, at least for our small programs.

While it may seem obvious that meaningful names should improve comprehension, previous research results have been mixed. Many studies have found either no effect or inconclusive results for the effectiveness of meaningful names at improving program comprehension [Shneiderman, 80; Sheppard, Curtis, Milliman and Love, 79]. Still, most books on programming style include the use of meaningful names as one of their guidelines [Ledgard and Tauer, 87; Kernighan and Plauger, 74].

Wiedenbeck [86] showed that all lines in a procedure do not carry the same diagnostic weight, rather that beacon lines serve as a focal point for study.
We suspect that identifier names exhibit this same property. Procedure names are, in general, highly diagnostic of the procedure’s function and hence programmers have learned to rely on them during comprehension. Variable names that draw attention to and clarify key sections of code should also aid comprehension. However, we feel that meaningful variable names that highlight non-critical sections of code may actually decrease high-level comprehension by drawing attention to them and away from code idiom beacons in the code.

While we had expected procedure names to be as highly diagnostic as the swap operation, our results did not confirm this. Very few of our subjects were fooled by the misleading procedure name *shuffle* when the sort procedure contained the swap operation. As with Wiedenbeck [89], the swap operation was so strong a beacon that subjects either ignored or disregarded conflicting information. On the other hand, when the sorting procedure did not contain the swap operation, a large number of subjects were fooled by the misleading procedure name.

Our results did not replicate Wiedenbeck’s [89], which showed a causal connection between the presence of a swap operation and superior identification of a procedure as a sort. When our two sorting procedures were presented to our subjects with neutral procedure names, the swap version was not significantly easier for them to identify than the no-swap version. Our failure to replicate Wiedenbeck’s is due to the differences in the no-swap sorting algorithms used in the two experiments. She used a distribution sort, a sort that features no comparisons or nested loops. We used an insertion sort, a familiar sort with nested loops and comparisons. Both experiments used the odd-even transposition sort for the swap version. We had hoped to show that even when the no-swap version was routine and familiar, the swap operation was so strong a beacon that a significance difference in high-level comprehension would still emerge.
We did show that in the presence of conflicting information, such as our misleading procedure name *shuffle*, the swap operation significantly aided high-level comprehension. Wiedenbeck’s distribution sort may be viewed as a misleading sort because it contains neither comparisons nor nested loops.

We compared the performance of our subjects identifying the binary search procedure as a search (includes those identifying it as a binary search) with Wiedenbeck’s [89]. She used a similar binary search procedure (neutral names) with her control groups in Experiment 3. After three minutes of study, 53% of her advanced programmers identified the procedure as a search. After one minute of study, only 24% of our subjects receiving version 4 (neutral procedure name, neutral variable name) of the binary search procedure identified it as a search. Apparently, the binary search procedure does not contain a sufficiently strong code idiom beacon for the majority of the subjects to quickly identify it as a search. The extra study time Wiedenbeck gave her subjects may account for their improved performance.

This suggests that in situations where there are no obvious beacons to aid comprehension, increasing the study time is an effective way to improve high-level comprehension of small procedures. Bottom-up comprehension theory would predict this. Given more time, programmers can study, understand, and integrate more of the code thus improving their high-level comprehension.

When we provided subjects with version 2 (neutral procedure name, meaningful variable name) of the binary search procedure, they were able to slightly outperform Wiedenbeck’s advanced programmers in one-third the study time. After one minute of study, 54% of our subjects were able to identify the procedure as a search. We conclude that an effective variable name aids
comprehension and significantly reduces the study time required for high-level comprehension of small procedures.

In other words, providing extra study time when strongly diagnostic information is present in the code may contribute little to improving high-level comprehension. Results from the sort procedure seem to back this up. Considering our subjects who received version 1 (neutral procedure name, swap operation) of the sort procedure, 74% of them were able to identify the procedure as a sort after only one minute of study. After three minutes of study, 84% of Wiedenbeck's [89] advanced programmers in her Experiment 2 identified the same procedure as a sort.

We expected nearly all our subjects receiving a version of the binary search procedure containing the procedure name search to identify the procedure as a search. Surprisingly, this was not the case. Only 65% of the subjects receiving version 3 (procedure name search, neutral variable name) identified the procedure as a search; only 78% of the subjects receiving version 1 (procedure name search, meaningful variable name) identified it as a search.

Sheppard, Curtis, Milliman, and Love [79] found an upper limit to the amount of information their programmers needed to perform a comprehension task on short programs. They concluded that aids beyond this limit did not produce improved performance. We observed this in our experiment as well. Providing programmers with both the meaningful procedure and variable names did not significantly improve comprehension compared to versions that contained only one of the two meaningful names.

The diagnostic strength of code idiom beacons and meaningful names may be a function of the size of the comprehension task. When presented with very few lines of code to study, programmers may find a bottom-up comprehension
process to be as effective as a top-down comprehension process. In small programs, the code idioms and names embedded in the procedure body may be easily and quickly located and used to guide high-level comprehension.

We expect the diagnostic strength of meaningful procedure names to be stronger when the procedure occurs inside larger programs. With larger programs, given limited time, a top-down comprehension process will be more efficient than a bottom-up process for high-level comprehension. Therefore, we expect programmers to use procedure names to suggest and confirm hypotheses to a much greater extent and to use code idiom beacons and variable names embedded inside the procedure bodies to a much lesser extent. The reason for this is that procedure names are easily and predictably located in the code due to the programming language syntax and formatting conventions. However, code idiom beacons, like the swap, and meaningful variable names must be located by scanning the entire listing. Therefore, using procedure names during comprehension saves time.

As Brooks [83] suggests, the diagnostic strength of beacons will vary during different stages of comprehension. We looked at only the problem of comprehending the high-level function of a procedure in isolation. We examined two cases: (a) the degenerate special case in which there is no information available to the programmer with which to form initial high-level hypotheses about the procedure’s function without examining the code in the body of the procedure and hence comprehension must begin as a bottom-up process; and (b) the case when a meaningful procedure name was available to the programmers for use in forming initial high-level hypotheses about the procedure’s function and hence comprehension may begin as a top-down process.

Our subjects studying a version of the procedures that used a neutral procedure name were faced with the degenerate special case. Our results,
combined with Wiedenbeck's, suggested in the absence of meaningful names or
code idiom beacon, high-level comprehension was a function of study time. However, when the procedure body contained a meaningful name, high-level comprehension was achieved much faster. We conclude that names can form the basis for bottom-up study and that high-level comprehension can be achieved even before all the low-level details are understood.

By supplying some of the versions of our procedures with meaningful procedure names, we had hoped to induce a top-down comprehension process and observe a dramatic improvement in high-level comprehension compared with the degenerate special case. This did not prove to be the case. It may be that with small procedures a bottom-up process is as effective as a top-down process for high-level comprehension. Another explanation may be that even though a procedure name was available with which to form high-level hypotheses and initiate a top-down process, since the procedure was small and the comprehension task was not well-defined, subjects used a bottom-up process anyway.

5.2 Experiment 2

Experiment 1 demonstrated that meaningful variable names can draw attention to and clarify key sections of source code. The results showed that when the version of the binary search procedure contained the variable name middle, subjects were able to identify its function as a binary search significantly more often than when the version contained all neutral, one-letter variable names. In this case, the variable name middle served as a strongly diagnostic source of information in the binary search procedure.
We feel that not all meaningful variable names carry the same diagnostic strength. In fact, meaningful variable names that highlight non-critical sections of code may actually hurt high-level comprehension by drawing attention away from the actual code idiom beacons in the program.

Experiment 2 was designed to test the diagnostic strength of several meaningful variable names contained in the binary search procedure. We felt that the variable name middle was highly diagnostic since its name suggests the key idea behind the binary search operation: divide and conquer. We felt the variable names first and last would suggest an array operation, and perhaps searching, but not necessarily a binary search operation. Finally, the control version contained only neutral, one-letter variable names. Based on the previous experiment, we expected this version would be the hardest for our subjects to comprehend since none of its variable names carried any meaning.

5.2.1 Method

A between-subject, completely randomized design experiment was run. The independent variable was the variable names contained in the procedure. The dependent variable was whether or not the subject correctly identified the procedure as a binary search.

Subjects. One-hundred and one students were recruited from a senior-level computer science course at Oregon State University to serve as subjects. The experiment was held during regular classroom hours and participation was voluntary.

Eighty-two of the subjects were male; 17 were female (2 subjects sex unknown). One of the subjects was a sophomore, 14 were juniors, 57 were
seniors, and 25 were graduate students (4 subjects class standing unknown). Their average age was 24.4 years.

Over one-third of the subjects had professional programming experience. The subjects knew, on the average, more than five programming languages and had taken, on the average, 2.7 years of undergraduate and graduate-level computer science classes.

*Materials.* Source code listings in Pascal for all three versions of the binary search procedure were prepared (see Appendix D). We chose Pascal as the source code language because it is the programming language most widely known by students at Oregon State University.

The three versions of the binary search procedure differed only by the variable names contained in the procedure. Version 1 contained only neutral, one-letter variable names throughout. This was the control condition, since its variable names contained no information to help the subject comprehend the procedure's overall function. This version is exactly the same as the binary search procedure Version 4 used in Experiment 1.

Version 2 and 3 were the experimental conditions. We felt a code idiom beacon for the binary search procedure was the line that calculated the middle index into the array being searched, \( v := (t + s) \div 2 \). To highlight this beacon, we substituted the variable name `middle` for \( v \) in Version 3. This version is exactly the same as the binary search procedure Version 3 in Experiment 1.

Version 2 was designed to suggest search, but not necessarily binary search to our subjects. To accomplish this, we substituted the variable names `first` for \( t \) and `last` for \( s \). Version 2 did not contain the variable name `middle`; thus we
felt its variable names were not as strongly diagnostic as the variable names in Version 3.

Each subject was given a three page packet of materials, stapled together. Page one consisted of background questions, which the subjects answered before the experiment began (see Appendix C). Page two contained one of the three versions of the binary search procedure. Page three was blank and was filled in by the subjects with their written descriptions of the procedure’s function.

All instructions were given orally. Overhead transparencies that contained the same instructions were displayed along with the oral instructions.

**Procedure.** Packets were distributed to subjects in random order. Subjects were told that they would have one minute to study the Pascal procedure that appeared on page two. They were told to try to understand as well as possible what the procedure did (its function). At the end of the minute, they were to fold the procedure over so they could no longer see it. Next, they were given two minutes to provide a written description of the procedure they had just studied. They were instructed to describe the procedure’s function and to be as specific as possible.

### 5.2.2 Results

Table 5.7 gives the percentage of subjects who correctly identified the function for each of the versions of the binary search procedure. The subjects’ descriptions were considered correct if the description contained the words binary search. Overall, 33% of the subjects identified their version as a binary search.
Table 5.7
Percentage of Subjects Correctly Identifying the Binary Search Procedure, Experiment 2

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Neutral</th>
<th>First, Last</th>
<th>Middle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>17%</td>
<td>40%</td>
<td>42%</td>
</tr>
<tr>
<td></td>
<td>n=35</td>
<td>n=35</td>
<td>n=31</td>
</tr>
<tr>
<td>Version 1</td>
<td></td>
<td>Version 2</td>
<td>Version 3</td>
</tr>
</tbody>
</table>

A Chi square test was used to analyze differences in comprehensibility between the procedure versions (see Table 5.8). Subjects did not perform equally well on all versions ($p=.052$). Although versions 2 and 3 did not differ significantly in difficulty ($p=.873$); version 1, which contained no meaningful variable names, was significantly harder for the subjects to comprehend than the other two versions ($p=.015$).

Table 5.8
Chi Square Tests of Differences: Binary Search Procedures, Experiment 2

<table>
<thead>
<tr>
<th>Effect of Version</th>
<th>$\chi^2$</th>
<th>df</th>
<th>N</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 vs. 2 vs. 3</td>
<td>5.90</td>
<td>2</td>
<td>101</td>
<td>.052</td>
</tr>
<tr>
<td>1 vs. 2</td>
<td>4.48</td>
<td>1</td>
<td>70</td>
<td>.034</td>
</tr>
<tr>
<td>1 vs. 3</td>
<td>4.93</td>
<td>1</td>
<td>66</td>
<td>.026</td>
</tr>
<tr>
<td>2 vs. 3</td>
<td>0.03</td>
<td>1</td>
<td>66</td>
<td>.873</td>
</tr>
<tr>
<td>1 vs (2 + 3)</td>
<td>5.87</td>
<td>1</td>
<td>101</td>
<td>.015</td>
</tr>
</tbody>
</table>

5.2.3 Discussion

The results showed that meaningful variable names in the binary search procedure helped our subjects' high-level comprehension. However, no
conclusions can be drawn about the differences in diagnostic strength between the variable names *middle* versus *first, last*. Our subjects understood Versions 2 and 3, which contained the meaningful variable names, about equally well.

The similarity in diagnostic strength between the variable names *middle* and *first, last* may be due to the fact that both names highlighted the statement line that calculated the midpoint of the section of the array being searched: \[ v := (t + s) \div 2; \]
Version 2 replaces this line with \[ v := (first + last) \div 2; \]
Version 3 replaces this line with \[ middle := (t + s) \div 2; \] We consider this line to be the key code idiom of the binary search routine. Since both Versions 2 and 3 effectively draw attention to this line, and no differences in our subjects’ comprehension were observed, this suggests that drawing attention to the key line may be as important as assigning meaningful names to the variables.

The variable names *first* and *last* occur 8 times in the body of the binary search procedure; *middle* occurs only 6 times. Since *first, last* occur more often, this may have confounded the results. In addition, the operation \((first + last) \div 2\) suggests calculating the midpoint of the array and therefore also suggests the key concept behind the binary search operation: divide and conquer.

We might have produced the results we expected if we had chosen the variable *b* rather than *s* and *t* for assignment of a meaningful name. If *b* is assigned the name *found*, then this should suggest a search operation but not necessarily a binary search because it does not highlight the critical line \[ v := (t + s) \div 2; \]
One confounding variable if *b* is used and compared with *middle* is that *b* occurs 4 times in the body of the binary search procedure while *middle* occurs 6 times.
5.3 Conclusions

Both these experiments showed that the meaningful identifier names we chose aided high-level comprehension of small programs. While we found no significant differences in relative diagnostic strength between the procedure and variable names we used, we expect programmers to rely more heavily on procedure names than on variable names as the size of the program increases. In addition, while we found that programmers relied more heavily on the swap beacon than the procedure name during comprehension of a small program, we expect programmers to rely more heavily on procedure names than on code idiom beacons as the size of the program increases.
Chapter 6

Experiment 3:

Does Signaling Aid Professional Programmers?

In this chapter we report on Experiment 3, a controlled experiment we conducted that investigated ways signaling may be used to aid computer program comprehension. This experiment had professional programmers study a large C program and then respond to a series of questions about the program.

Since program comprehension is a critical subtask of so many programming activities, it has been frequently investigated. Unfortunately, these investigations have produced few suggestions about ways to make programs easier for experienced programmers to understand. As a consequence, most researchers agree that future investigations should study professional programmers trying to understand larger programs.

Experiment 3 investigated three types of signals which may make source code listings easier for professional programmers to understand. This is an important contribution because of the size of the problem and because we studied professional programmers. Millions of lines of source code are maintained yearly.
Therefore, ways of making code easier to read and understand can yield enormous benefits in terms of lowering personnel costs and improving the quality of the maintenance work performed.

Experiment 3 differed from previous work in program comprehension by drawing on research from text comprehension studies that investigated signaling as an aid to text comprehension. While signaling has been studied by text comprehension researchers since the 1970's, as far as we know, no one has applied this research on signaling to produce more readable source code listings.

The remainder of the chapter is divided into four sections: (a) introduction, (b) method, (c) results, and (d) discussion. Since much of the motivation for this study and a review of the literature has been covered in the first four chapters of this thesis, the introduction section is directed towards discussing the types of signaling under investigation. Chapter 7 presents guidelines for using signaling to aid program comprehension based on the results of Experiment 3.

6.1 Introduction

Experiment 3 investigated three types of signals we thought would aid comprehension of source code listings. They were typographic signals, preview statements, and headings. Discussion on each of these types follows.
6.1.1 Typographic Signals

Typographic signals are physical cues, like boldface type and ruled lines, which are added to text to aid readers during comprehension. Researchers are just beginning to propose and investigate ways to take advantage of laser printers to add typographic signals to source code listings. Baecker and Marcus [90] have offered the most detailed suggestions for using typography to produce readable programs. Their suggestions included using variable character fonts, typefaces, sizes, ruled lines, and background grey-scale tints to format source code listings.

Baecker and Marcus reported on two controlled studies. They had hoped to show that their style guidelines help comprehension of moderately-sized C programs. Unfortunately, their results were mixed and the authors suggested further experimentation.

Experiment 3 was designed to show that typographic signals can help segment programs into modules and emphasize high-level comments and meaningful identifier names, thereby making it easier for experienced programmers to understand an unfamiliar program and to locate material within the program.

6.1.2 Preview Statements

Preview statements are short statements added to text that reveal information appearing later in the text. They have been shown to aid text comprehension [Spyridakis, 89b; Spyridakis and Standal, 87; Glover, et al., 88; Lorch, 85].

Programmers are taught and encouraged to use header comments before each module to state the high-level action of the routine. Often, however, header
comments contain low-level details and can be quite lengthy. Experiment 3 was designed to show that when header comments are written as preview statements, they significantly aid comprehension of the routines they reference.

While the usefulness of header comments in aiding comprehension may seem apparent, results from controlled studies have been mixed. Shneiderman [77], using beginning programmers as subjects, showed that a 26-line FORTRAN program that began with a high-level comment block was easier to modify compared to the program without the high level comment but with 19 one-line comments interspersed throughout the code.

In contrast, Tolman [88], using computer science students as subjects and moderately-sized Pascal programs, showed that header comments were not as important as good mnemonic procedure names for aiding comprehension and that comments embedded in the procedure body were more effective than header comments for improving performance on a maintenance task.

6.1.3 Headings

Headings are words or short phrases added to text to label the material that follows. They tell readers where they are and where they are going. Headings are usually typographically signaled to differentiate them from the material they reference.

Text researchers have shown that headings help readers scan, select, and retrieve material as well as aiding comprehension [Hartley, 81; Mayer, Dyck, and Cook, 84; Hartley, Kenely, Owen, and Trueman, 80; Loman and Mayer, 83; Spyridakis, 89b; Spyridakis, and Standal, 87].
We feel that meaningful module names, especially when typographically signaled, can act as headings in source code. We use the term *module* to refer to C functions, Pascal procedures, FORTRAN subroutines, etc. Experiment 3 was designed to show that when meaningful module names are used, they significantly aid comprehension of the routines they reference.

Programmers are taught and encouraged to use meaningful names for program identifiers. In particular, module names should be descriptive of the actions the routines performs. However, even though the usefulness of meaningful module names for aiding comprehension seems apparent, controlled studies have found either no effect or inconclusive results [Shneiderman, 80; Sheppard, Curtis, Milliman, and Love, 79].

In Experiment 1, we showed that a meaningful module name helped computer science students understand a short Pascal procedure. Experiment 3 was designed to extend that result to situations where experienced programmers attempted to understand larger programs.

### 6.1.4 Expected First-Order Interactions

As important as showing the effects these three types of signaling have on comprehension of source code listings, Experiment 3 was designed to answer questions about the first-order interaction of effects these signals have on comprehension.

Since typographic signaling has been shown as an effective way to draw attention to and improve retention of selected material, we expected a significant interaction between typographic signaling and preview statements. In other words, by drawing attention to the contents of the preview statements, typographic
signaling should improve comprehension more when a module has a preview statements compared to modules without preview statements.

On the other hand, we expected no interaction between typographic signaling and the type of module name used. Since typographic signaling is expected to aid comprehension through improved retention of the signaled material and faster location during searches through the source code, we predicted that using typographic signaling to highlight module names will improve comprehension to about the same degree regardless of the type of module name used.

Finally, we expected a significant interaction of effects between names and preview statements. We felt that when a module already had a meaningful name acting as a heading, then the addition of a preview statement would not significantly improve comprehension. In other words, the positive effects of meaningful names and preview statements would not be additive since they supply redundant information.

6.2 Method

A within-subject, 2x2x2 factorial experiment was run. Eight professional programmers studied the C source code that implemented the X_Windows information feature and then answered a series of questions about the program.

The source code was modified to test the effects of three independent variables. The independent variables were (a) use of typographic signaling, (b) presence of a preview statement, and (c) type of module name. Each independent variable had two levels.
Subjects answered two types of questions about the program: hypothesis questions and location questions. Hypothesis questions proposed an assertion about the actions of a particular routine and required the subjects either to confirm or deny the hypothesis. Location questions required the subjects to locate the routine that performed the specified action.

The dependent variables were (a) accuracy of responses to the hypothesis questions, (b) confidence in the accuracy of responses to the hypothesis questions, (c) time to respond to the hypothesis questions, (d) accuracy of responses to the location questions, (e) types of errors made on the location question, and (f) time to respond to the location questions.

6.2.1 Subjects

Eight professional programmers served as subjects. They were chosen from a pool of 14 programmers who responded to the following electronic bulletin board solicitation:

Ed Gellenbeck, an Oregon State University Ph.D. student, is seeking professional programmers to participate in an experiment investigating program comprehension. He is working with Professor Curt Cook at OSU.

If you are interested, proficient in the C programming language, willing to devote around 1.5 hours in early January 1991, and would like to earn an extra $50.00 please contact Ed at edward@mist.cs.orst.edu

The experiment was conducted in Portland, Oregon from January 14 to January 16, 1991. Subjects were paid $50.00 for participating.

The Research Office at Oregon State University approved the experiment’s methodology and determined it to be exempt from review by Oregon State University’s Committee for the Protection of Human Subjects. All subjects
signed and received a copy of an informed consent form (Appendix F). They were
debriefed at the end of the experiment and mailed a synopsis of the results.

Table 6.1 provides statistics on the subjects’ background. This
information was compiled from questionnaires completed by each subject
(Appendix G). All subjects had over five years of professional programming
experience and considered themselves very familiar with the C programming
language.
### Table 6.1

Subject Background Data, Experiment 3

<table>
<thead>
<tr>
<th>Subject</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Years professional programming experience</td>
<td>9</td>
<td>25</td>
<td>5</td>
<td>13</td>
<td>8</td>
<td>14</td>
<td>23</td>
<td>8</td>
<td>13.1</td>
</tr>
<tr>
<td>2. Years professional maintenance programming experience</td>
<td>9</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>4.3</td>
</tr>
<tr>
<td>3. Years C programming language experience</td>
<td>9</td>
<td>10</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>7</td>
<td>7.3</td>
</tr>
<tr>
<td>Degree of familiarity</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3.0</td>
</tr>
<tr>
<td>4. Years using windowing interface systems</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>3</td>
<td>3.4</td>
</tr>
<tr>
<td>Degree of familiarity</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2.1</td>
</tr>
<tr>
<td>5. Years programming windowing interface system software</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1.3</td>
</tr>
<tr>
<td>Degree of familiarity</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td>6. Years using X_Windows</td>
<td>.1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0.8</td>
</tr>
<tr>
<td>Degree of familiarity</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1.6</td>
</tr>
<tr>
<td>7. Years programming X_Windows software</td>
<td>.1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>Degree of familiarity</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Degree of familiarity:  
1 = not familiar  
2 = somewhat familiar  
3 = very familiar
6.2.2 Material

Preparing the materials for Experiment 3 centered around three main tasks. They were (a) preparation of the source code, (b) preparation of the hypothesis questions, and (c) preparation of the location questions.

Characteristics of the Source Code. Subjects studied the source code that implemented the X_Windows information feature. This feature allowed users of X_Windows to request information about a specific window. The source code was originally developed at M.I.T. and is now bundled with the HP Unix system. It had been used before in a program comprehension experiment by Oman and Cook [90b].

The code was written in the C programming language. It contained 913 lines of code and comments. In addition to the main routine, it contained 24 other routines. The source code was modified for Experiment 3 by the insertion of preview statements, changes to its function identifier names, and the use of typographic signaling.

Preview statements. We developed preview statements for all 24 routines in the program. Preview statements are one or two sentences that explain the high-level operation or purpose of a routine and that appear in the source code directly before the code that implements the routine.

Preparation of the preview statements was fairly easy. In all cases, the source code already contained comments preceding the routines that acted as preview statements. These comments were used to develop the preview statements. In some cases, we were able to use the comments exactly as they appeared in the original source code. In other cases, the comments were shortened
or elaborated to reduce differences in length and descriptiveness among the preview statements.

Appendix H lists the preview statements. Two independent judges evaluated the preview statements for appropriateness and the preview statements judged inappropriate were revised.

As an independent variable, preview statements had two levels: either it was present before the routine or it was not present. Therefore, in the 2x2x2 factorial design, half of the routines had preview statements, the other half did not.

**Module Names.** We changed every function name in the program to either a meaningful name or a neutral name. Meaningful names were descriptive of the high-level operation or purpose of the routine; neutral names were ambiguous and required the programmer to look at the code in order to understand what the routine did. We assumed that meaningful function names, especially when typographically signaled, acted as headings in the source code.

Meaningful names were developed by restating the preview statement as a short phrase beginning with a verb. Neutral names were developed by concatenating neutral words and numbers together. For each routine, the meaningful and neutral names had exactly the same length.

Appendix I lists the module names. Two independent judges evaluated the names for appropriateness and the names judged inappropriate were revised.

As an independent variable, module names had two levels: meaningful or neutral. Therefore, in the 2x2x2 factorial design, half of the routines had meaningful names, the other half had neutral names.

**Typographic Signaling.** Typographic signaling was used to draw attention to the preview statements and module names. We developed the
typography standard used to format the source code based on some of the suggestions of Baecker and Marcus [90]. Source code listings were printed on 8.5" by 11" standard copier paper using a laser printer. The control font was 8.5 point lineprinter. Routine names were typographically signaled using 14 point boldface type. Preview statements were typographically signaled using 14 point boldface type, drawing a box around the statements, and using a light grey background shading.

As an independent variable, typographic signaling had two levels: either it was present or it was not present. Therefore, in the 2x2x2 factorial design, half of the routines were typographically signaled, the other half were not.

**Assigning the Conditions.** The 2x2x2 factorial design produced eight conditions. Figure 6.1 illustrates the eight conditions. We prepared eight versions of the source code, one for each subject. For each version, each of the 24 routines was randomly assigned one of the eight conditions. Therefore, each version of the program had three routines corresponding to each of the eight conditions.

For all versions, the main routine was assigned the identifier name *main*, given a preview statement, and was typographically signaled.
Figure 6.1
Eight Conditions, Experiment 3

1. (none)          function Wind_14_12_Meth1 (code, table)

2. (n)             function Look_Up_In_Table (code, table)

3. (p)             /*
                    * Find a code in an array and return the
                    * text string associated with the code.
                    */
                    function Wind_14_12_Meth1 (code, table)

4. (n, p)          /*
                    * Find a code in an array and return the
                    * text string associated with the code.
                    */
                    function Look_Up_In_Table (code, table)

5. (t)             function Wind_14_12_Meth1 (code, table)

6. (n, t)          function Look_Up_In_Table (code, table)

7. (p, t)          /*
                    * Find a code in an array and return the
                    * text string associated with the code.
                    */
                    function Wind_14_12_Meth1 (code, table)

8. (n, p, t)       /*
                    * Find a code in an array and return the
                    * text string associated with the code.
                    */
                    function Look_Up_In_Table (code, table)

Factors:
Module Name (N): meaningful or neutral (none)
Preview Statement (P): present or none
Typographic Signaling (T): present or none
Hypothesis Questions. We developed 24 hypothesis questions, one for each routine. These questions proposed an assertion about the routine and were answered with either a Yes or No response. One-half of the questions were correctly answered with a Yes response, the other half were correctly answered with a No response.

Each question had two versions: one in which the routine mentioned had a meaningful name, the other in which the routine mentioned had a neutral name. Subjects were asked the version of the question appropriate to the routine’s name in the version of the source code they were attempting to understand.

Appendix J lists the hypothesis questions. Two independent judges evaluated the hypothesis questions for appropriateness and the questions judged inappropriate were revised.

Subjects were asked the 24 hypothesis questions in random order and each subject received a different random order.

Location Questions. We developed 24 location questions, one for each routine. Location questions took the form "Locate the routine that does ..." Subjects answered location questions by typing in the line number (+-5 lines) where the routine was declared in the source code listing.

Appendix K lists the location questions. Two independent judges evaluated the location questions for appropriateness and the questions judged inappropriate were revised.

Subjects were asked the 24 location questions in random order and each subject received a different random order.

Pilot Study. We conducted a pilot experiment to test the appropriateness of the materials and to fine-tune the procedure. Two computer science graduate
students were used as subjects. The pilot study was successful and produced no significant changes in the design of the experiment.

6.2.3 Procedure

The procedure consisted of six parts. These parts were (a) experimenter orienting the subject, (b) subject studying of the source code, (c) subject answering the hypothesis questions, (d) subject answering the location questions, (e) subject completing the background questionnaire, and (f) experimenter debriefing the subject.

We wrote a computer program to administer parts (b), (c), and (d). It provided the instructions to the subjects, asked the questions, and recorded the subjects' responses to the questions.

Before the experiment, subjects were randomly assigned to one of the eight versions of the source code. Each subject was tested individually. The experiment was scheduled to run for ninety minutes. Seven of the subjects finished within 90 minutes, the other subject finished within two hours.

Orienting the Subject. The experimenter greeted the subjects, thanked them for participating, and asked them to read and sign an informed consent form (Appendix F). Subjects read that they would be asked to study a C program and then answer some questions about the program.
Once the informed consent form was signed, subjects were seated at a desk containing a source code listing and a personal computer. They were told that the experiment was completely computer-directed, but in case problems arose, the experimenter would remain in the room the entire time.

*Studying the Source Code.* Subjects read on the computer that they were to imagine themselves in the following situation:

You are asked to make some changes to an existing program. However, this program is completely unfamiliar to you. The programmer who normally maintains this program is leaving on vacation. Since the modification is urgent, it must be done by you.

Your task is to become familiar with this program during the next fifteen minutes at which time the "vacationing" programmer will come and explain the changes you need to make. You will familiarize yourself with the program by studying its source code listing. During the next ten minutes, you cannot run the program, make notes, or ask any questions about it. Just learn it the best you can by reading its source code.

The computer will keep track of your study time and will sound a beep when the ten minutes have elapsed. When the ten minutes of study time are over, put down the source code listing and follow the directions given on the computer.

This situation is similar to ones used by Pennington [87a] and Oman and Cook [90b].

Subjects studied the program for ten minutes by reading its source code listing. The computer program kept track of the time and produced a beep when the ten minutes were up. Along with the beep, subjects read on the computer that they were to put down the source code listing and proceed with the next phase of the experiment.
Answering the Hypothesis Questions. The next phase of the experiment had subjects respond to the 24 hypothesis questions. Subjects read the following instructions on the computer display:

Now you will respond to a series of statements about the program. After each statement, type 'y' if the statement is correct or 'n' if the statement is incorrect. You may refer to the source code listing to check the correctness of any of the statements.

Your 'y' or 'n' answers are being timed. Your goal is to respond as quickly and as accurately as possible. If you do not respond 'y' or 'n' to any one of the statements within 5 minutes, your time is up and the computer will beep. At this point, enter your best guess at a 'y' or 'n' answer and go on to the next statement.

After each 'y' or 'n' response, you will be asked to rate how confident you are that the answer you gave is correct. Rate your confidence by entering a value from 0 to 100 where:

\[
\begin{align*}
0 & \quad \text{no confidence} \\
100 & \quad \text{positive} \\
\text{the answer you gave is correct} & \quad \text{the answer you gave is correct}
\end{align*}
\]

Subjects were first presented with one practice hypothesis question that referred to the main routine. After completing the practice question, subjects answered the 24 hypothesis questions, one for each routine. The order of presentation of the questions was random and differed for each subject. The computer program recorded each question's y or n response, confidence rating (0..100), and the elapsed time to respond (msec.).

The computer program kept track of the elapsed time for each question. The timer was started as soon as the question was displayed on the computer screen and stopped when the subject entered a y or n response. If the elapsed time exceeded five minutes, the computer sounded a beep to signal that time for any one particular question was up. When this occurred, subjects were required to enter a
y or n response and continue on with the experiment. Only one subject took over five minutes to answer any one of the hypothesis questions.

**Answering the Location Questions.** After the 24 hypothesis questions, subjects answered the 24 location questions. To begin, they received the following instructions:

Now you will be asked to locate specific modules in the program. After each question, type in the line number from the source code listing where the module being described is declared. You will need to look through the source code listing to find where the module being described is declared. When you find it, enter the line number (followed by a <CR>).

The line number you enter must be correct. If it is incorrect, the computer will beep and you should re-enter the correct line number as soon as you locate it. You responses are being timed. Your goal is to respond as quickly and with as few errors as possible. If you do not enter the correct line number within five minutes, your time is up, the computer will beep, and you should go on to the next question.

Subjects were first presented with one practice location question that referred to the main routine. After completing the practice question, subjects answered 24 location questions, one for each routine. The order of presentation of the questions was random and differed for each subject.

Subjects were required to answer the location question correctly. If an incorrect line number was entered, the computer would beep and the subject informed that the answer they had entered was incorrect and to continue with the question. The computer program recorded the time required for the subject to answer each question correctly. In addition, if the subject entered an incorrect response, the computer recorded the line number of the first incorrect response.

For any one question, the computer program sounded a beep after five minutes had elapsed to signal that time for that particular question was up. Six out
of the 192 location questions were not answered correctly within the five minutes allowed. For those six questions, the value of five minutes was used as the elapsed time.

Completing the Background Questionnaire. After answering the location questions, the computer-directed parts of the program were over. At this point, the experimenter asked the subjects to fill out the background questionnaire (Appendix G).

Debriefing the Subject. Finally, the experimenter told subjects about the purpose of the experiment, the independent variables manipulated, the dependent variables measured, the expected results, asked for subjects’ comments, and answered any questions. All subjects were mailed a synopsis of the results three weeks after the experiment was conducted.

6.3 Results

This section reports on the analysis of the data from Experiment 3. The data was collected from the subjects’ responses to the two types of questions: hypothesis questions and location questions.

In addition to testing the effects of the three factors under investigation, we examined for possible differences between the questions we used or modules in the program. Since question number and module number corresponded exactly, i.e. question 3 referred to material in module 3, any differences we found between questions or modules could be due to either the question being harder (or easier) to answer than the other questions or the module being harder (or easier) to understand than the other modules.
6.3.1 Hypothesis Questions

Three dependent variables were recorded for each of the 24 hypothesis questions. They were (a) accuracy, (b) confidence rating, and (c) response time.

Accuracy. Accuracy was measured as the number of incorrect responses to the hypothesis questions. Table 6.2 gives the total incorrect responses for each of the eight conditions across subjects. Since each of the eight subject answered three hypothesis questions for each condition, the number of incorrect responses could range from zero to 24. Appendix L provides the accuracy data per subject and the Chi-Square test results.

Table 6.2

Hypothesis Questions -- Number of Incorrect Responses by Condition, (0..24)

<table>
<thead>
<tr>
<th></th>
<th>No Typographic Signaling</th>
<th>Typographic Signaling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Preview Statement</td>
<td>Preview Statement</td>
</tr>
<tr>
<td>Neutral Name</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Meaningful Name</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Subjects were very accurate in their responses. Overall, only 7% of the hypothesis questions were answered incorrectly. The data suggests that subjects were more accurate answering hypothesis questions when the module was assigned a meaningful name (p = .01). Neither preview statements nor typographic signaling
appeared to make a significant difference in the accuracy of responses ($p = .15$ and
$p = .96$ respectively).

Table 6.3 gives the total number of incorrect responses for each
hypothesis question. Since errors were low and spread fairly evenly across the
modules, it appears that the questions were valid and that no question was
significantly harder to answer than the others.

Table 6.3
Hypothesis Questions -- Number of Incorrect Responses by Question (0..8)

<table>
<thead>
<tr>
<th>Question</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>number incorrect</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>number incorrect</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>number incorrect</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Confidence Ratings. After every hypothesis question, subjects rated their confidence that the answer they had just entered was correct. Confidence was rated on the scale 0..100. A 0 rating meant that the subject had no confidence that the answer they entered was correct; a 100 rating meant that they were positive the answer they entered was correct. Each subject answered three hypothesis questions for each condition, therefore, their confidence ratings for each condition were the total of their three ratings (0..300).

Table 6.4 gives the mean confidence ratings for each of the eight conditions across subjects. Appendix M provides the confidence data per subject and the Multifactor Analysis of Variance test results. As the means indicate, subjects were very confident in their answers. There was no evidence of significant main or first-order interaction effects, \( p > .05 \).

Table 6.4

Hypothesis Questions -- Mean Confidence Ratings by Condition (0..300)

<table>
<thead>
<tr>
<th></th>
<th>Neutral Name</th>
<th>Meaningful Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Typographic Signaling</td>
<td>Typographic Signaling</td>
</tr>
<tr>
<td></td>
<td>No Preview Statement</td>
<td>Preview Statement</td>
</tr>
<tr>
<td></td>
<td>265.1</td>
<td>269.4</td>
</tr>
<tr>
<td></td>
<td>266.5</td>
<td>267.4</td>
</tr>
</tbody>
</table>

S.E. = 10.1
N = 8
Table 6.5 gives the mean confidence ratings for each question. They are listed in order of most confident to least confident, or, in more general terms, easiest to hardest. The analysis of variance, using subjects as the blocking factor, showed no significant differences in confidence ratings between questions, $F(23,191) = 1.28, p = .19$. This is further evidence that the hypothesis questions were valid and that subjects did not perceive any question as significantly harder or easier to answer than the others.

Table 6.5

Hypothesis Questions -- Mean Confidence Rating by Question (0..100)

<table>
<thead>
<tr>
<th>Question</th>
<th>Rating</th>
<th>Homogeneous Groups (Tukey HSD, 95%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>97.9</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>97.3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>97.3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>97.1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>96.8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>96.5</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>96.1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>96.1</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>95.4</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>94.9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>94.6</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>93.1</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>92.1</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>91.8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>89.1</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>88.8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>87.8</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>87.5</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>86.5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>86.1</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>85.6</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>83.0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>83.0</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>78.6</td>
<td></td>
</tr>
</tbody>
</table>

S.E. = 4.9
N = 8
Response Times. The computer program measured the time that elapsed between displaying a hypothesis question on the computer display and the entering of a yes or no response. Each subject answered three hypothesis questions for each condition, therefore, their response times for each condition were the total of their three response times. It was assumed that faster response times indicated that modules were easier to understand.

Analysis of the plotted residuals versus predicted values indicated a log-transformation on the response time data was needed to meet the Analysis of Variance test requirements of constant variance and normality of the data. Appendix N provides the response time data per subject and the Multifactor Analysis of Variance test results on the transformed data.

Table 6.6 gives the mean response log times for each of the eight conditions across subjects.

Table 6.6
Hypothesis Questions -- Response Time Means by Condition, LOG(sec)

<table>
<thead>
<tr>
<th>Neutral Name</th>
<th>No Typographic Signaling</th>
<th>Typographic Signaling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Preview Statement</td>
<td>Preview Statement</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>5.419</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5.158</td>
</tr>
<tr>
<td>Meaningful Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5.130</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>4.785</td>
</tr>
</tbody>
</table>

S.E. = 0.1318
N = 8
Table 6.7 shows the ranking of the eight conditions in order of easiest to hardest. Conditions that are connected by a line do not have a significantly different degree of difficulty using the Tukey HSD method, 95 percent confidence intervals.

**Table 6.7**

Hypothesis Questions -- Ranking of Conditions by Response Times

<table>
<thead>
<tr>
<th>Condition</th>
<th>8</th>
<th>6</th>
<th>4</th>
<th>7</th>
<th>3</th>
<th>5</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
</table>

All the main effects were significant and none of the interactions were. Table 6.8 shows the differences between the mean log times for each of the main effects and the significance levels.

**Table 6.8**

Hypothesis Questions -- Response Time Main Effects, LOG(sec)

<table>
<thead>
<tr>
<th>Module name</th>
<th>Neutral</th>
<th>Meaningful</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5.125</td>
<td>4.855</td>
<td>.0052</td>
</tr>
<tr>
<td>Preview Statement</td>
<td>None</td>
<td>Present</td>
<td>$p$</td>
</tr>
<tr>
<td>Typographic Signaling</td>
<td>5.123</td>
<td>4.867</td>
<td>.0058</td>
</tr>
<tr>
<td></td>
<td>5.107</td>
<td>4.872</td>
<td>.0141</td>
</tr>
</tbody>
</table>

S.E. = 0.06529
N = 32
Table 6.9 gives the mean log times for each hypothesis question. The analysis of variance, using subjects as the blocking factor, showed a significant difference between questions, $F(23,161) = 1.975, p = .0078$. However, as Table 6.9 shows, there were only two homogeneous groups identified and these groups differed by only one element (Tukey HSD method, 95 percent confidence intervals).

Hypothesis question 19 appeared to be the hardest to answer. In particular, one subject had the most difficult time with this question, taking the entire five minutes to respond. In this case, module 19 had the control treatment assigned, that is, neutral name, no preview statement, and no typographical signaling.

Hypothesis question 1 appeared to be the easiest to answer. This question dealt with the first module in the program. Therefore, its location in the source code probably aided the subjects' initial understanding of it and later location.

We conclude from our analysis of the correctness, confidence rating, and response time data by question that the hypothesis questions used in Experiment 3 were valid and close enough in difficulty that our experimental results dealing with differences between treatments were not biased.
Table 6.9
Hypothesis Questions -- Response Time Means by Question, LOG(sec)

<table>
<thead>
<tr>
<th>Question</th>
<th>Mean</th>
<th>Homogeneous Groups (Tukey HSD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.010</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>3.071</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>3.192</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3.202</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3.311</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>3.361</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3.391</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3.506</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>3.609</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>3.648</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>3.748</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>3.772</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3.804</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>3.821</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3.829</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>3.840</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3.840</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>3.851</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>3.940</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>3.957</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>4.149</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4.202</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>4.212</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>4.446</td>
<td></td>
</tr>
</tbody>
</table>

S.E. = 0.2694
N = 8
6.3.2 Location Questions

Three dependent variables were measured for each of the 24 location questions. They were (a) accuracy, (b) types of errors, and (c) response time.

Accuracy. Even though subjects were required to enter the correct answer for a location question before preceding on to the next question, subjects occasionally attempted incorrect answers. When this would happen, the computer would beep and inform the subject that an incorrect answer had been entered. Subjects were required to continue with the question until they got the correct answer or five minutes had elapsed.

The computer program kept track of the first wrong answer attempted for each question. It was assumed that conditions harder to locate would have more wrong answers associated with them.

Table 6.10 shows the total number of wrong answers attempted for each of the eight conditions across subjects. Appendix O provides the data per subject and the Chi-Square test results. In some cases, more than one incorrect answer was entered for a particular question. However, regardless of the number of incorrect answers attempted for any one question, it was treated as one incorrect answer attempted.
Table 6.10

Location Questions -- Number of Incorrect Answers Attempted
by Condition (0..24)

<table>
<thead>
<tr>
<th></th>
<th>No Typographic Signaling</th>
<th>Typographic Signaling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Preview Statement</td>
<td>Preview Statement</td>
</tr>
<tr>
<td>Neutral Name</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Meaningful Name</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Thirty-six out of the 192 location questions answered, or 19 percent, had an incorrect answer attempted. The errors were spread fairly evenly across the conditions. The Chi-Square test showed no significant differences attributable to any of the signal types investigated (p > .05).

Table 6.11 gives the number of incorrect responses for each location question. Errors were spread fairly evenly across modules indicating that the location questions were valid and no question was significantly harder to answer than the others.
Table 6.11
Location Questions -- Number of Incorrect Responses by Question (0..8)

<table>
<thead>
<tr>
<th>Question</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>number incorrect</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>number incorrect</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>number incorrect</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Error Types.** The conditions of each of the wrong answers attempted were examined to see if any of the signals investigated influenced the types of errors made. For example, it would be interesting to show a bias by subjects to incorrectly select typographically signaled modules as wrong answers to the location questions.

Table 6.12 shows the total number of each of the eight conditions assigned to the wrong answers attempted. Appendix P provides the data per subject and the Chi-Square test results.
Table 6.12
Location Questions -- Type of Errors by Condition

<table>
<thead>
<tr>
<th></th>
<th>No Typographic Signaling</th>
<th>Typographic Signaling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Preview Statement</td>
<td>Preview Statement</td>
</tr>
<tr>
<td>Neutral Name</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Meaningful Name</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

The types of errors were spread fairly evenly across the conditions. Results from the Chi-Square test did not show any bias by subjects to incorrectly select modules as answers to location questions based on signaling (p > .05).

Location Times. The computer program measured the time that elapsed between displaying a location question on the computer display and the subjects' entering of the correct response. Each subject answered three location questions for each condition, therefore, their location times for each condition were the total of their three responses. It was assumed that faster location times indicated that modules were easier to locate.

Analysis of the plotted residuals versus predicted values indicated a log-transformation on the location time data was needed to meet the Analysis of Variance test requirements of constant variance and normality of the data. Appendix Q provides the data per subject and the Multifactor Analysis of Variance test results on the transformed data.
Table 6.13 gives the subjects' mean location log times for each of the eight conditions. Six of the 192 location questions were not answered correctly within the five minutes allowed. Those six questions were assigned a location time of five minutes.

Table 6.13

Location Questions -- Mean Location Times by Condition, LOG(sec)

<table>
<thead>
<tr>
<th></th>
<th>No Typographic Signaling</th>
<th>Typographic Signaling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Preview Statement</td>
<td>Preview Statement</td>
</tr>
<tr>
<td>Neutral Name</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>5.151</td>
<td>4.884</td>
</tr>
<tr>
<td>Meaningful Name</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4.942</td>
<td>4.485</td>
</tr>
<tr>
<td></td>
<td>S.E. = 0.2225</td>
<td></td>
</tr>
<tr>
<td></td>
<td>N = 8</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.14 shows the ranking of the eight conditions in order of easiest to hardest to locate. Conditions that are connected by a line do not have a significantly different degree of difficulty using the Tukey HSD method, 95 percent confidence intervals.

Table 6.14

Location Questions -- Ranking of Conditions by Response Times

<table>
<thead>
<tr>
<th>Condition</th>
<th>7</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
</table>
The results showed that only two of the main effects were significant. Meaningful names helped location \( p = .05 \), preview statements helped more \( p = .01 \), and typographic signaling did not help \( p = .37 \). In addition, none of the first-order interactions were significant. Table 6.15 shows the mean log times for each of the significant main effects and their significance levels.

**Table 6.15**

Location Questions -- Location Time Main Effects, LOG(sec)

<table>
<thead>
<tr>
<th>Module name</th>
<th>Neutral 4.953</th>
<th>Meaningful 4.628</th>
<th>( p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neutral</td>
<td></td>
<td></td>
<td>.0501</td>
</tr>
<tr>
<td>Preview Statement</td>
<td>None 4.998</td>
<td>Present 4.584</td>
<td>.0136</td>
</tr>
</tbody>
</table>

S.E. = 0.1146
N = 32

Table 6.16 gives the mean location log times for each location question. The Analysis of Variance test results with this data, using subjects as the blocking factor, did show a significant difference between questions, \( F(23,161) = 5.24, p < .0001 \). Table 6.16 shows the four homogeneous groups identified (Tukey HSD method, 95 percent confidence intervals).
Table 6.16

Location Questions -- Location Time Means by Question, LOG(sec)

<table>
<thead>
<tr>
<th>Question</th>
<th>Mean</th>
<th>Homogeneous Groups (Tukey HSD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>2.214</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>2.313</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2.352</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>2.878</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>2.967</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2.974</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>2.989</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3.228</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>3.249</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3.300</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>3.312</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3.380</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3.428</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3.464</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3.503</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>3.732</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>3.761</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>3.839</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>3.878</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3.899</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>4.082</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>4.143</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>4.541</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4.701</td>
<td></td>
</tr>
</tbody>
</table>

S.E. = 0.2806

N = 8
As expected, the data indicated that modules appearing near the beginning or end of the source code listing tended to be located quicker than modules in the middle of the listing. One exception to this is Module 3. Module 3 may be characterized as an utility routine and would more logically appear in the second half of the program listing rather than as the third module from the start. If subjects started their search for Module 3 near the end of the program, this could explain why they had so much difficulty locating it.

Even though the location questions used in Experiment 3 differed in difficulty, since the experimental conditions were applied randomly to the modules, our results dealing with differences between treatments were not biased by the differences in difficulty among the location questions.

6.4 Discussion

The results showed that preview statements, meaningful names, and typographic signaling all aided comprehension of computer program listings. However, of the three, only preview statements and meaningful names helped programmers locate information within the program. We observed no significant interaction of effects.

Experiment 3 was designed to minimize guessing by subjects. We wanted to minimize guessing in order to strengthen the use of response time as the primary dependent variable. If subjects had done a lot of guessing, this could have accounted for differences in response times.

Subjects could err in their responses to the hypothesis questions. Fortunately, they made very few errors. Only 7 percent of the questions were answered incorrectly. In addition, they were uniformly high in their confidence in
the accuracy of their responses. Analysis of the confidence data showed no significant differences between conditions.

Since subjects had to answer the location questions correctly, guessing was not a concern. While subjects did enter wrong responses to almost one-fifth of the location questions, they were required to continue with the question until they answered it correctly. As a consequence, location time measured the time to correctly answer the location question, regardless of guessing.

Therefore, we conclude that guessing by subjects was effectively minimized, and we discuss the effects of signaling on comprehension based on the differences we observed in the response time data.

Because we observed no significant interaction of effects, we can assume that preview statements, meaningful module names, and typographic signaling were acting independently and can examine their main effects on comprehension. Our results showed that all three types of signals helped subjects confirm hypotheses about the program. Only preview statements and meaningful names helped with location.

Surprisingly, while we had expected some interaction of effects, we observed none. Our results indicated that signals may be combined for a cumulative effect. In other words, the addition of another signal to conditions that already used one signal always resulted in modest improvements in response times.

We had expected to see dramatic improvements in comprehension when typographic signaling was used to provide emphasis to preview statements. However, only modest improvements resulted. Our results agree with the conclusions of Sheppard, Curtis, Milliman and Love [79]. There appears to be an upper limit to the usefulness of style in aiding comprehension. Aids beyond this limit may contribute little to improved performance.
The rationale behind using two types of questions in the design of Experiment 3 was to provide insight into how signaling aided comprehension. Signaling may have helped our subjects confirm hypothesis about the program by aiding access or location of material in the code and/or by making it easier for them to understand the material once they locate it. Analysis of the location time data failed to show that typographic signals significantly sped up location of material in the program.

We considered the possibility that, in this case, typographic signaling may not have aided location since the code already used vertical spacing and pagination to cue the module structure. All but five of the modules appeared at the top of a new page. The five modules that did not begin on a new page were separated from the previous module by either seven blank lines (conditions 1, 2, 5, and 6) or by two blank lines followed by a preview statement (conditions 3, 4, 7, and 8).

Analysis of the location time data for just the location questions associated with these five modules that did not begin on a new page showed no significant differences in location times between those modules with typographic signaling and those modules without typographic signaling. Indeed, we found a slightly quicker mean location time for those modules without typographic signaling compared to those modules with typographic signaling.

Text comprehension researchers have shown that spatial cues are often more effective than typographic cues when presenting list-like information [Hartley, Trueman, and Burnhill, 80]. Source code may share this characteristic, that is, indentation and vertical spacing may be more effective than boldface type and vertical lines for segmenting code into modules.

We feel that, of the three signals investigated, preview statements provided programmers with the easiest access to high-level information about the
program. As a consequence, preview statements speed up location of material much more the meaningful names or typographic signaling.

Our results differ from Tolman [88]. He showed that meaningful procedure names were more effective than header comments for improving comprehension. However, his header comments were longer and more detailed than our preview statements. We speculate that when comment blocks become too long, programmers tend to rely on the comments less and look for other clues in the program to aid comprehension.

Our results elaborate on Baecker and Marcus [90] by showing that professional programmers can benefit from typographic signaling. Baecker and Marcus, using two 200-line C programs and 44 college students as subjects, showed that typographic signaling helped their subjects answer questions on a comprehension test when the program was relatively dense and contained embedded comments. Typographic signaling did not help when the program was relatively clean. We considered the 913-line program we used as dense and fairly hard to understand.

We feel that the benefits of typographic signaling as an aid to program comprehension are more apparent when the source code is fairly long and difficult to understand. This agrees with research on the effectiveness of signaling text passages such as Spyridakis' [89a] conclusion that signaling produced strong and consistent benefits only when applied to text passages of some length and difficulty.

In conclusion, while typographic signaling is helpful, it does not produce dramatic improvements in comprehension and, at least in Experiment 3, did not aid location. Our results showed that careful use of commenting to preview the contents of each module and carefully chosen meaningful module names aided
comprehension more than typographic signaling. In addition, we conclude that multiple signaling of source code may produce only marginal improvements in comprehension once some threshold is reached. Designers of systems should carefully weigh the costs associated with adding signals to code.
Chapter 7

Guidelines for Using Signaling to Aid Comprehension

In this chapter, we present guidelines for using signaling to make programs easier to understand. We based these guidelines on the results from Experiment 3, results from an observational experiment we conducted investigating on-line browsing of source code, and from previous research from both text and program comprehension literature.

Chapter 7 is divided into two sections. The first section presents guidelines for adding preview statements, headings, and typographic cueing to hard-copy source code listings to aid comprehension. Following that, we discuss guidelines for using signaling to aid on-line browsing of source code.

7.1 Using Signals with Hard-Copy Listings.

Experiment 3 investigated the use of signals to aid comprehension of hard-copy listings. Even with the large-screen, multi-window workstations environments in use today, we feel the majority of programmers still make use of hard-copy listings for comprehension. Six out of the eight professional programmers used in Experiment 3 reported that they regularly used hard-copy listings when studying source code. Oman and Cook [90a] polled several
professional programmers and all of them reported that they used hard-copy listings for program study.

7.1.1 Preview Statements

Hartley [81] summarized the research on preview statements by claiming that they aided text comprehension by providing (a) an overview of the content of the text so readers may decide whether or not to read it, (b) an overview to its organization, and (c) information about the text’s main points and conclusions. Preview statements in source code should provide these same types of information.

Program files should begin with a preview statement that supplies an overview, or concise summary, of what the program does. For example, the X_Wininfo program used in Experiment 3 could have as its overview preview statement:

The program reports user-requested information about a specific window. It is located in three files: xwininfo.c, dsimple.c, and dsimple.h.

This preview statement should appear at the start of file xwininfo.c. Likewise, the files dsimple.c and dsimple.h should have preview statements operating as overviews that provide readers with an instant summary of what each file is about and its organization.

In addition, every module of the program needs a preview statement that concisely states the high-level purpose or operation of the routine. We recommend writing the preview statement before writing the source code. Then, if one has difficulty writing the preview statement, this suggests that the module is not well-defined or may be performing more than one high-level operation.
Experiment 3 showed that preview statements, in addition to aiding comprehension, helped readers locate material in the program. Therefore, even if a module is short and easily understood from its code, including a preview statement before the module should help readers access the material.

Preview statements should be short, seldom exceeding two to three lines in length. They can be kept short by keeping detailed comments separate from the preview statements. Preview statements should appear immediately before the material they reference, allowing readers to quickly decide whether or not the material is of interest to them.

Finally, typographic signaling should be used to differentiate preview statements from the material they reference. We recommend using boldfaced lower case type and ruled lines or a box to differentiate the preview statement from the code. Avoid writing preview statement in all capital letters or italics as this has been shown to be hard to read [Hartley, 85].

7.1.2 Headings

Hartley [81] claimed "headings in text orient the reader. They label parts of the text so that readers know where they are and where they are going. Headings in text help the reader to scan, select, and retrieve materials as well as help him comprehend the content."

Meaningful module names act as headings in source code. Providing emphasis to these names with boldface type helps readers use the names to scan and select relevant material. As with preview statements, avoid using all capital letters or italics to signal the module names as they has been shown to be hard to read.
Coming up with meaningful module names can be difficult. In general, published guidelines have recommended that names be descriptive of the operation of the routine, begin with a verb, and be relatively short [Ledgard, 87; Anand, 87; Keller, 90].

We suggest that meaningful module names be based on the preview statements that reference the modules. That is, starting with the preview statement, come up with a short phrase that captures its essence. There are a couple of advantages to this approach. First, it is easier to reduce one to three sentences down into a phrase than to come up with a name based on a vague notion of the function of the module. Second, if a readers are unclear about what the name means, they can simply look at the preview statement for clarification.

Of course, this approach means that one has to develop a preview statement for every module. One could argue that it is as hard to develop a preview statement as it is to come up with a meaningful name. We disagree. We feel it is easier to initially express an idea in one to three sentences than as a name.

7.1.3 Typographic Cues

We began this research with the notion that using typographic cues to highlight important information sources in programs would lead to dramatic improvements in comprehension. The improvements we observed were more modest, dampening our enthusiasm. Typographic signaling helps, but not that much.

Research on text comprehension agrees with our findings. It has shown that typographic cues, while helpful, do not usually make huge differences in
comprehension. Instead of improving overall comprehension, they improve comprehension of the signaled material at the expense of the non-signaled material.

We warn against using too many typographic cues to signal source code. Hershberger and Terry [65] and Glynn and Di Vesta [79] showed that readers find multiple cueing confusing and their presence can actually hurt comprehension. Prettyprinter programs, designed to make code attractive, may in fact make the code harder to understand by introducing so many typographic cues that they confuse the readers.

Typographic cues have been shown to be most effective when readers have been informed in advance why certain things have been signaled [Foster, 77; Hartley, Bartlett and Branthwaite, 80]. Therefore, once an organization has accepted a typographic standard for signaling source code, the reasons behind the standard should be made known to the intended readers of the code.

Spatial cues, rather than typographic cues, have been shown to be more effective for presenting list-like materials in text [Hartley, Trueman and Burnhill, 80]. We expect that this is the case with segmenting modules in source code listings. That is, the use of blank lines and pagination may be more effective for segmenting modules in a program than boldfacing the module names.

7.2 Using Signaling to Aid On-Line Browsing

A number of researchers are working on implementations of on-line browsing environments that manage software documentation [Bigelow, 88; Blum, 88; Fletton and Munro, 88; Wild and Maly, 88]. In this section, we offer suggestions on ways signaling can aid on-line browsing of source code for comprehension.
Our suggestions are based on a pilot study we conducted that observed student programmers working with a prototype on-line, window-based browsing environment. Six computer science students were videotaped using this environment while they attempted to understand and then locate a bug in a 1500-line Pascal program.

While on-line environments reduce paper shuffling and make searching easy, there are some inherent disadvantages. Screen size and scroll rate are limiting factors. It is simply faster to browse a hardcopy listing for information than to browse the same listing on-line.

Our prototype browsing environment provided preview statements for every module. However, instead of appearing before each module in the code, they were not visible until accessed by browsers through a mouse click. We found that our subjects seldom accessed the preview statements, rather they used meaningful module names and study of the code for comprehension.

Based on the results of Experiment 3, we suggest that when a module first appears on the display, its preview statement should be displayed and its code kept invisible until accessed by a mouse click. In this way, the order of presentation is used to highlight the preview statement.

Our prototype browsing environment used boldface type to provide emphasis to module names. However, since the names appeared in the same type size as the code, they did not particularly stand out as headings. A subsequent version of our prototype environment used a larger type size for the module names to provide even more emphasis.

We observed subjects making extensive use of the module names as labels to selectively jump around in the program. Without meaningful names, navigation around the program would have been much harder.
We feel limiting the amount of material which appears in a window and using a consistent layout are critical features to a successful implementation. When the amount of information contained in a window was more than could be contained in one window on the display screen, we observed subjects having difficulty scrolling up and down while trying to understand a module.

Finally, blank space should be liberally used as part of the layout. Reading a full window of text is harder than reading a printed page of text [Rubens, 88]. We feel it is better to use multiple windows of information, accessible through mouse clicks, rather than trying to fit as much information as possible into one window.
Chapter 8

Conclusions and Future Work

In this chapter, we conclude this thesis with a review of the major results, point out some limitations to generalizing these results, summarize our conclusions, and recommend future areas for possible investigation.

8.1 Review of Results

Program comprehension, while much investigated, has focused on student programmers working with small programs. In chapter 4, we presented a framework useful for designing experiments that yield results generalizable to conclusions about ways to make programs easier for professional programmers to read and understand.

We conducted three controlled experiments that investigated several factors which influence comprehension of source code listings. The major results were

(a) Meaningful identifier names helped student programmers gain a high-level understanding of short programs.

(b) A strong code beacon, like the swap operation, was more influential during comprehension of short programs than the program’s name.
(c) Meaningful module names helped professional programmers understand and locate information in a large program.

(d) Header comments, when written as preview statements, helped professional programmers understand and locate information in a large program.

(e) Typographic signaling, designed to provide emphasis and segmentation cues, helped programmers understand a large program.

(f) The effects of meaningful module names, header comments, and typographic cueing were additive. No significant interaction of effects was observed.

(g) Header comments and meaningful module names were more effective at aiding comprehension than typographic signaling.

6.2 Limitations of the Study

This section points out some limitations to generalizing our results to real world situations. Results (a) and (b) were based on data obtained from a controlled experiment in which student programmers studied a small program for a short period of time. While this is a fairly typical scenario for controlled experiments in program comprehension, generalizing the results from such a scenario to comprehension situations where experienced programmers are dealing with large programs is tenuous. Indeed, as chapter 5 noted, different results are expected from professional programmer working with large programs. In particular, we expect procedure names to carry more weight during comprehension of large programs and code idiom beacons, like the swap operation, to carry less weight.
Results (c), (d), and (e) were based on data obtained from a controlled experiment in which eight professional programmers studied a large C program. The variability across subjects was offset by the use of a within-subject design. Only one program was used in the experiment; different results may be observed with different programs or programming languages.

This research focused on high-level comprehension and did not address the detailed understanding required for many re-engineering tasks. However, since many studies have shown a strong relationship between high-level comprehension and performance on a re-engineering task, we assume that by aiding high-level comprehension this will, in turn, aid re-engineering work.

8.3 Conclusions

We extended the notion of signaling to source code listings. In particular, adding preview statements, headings, and typographic cueing to source code listings makes programs easier to read and understand. However, the improvements we observed were modest.

We recommend that programmers use preview statements and meaningful module names when writing programs in order to aid future readers. In addition, typographic signaling should be used to provide emphasis to the preview statements and meaningful module names. However, we caution against using too many typographic cues as this may confuse readers.
8.4 Future Research

The underlying philosophy behind this research was to take techniques that make expository texts easier to read and understand and see if they apply to source code listings. Our focus was on signaling, there remains a variety of other techniques yet to investigate.

Textbooks come with supplemental materials in the form of table of contents, lists of contents, glossaries, and indexes. Oman and Cook [90b] showed that a table of contents helps with a maintenance task; we recommend controlled studies of the effects of other types of supplemental material on comprehension.

Our research showed that preview statements preceding modules helped comprehension. In-line comments are also commonly used to summarize the meaning of several lines of code inside a module. We recommend future research on the effectiveness of these in-line comment as preview statements for short sections of code sections. In this case, it may be shown that typographic signaling that differentiates the comment from the code may make a significant difference in their effectiveness.

We speculate that adding too many typographic cues to source code may actually hurt comprehension by confusing readers. We recommend future research designed to investigate this phenomenon.

An exciting areas for future research is adapting what is known from text comprehension and program comprehension research to the design of on-line browsing environments. Observational studies that analyze the types of information programmers access while using these environments would be useful to designers of these systems.
Our observational study on on-line browsing showed programmers experienced difficulty scrolling through code in a window. We recommend future research examining the tradeoffs between scrolling versus segmenting the material into smaller units and viewing it in multiple windows accessible through a browsing tool.
Bibliography


Glynn, S. W. and Di Vesta, F. J. Control of prose processing via instructional and typographic cues. *Journal of Educational Psychology*, Volume 71, Number 5, 1979, pp. 595-603.


Shneiderman, B. Control flow and data structure documentation: two experiments.  

Shneiderman, B. Measuring computer program quality and comprehension.  


Shneiderman, B. and Mayer, R. Syntactic/semantic interactions in programmer behavior: A model and experimental results.  
*International Journal of Computer and Information Sciences*, Volume 8, Number 3, 1979, pp. 219-238.

Shneiderman, B. and McKay, D. Experimental investigations of computer program debugging and modification.  

Soloway, E. and Ehrlich, K. Empirical studies of programming knowledge.  

Soloway, E., Pinto, J., Letovsky, S., Littman, D. and Lampert, R. Designing documentation to compensate for delocalized plans.  


Spyridakis, J. H. Signaling effects: Increased content retention and new Answers - Part 2.  


Appendices
Appendix A -- Binary Search Procedures, Experiment 1

procedure search (a: arraytype; i, j: integer; var n: integer);
var
  b: boolean;
  s, t, middle: integer;
begin
  b := false;
  t := 1;
  s := i;
  while (t <= s) and not b do begin
    middle := (t + s) div 2;
    if j = a[middle] then
      b := true
    else
      if j < a[middle] then
        s := middle -1
      else
        t := middle + 1;
  end;
  if b then
    n := middle
  else
    n := 0;
end;

Version 1
Meaningful Variable Name,
Meaningful Procedure Name

procedure x (a: arraytype; i, j: integer; var n: integer);
var
  b: boolean;
  s, t, v: integer;
begin
  b := false;
  t := 1;
  s := i;
  while (t <= s) and not b do begin
    v := (t + s) div 2;
    if j = a[v] then
      b := true
    else
      if j < a[v] then
        s := v -1
      else
        t := v + 1;
  end;
  if b then
    n := v
  else
    n := 0;
end;

Version 2
Meaningful Variable Name,
Neutral Procedure Name

procedure search (a: arraytype; i, j: integer; var n: integer);
var
  b: boolean;
  s, t, middle: integer;
begin
  b := false;
  t := 1;
  s := i;
  while (t <= s) and not b do begin
    middle := (t + s) div 2;
    if j = a[middle] then
      b := true
    else
      if j < a[middle] then
        s := middle -1
      else
        t := middle + 1;
  end;
  if b then
    n := middle
  else
    n := 0;
end;

Version 3
Neutral Variable Name,
Meaningful Procedure Name

procedure x (a: arraytype; i, j: integer; var n: integer);
var
  b: boolean;
  s, t, v: integer;
begin
  b := false;
  t := 1;
  s := i;
  while (t <= s) and not b do begin
    v := (t + s) div 2;
    if j = a[v] then
      b := true
    else
      if j < a[v] then
        s := v -1
      else
        t := v + 1;
  end;
  if b then
    n := v
  else
    n := 0;
end;

Version 4
Neutral Variable Name,
Neutral Procedure Name
Appendix B -- Sort Procedures, Experiment 1

procedure y (var a: arraytype; n: integer);
var i, j, k: integer;
t: integer;
begin
  k := 0;
  for i := 1 to n-1 do begin
    j := k + 1;
    while j < n do begin
      if a[j] > a[j+1] then begin
        t := a[j]; a[j] := a[j+1]; a[j+1] := t;
        j := j + 1;
      end;
    end;
    k := t mod 2;
  end;
end;

Version 1
Swap Beacon,
Neutral Procedure Name

procedure y (var a: arraytype; n: integer);
var i, j, k: integer;
b: boolean;
begin
  for i := 2 to n do begin
    k := a[i];
    j := i - 1;
    b := false;
    while (j >= 1) and (not b) do begin
      if k < a[j] then begin
        a[j + 1] := a[j]; a[j] := k;
        j := j - 1;
      end;
    end;
    b := true;
  end;
end;

Version 3
No Swap Beacon,
Neutral Procedure Name

procedure shuffle (var a: arraytype; n: integer);
var i, j, k: integer;
t: integer;
begin
  k := 0;
  for i := 1 to n-1 do begin
    j := k + 1;
    while j < n do begin
      if a[j] > a[j+1] then begin
        t := a[j]; a[j] := a[j+1]; a[j+1] := t;
        j := j + 2;
      end;
    end;
    t := k + 1;
    k := t mod 2;
  end;
end;

Version 2
Swap Beacon,
Misleading Procedure Name

procedure shuffle (var a: arraytype; n: integer);
var i, j, k: integer;
b: boolean;
begin
  for i := 2 to n do begin
    k := a[i];
    j := i - 1;
    b := false;
    while (j >= 1) and (not b) do begin
      if k < a[j] then begin
        a[j + 1] := a[j]; a[j] := k;
        j := j - 1;
      end;
    end;
    b := true;
  end;
end;

Version 4
No Swap Beacon,
Misleading Procedure Name
Appendix C -- Background Questionnaire, Experiment 1

Please answer the questions on this page. Do not look at any of the other pages until you are instructed.

1. Number of years of computer science classes taken (include both undergraduate and graduate-level classes):

2. Number of years of professional programming experience:

3. Number of programming languages you know:

4. Age:

5. Class in school: freshman, sophomore, junior, senior, or grad

6. Sex: male or female
Appendix D -- Binary Search Procedures, Experiment 2

procedure x (a: arraytype;  
  i, j: integer;  
  var n: integer);  
var  
b: boolean;  
s, t, v: integer;  
begin  
b := false;  
t := 1;  
s := i;  
while (t <= s) and not b do  
begin  
v := (t + s) div 2;  
if j = a[v] then  
b := true  
else  
  if j < a[v] then  
s := v - 1  
  else  
    t := v + 1;  
end;  
if b then  
n := v  
else  
n := 0;  
end;  

Version 1  
Neutral Variable Names

procedure x (a: arraytype;  
  i, j: integer;  
  var n: integer);  
var  
b: boolean;  
last, first, v: integer;  
begin  
b := false;  
first := 1;  
last := i;  
while (first <= last) and not b do  
begin  
v := (first + last) div 2;  
if j = a[v] then  
b := true  
else  
  if j < a[v] then  
    last := v - 1  
  else  
    first := v + 1;  
end;  
if b then  
n := v  
else  
n := 0;  
end;  

Version 2  
Variable Names: first, last

procedure x (a: arraytype;  
  i, j: integer;  
  var n: integer);  
var  
b: boolean;  
var  
l: integer;  
begin  
  b := false;  
  t := 1;  
  s := 1;  
  while (t <= s) and not b do  
  begin  
    v := (t + s) div 2;  
    if j = a[v] then  
      b := true  
    else  
      if j < a[v] then  
        s := v - 1  
      else  
        t := v + 1;  
  end;  
  if b then  
    n := v  
  else  
    n := 0;  
end;  

Version 3  
Variable Name: middle
Appendix E -- Background Questionnaire, Experiment 2

Please answer the questions on this page. Do not look at any of the other pages until you are instructed.

1. Number of years of computer science classes taken (include both undergraduate and graduate-level classes):

2. Number of years of professional programming experience:

3. Number of programming languages you know:

4. Age:

5. Class in school: freshman, sophomore, junior, senior, or grad

6. Sex: male or female
Appendix F -- Informed Consent Form, Experiment 3

Understanding Source Code

You are invited to participate in a research study designed to identify important factors in comprehending computer program source code. I am a Phd. student at Oregon State University and this research is part of my dissertation. You were selected as a possible participant because of your professional programming experience and knowledge of the C programming language.

If you decide to participate, you will be asked to study a C program and then answer some questions about the program. This experiment will require about 90 minutes of your time.

There are no risks involved for participants. If you agree to participate and complete the 90 minute experimental session, you will receive $50.00 as compensation for your time.

Any information that is obtained in connection with this study and that can be identified with you will remain confidential.

Your participation is entirely voluntary. If you decide to participate, you are free to discontinue participation at any time.

In about three weeks, after the data has been analyzed and preliminary results determined, a synopsis of the research results will be sent to you.

If you have any questions about the research at any time, please call Ed Gellenbeck at 342-8050 (home), 737-3273 (Computer Science Department, Oregon State University, Corvallis, OR 97331), or by e-mail at edward@mist.CS.ORST.EDU

Your signature below indicates that you have read and understand the information provided above, that you willingly agree to participate, that you may withdraw your consent at any time and discontinue participation, that you will receive a copy of this form, and that you are not waiving any legal claims, rights or remedies.

Signature Date

Address

Social Security Number
Appendix G -- Background Questionnaire, Experiment 3

1. Number of years of professional programming experience: _____ years

2. Number of years professional maintenance programming: _____ years

3. Number of years using C programming language: _____ years
   Degree of familiarity with the C language: ___
   1 - not familiar
   2 - somewhat familiar
   3 - very familiar

4. Number of years experience using windowing interface systems: _____ years
   Degree of familiarity using windowing interfaces: ____
   1 - not familiar
   2 - somewhat familiar
   3 - very familiar

5. Number of years experience programming windowing interface systems software: _____ years
   Degree of familiarity with this software: ____
   1 - not familiar
   2 - somewhat familiar
   3 - very familiar

6. Number of years experience using X_Windows: _____ years
   Degree of familiarity using X_Windows: ____
   1 - not familiar
   2 - somewhat familiar
   3 - very familiar

7. Number of years experience programming X_Windows software: _____ years
   Degree of familiarity with this software: ____
   1 - not familiar
   2 - somewhat familiar
   3 - very familiar
# Appendix H -- Preview Statements, Experiment 3

<table>
<thead>
<tr>
<th>Module</th>
<th>Preview Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Routine to display to the user the command line options when calling xwininfo. note: not all these options may be used at the same time.</td>
</tr>
<tr>
<td>2</td>
<td>find a code in an array and return the text string associated with the code.</td>
</tr>
<tr>
<td>3</td>
<td>Error handler for an invalid window id number. Displays the error message and then exits the program.</td>
</tr>
<tr>
<td>4</td>
<td>Routine to display a window id in dec/hex with name if the window has one. If the window has no name, then print a &quot;no-name&quot; message.</td>
</tr>
<tr>
<td>5</td>
<td>Displays several attributes about the window passed as the routine’s parameter including its location, size, border width, and class.</td>
</tr>
<tr>
<td>6</td>
<td>Displays information about the bits in the window passed as the routine’s parameter including their gravities and backing store hints.</td>
</tr>
<tr>
<td>7</td>
<td>Routine to display all the events in the event mask passed in as the routine’s parameter.</td>
</tr>
<tr>
<td>8</td>
<td>Display info about a window’s events including which events are selected, which events are not propagated, and the status of the redirection override switch.</td>
</tr>
<tr>
<td>9</td>
<td>Display the window id numbers for the root, parent, and each of the children windows associated with the window passed in as the routine’s parameter.</td>
</tr>
<tr>
<td>10</td>
<td>Display the user-supplied and program-supplied window sizing hints including location, size, maximum size, and resizing increments.</td>
</tr>
<tr>
<td>11</td>
<td>Display either the normal window or the zoom window size hints information associated with the window passed in as this routine’s parameter.</td>
</tr>
</tbody>
</table>
## Appendix H -- Preview Statements, Experiment 3 (Cont.)

<table>
<thead>
<tr>
<th>Module</th>
<th>Preview Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Display the window manager and icon information associated with the window passed in as this routine’s parameter.</td>
</tr>
<tr>
<td>13</td>
<td>Standard fatal error routine. Called like printf but with a maximum of 7 arguments. Does not require dpy or screen defined.</td>
</tr>
<tr>
<td>14</td>
<td>Find, return, and remove the name of the display following -display or -d on the command line’s argument list. Don’t go past a lone -</td>
</tr>
<tr>
<td>15</td>
<td>Routine to open a display with correct error handling. Does not require dpy or screen defined upon entry.</td>
</tr>
<tr>
<td>16</td>
<td>This routine opens up a correct display and then stores a pointer to it in dpy. The default screen for this display is then stored in screen.</td>
</tr>
<tr>
<td>17</td>
<td>This routine opens a font. If an error occurs while the font is being opened, the routine prints an error message.</td>
</tr>
<tr>
<td>18</td>
<td>This routine produces a sound on the display terminal.</td>
</tr>
<tr>
<td>19</td>
<td>Locates a window with a given name on a display. If no window have the name, 0 is returned. If more than one window has the name, the first one found is returned.</td>
</tr>
<tr>
<td>20</td>
<td>Routine to provide a common command-line interface to allow users to select one window on the screen for special consideration.</td>
</tr>
</tbody>
</table>
Appendix H -- Preview Statements, Experiment 3 (Cont.)

<table>
<thead>
<tr>
<th>Module</th>
<th>Preview Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>This routine takes a color name and returns the pixel number that when used in the window will be of the color name.</td>
</tr>
<tr>
<td>22</td>
<td>A debugging routine. Used to print messages when past certain points in code so we can tell where we are. Called like printf with up to 7 arguments.</td>
</tr>
<tr>
<td>23</td>
<td>A debugging routine. Prints Blip! on stderr with flushing.</td>
</tr>
<tr>
<td>24</td>
<td>Routine to let user select a window using a mouse. Called by Select_Window_With_Argument_List when no arguments are present in the command line.</td>
</tr>
</tbody>
</table>
# Appendix I -- Module Names, Experiment 3

<table>
<thead>
<tr>
<th>Meaningful Name</th>
<th>Neutral Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display_Usage_Syntax</td>
<td>STD_Process_Argument</td>
</tr>
<tr>
<td>Look_Up_In_Table</td>
<td>Wind_14_12_Meth1</td>
</tr>
<tr>
<td>Print_Id_Error_And_Quit</td>
<td>Method_AX_Wind_145_Info</td>
</tr>
<tr>
<td>Display_Id_And_Name</td>
<td>Routine_23_XFD_Wind</td>
</tr>
<tr>
<td>Display_Stats_Info</td>
<td>Routine_XWind_99XF</td>
</tr>
<tr>
<td>Display_Pixel_Info</td>
<td>Process_Terml_Data</td>
</tr>
<tr>
<td>Show_Events_In_Mask</td>
<td>Wind_Method_AU_Info</td>
</tr>
<tr>
<td>Display_All_Events_Info</td>
<td>Window_X11_System_Proc</td>
</tr>
<tr>
<td>Display_Tree_Info</td>
<td>Window_Proc_A1124</td>
</tr>
<tr>
<td>Show_Size_Hints</td>
<td>Proc_12XX_XWind</td>
</tr>
<tr>
<td>Display_Normal_Or_Zoom_Size_Hints</td>
<td>X_Window_Proced_X1_Wind_Info_Proc</td>
</tr>
<tr>
<td>Display_Window_Manager_Hints</td>
<td>X_Window_Method_Display_Term</td>
</tr>
<tr>
<td>Print_Error_Message_And_Qui t</td>
<td>X_Window_ID20_System_Routine</td>
</tr>
<tr>
<td>Get_Display_Name_From_Argument_List</td>
<td>Procedure_X_Window_Termnl_Procedure</td>
</tr>
<tr>
<td>Open_Legal_Display</td>
<td>Proc_Systm_Process</td>
</tr>
<tr>
<td>Select_Display_With_Argument_List</td>
<td>Argument_Two_Process_System311_AU</td>
</tr>
<tr>
<td>Open_Font</td>
<td>Proc_XW13</td>
</tr>
<tr>
<td>Beep_Display</td>
<td>Proc_Win0133</td>
</tr>
<tr>
<td>Locate_Window</td>
<td>Systm_X_W_Dpy</td>
</tr>
<tr>
<td>Select_Window_With_Argument_List</td>
<td>Systm_Routine_3451_Argument_Proc</td>
</tr>
<tr>
<td>Convert_Color_to_Number</td>
<td>Long_X11_T2_Window_Proc</td>
</tr>
<tr>
<td>Print_Debug_Message</td>
<td>_Term_Xwind_Rout311</td>
</tr>
<tr>
<td>Print_Blip_Message</td>
<td>MT_Wind_PR_Routine</td>
</tr>
<tr>
<td>Select_Window_With_Mouse</td>
<td>PR_X11_Window_Routine311</td>
</tr>
</tbody>
</table>
Appendix J -- Hypothesis Questions, Experiment 3

<table>
<thead>
<tr>
<th>Module</th>
<th>Hypothesis Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Function Display_Usage_Syntax is used to show the correct way to call xwininfo. Function STD_Process_Argument is used to show the correct way to call xwininfo.</td>
</tr>
<tr>
<td>2</td>
<td>Function Look_Up_In_Table is used to enter a value into an array Function Wind_14_12_Meth1 is used to enter a value into an array</td>
</tr>
<tr>
<td>3</td>
<td>Function Print_Id_Error_And_Qui is used to display an error message and then exit the program. Function Method_AX_Wind_145_Info is used to display an error message and then exit the program.</td>
</tr>
<tr>
<td>4</td>
<td>Function Display_Id_And_Name is used to change a window's id number Function Routine_23_XFD_Wind is used to change a window's id number</td>
</tr>
<tr>
<td>5</td>
<td>Function Display_Spats_Info is used to show data about a window's location and size. Function Routine_XWind_99XF is used to show data about a window's location and size.</td>
</tr>
<tr>
<td>6</td>
<td>Function Display_Pixel_Info is used to show data about a display's location. Function Process_Terml_Data is used to show data about a display's location.</td>
</tr>
<tr>
<td>7</td>
<td>Function Show_Events_In_Mask is used to change the status of XWindow's event queue. Function Wind_Method_AU_Info is used to change the status of XWindow's event queue.</td>
</tr>
<tr>
<td>8</td>
<td>Function Display_All_Events_Info is used to show which windows are active for a given display. Function Window_X11_System_Proc is used to show which windows are active for a given display.</td>
</tr>
</tbody>
</table>
### Appendix J -- Hypothesis Questions, Experiment 3 (Cont.)

<table>
<thead>
<tr>
<th>Module</th>
<th>Hypothesis Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Function Display_Tree_Info is used to show a window’s root, parent, and children ID #’s. Function Window_Proc_A1124 is used to show a window’s root, parent, and children ID #’s.</td>
</tr>
<tr>
<td>10</td>
<td>Function Show_Size_Hints is used to reposition a window on a display. Function Proc_12XX_XWind is used to reposition a window on a display.</td>
</tr>
<tr>
<td>11</td>
<td>Function Display_Normal_Or_Zoom_Size_Hints is used to show which events are selected for the window. Function X_Window_Proced_X1_Wind_Info_Proc is used to show which events are selected for the window.</td>
</tr>
<tr>
<td>12</td>
<td>Function Display_Window_Manager_Hints is used to show information about a window’s position on the display. Function X_Window_Method_Display_Term is used to show information about a window’s position on the display.</td>
</tr>
<tr>
<td>13</td>
<td>Function Print_Error_Message_And_Quit is used to handle fatal errors. Function X_Window_ID20_System_Routine is used to handle fatal errors.</td>
</tr>
<tr>
<td>14</td>
<td>Function Get_Display_Name_From_Argument_List is used to read the name of the display entered by the user. Function Procedure_X_Window_Termnl_Procedure is used to read the name of the display entered by the user.</td>
</tr>
<tr>
<td>15</td>
<td>Function Open_Legal_Display is used to return the name of the display currently open. Function Proc_Systm_Process is used to return the name of the display currently open.</td>
</tr>
<tr>
<td>16</td>
<td>Function Select_Display_With_Argument_List is used to open the display specified in the command line. Function Argument_Two_Process_System311_AU is used to open the display specified in the command line.</td>
</tr>
</tbody>
</table>
## Appendix J -- Hypothesis Questions, Experiment 3 (Cont.)

<table>
<thead>
<tr>
<th>Module</th>
<th>Hypothesis Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>Function Open_Font is used to load a font or, if unable, to print an error message. Function Proc_XW13 is used to load a font or, if unable, to print an error message.</td>
</tr>
<tr>
<td>18</td>
<td>Function Beep_Display is used to close the display. Function Proc_Win0133 is used to close the display.</td>
</tr>
<tr>
<td>19</td>
<td>Function Locate_Window is used to return a pointer to a window, given its name and display. Function Systm_X_W_Dpy is used to return a pointer to a window, given its name and display.</td>
</tr>
<tr>
<td>20</td>
<td>Function Select_Window_With_Argument_List is used to let the user resize a window. Function Systm_Routine_3451_Argument_Proc is used to let the user resize a window.</td>
</tr>
<tr>
<td>21</td>
<td>Function Convert_Color_To_Number is used to return the pixel # which produces the specified color. Function Long_X11_T2_Window_Proc is used to return the pixel # which produces the specified color.</td>
</tr>
<tr>
<td>22</td>
<td>Function Print_Debug_Message is used to display a message on stderr. Function _Term_Xwind_Rout311 is used to display a message on stderr.</td>
</tr>
<tr>
<td>23</td>
<td>Function Print_Blip_Message is used to cause the borders of a window to blink on and off. Function MT_Wind_PR_Routine is used to cause the borders of a window to blink on and off.</td>
</tr>
<tr>
<td>24</td>
<td>Function Select_Window_With_Mouse is used to let the user choose a window using the mouse. Function PR_X11_Window_Routine311 is used to let the user choose a window using the mouse.</td>
</tr>
</tbody>
</table>
Appendix K-- Location Questions, Experiment 3

<table>
<thead>
<tr>
<th>Module</th>
<th>Location Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Locate the routine which displays the options available to the user when calling xwininfo?</td>
</tr>
<tr>
<td>2</td>
<td>Locate the routine which returns a text string obtained from an array?</td>
</tr>
<tr>
<td>3</td>
<td>Locate the routine which is used to process errors dealing with invalid window ID numbers</td>
</tr>
<tr>
<td>4</td>
<td>Locate the routine which is used to display a window’s id number and name (if the window has one)?</td>
</tr>
<tr>
<td>5</td>
<td>Locate the routine which is used to display information about a window’s location, size, and class?</td>
</tr>
<tr>
<td>6</td>
<td>Locate the routine which is used to display information about a window’s gravity and backing store attributes?</td>
</tr>
<tr>
<td>7</td>
<td>Locate the routine which is used to display the events in one specific event mask passed in as a parameter?</td>
</tr>
<tr>
<td>8</td>
<td>Locate the routine which is used to display both the events selected and the events not propagated for a window?</td>
</tr>
<tr>
<td>9</td>
<td>Locate the routine which is used to display the id numbers for a window’s root, parent, and children windows?</td>
</tr>
<tr>
<td>10</td>
<td>Locate the routine which is used to display one set of size hints?</td>
</tr>
<tr>
<td>11</td>
<td>Locate the routine which is used to show the normal and zoom size hints for a window?</td>
</tr>
<tr>
<td>12</td>
<td>Locate the routine which is used to display the window manager hints and icon information?</td>
</tr>
<tr>
<td>13</td>
<td>Locate the routine which is used to terminate the program when fatal errors are encountered?</td>
</tr>
<tr>
<td>14</td>
<td>Locate the routine which is used to find the name of a display from xwininfo’s command line argument list?</td>
</tr>
</tbody>
</table>
## Appendix K-- Location Questions, Experiment 3 (Cont.)

<table>
<thead>
<tr>
<th>Module</th>
<th>Location Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Locate the routine which is used to open a display, given the display’s name?</td>
</tr>
<tr>
<td>16</td>
<td>Locate the routine which is used to open a display and its default screen?</td>
</tr>
<tr>
<td>17</td>
<td>Locate the routine which is used to load a font?</td>
</tr>
<tr>
<td>18</td>
<td>Locate the routine which is used to produce a noise on the display?</td>
</tr>
<tr>
<td>19</td>
<td>Locate the routine which is used to locate a window on a display, given the window’s name?</td>
</tr>
<tr>
<td>20</td>
<td>Locate the routine which is used to get the window specified on xwininfo’s command line argument list?</td>
</tr>
<tr>
<td>21</td>
<td>Locate the routine which is used to return the number, which when used in the window will be of the specified color?</td>
</tr>
<tr>
<td>22</td>
<td>Locate the routine which is used to print debugging messages and then continue execution of the program?</td>
</tr>
<tr>
<td>23</td>
<td>Locate the routine which is used to print blip! on the display?</td>
</tr>
<tr>
<td>24</td>
<td>Locate the routine which is used to find the window selected with a mouse?</td>
</tr>
</tbody>
</table>
Appendix L

Incorrect Responses to Hypothesis Questions, Experiment 3

Number of Incorrect Responses (0..3)

<table>
<thead>
<tr>
<th>Subject</th>
<th>1 (n)</th>
<th>2 (p)</th>
<th>3 (t)</th>
<th>4 (n,p)</th>
<th>5 (n,t)</th>
<th>6 (p,t)</th>
<th>7 (n,p,t)</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>total</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>13</td>
</tr>
</tbody>
</table>

Chi-Square Tests Of Differences

<table>
<thead>
<tr>
<th>Independent Variable</th>
<th>$\chi^2$</th>
<th>df</th>
<th>N</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Name</td>
<td>6.68</td>
<td>1</td>
<td>192</td>
<td>p=.01</td>
</tr>
<tr>
<td>Preview Statement</td>
<td>2.06</td>
<td>1</td>
<td>192</td>
<td>p=.15</td>
</tr>
<tr>
<td>Typographical Signaling</td>
<td>0.08</td>
<td>1</td>
<td>192</td>
<td>p=.96</td>
</tr>
</tbody>
</table>
Appendix M

Confidence Ratings for Hypothesis Questions, Experiment 3

Confidence Rating Data per Subject (0..300)

<table>
<thead>
<tr>
<th>Subject</th>
<th>1 (n)</th>
<th>2 (p)</th>
<th>3 (t)</th>
<th>4 (n)</th>
<th>5 (n,p)</th>
<th>6 (n,t)</th>
<th>7 (p,t)</th>
<th>8 (n,p,t)</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>299</td>
<td>300</td>
<td>300</td>
<td>299</td>
<td>293</td>
<td>300</td>
<td>290</td>
<td>290</td>
<td>296.4</td>
</tr>
<tr>
<td>2</td>
<td>230</td>
<td>275</td>
<td>290</td>
<td>270</td>
<td>285</td>
<td>295</td>
<td>300</td>
<td>255</td>
<td>275.0</td>
</tr>
<tr>
<td>3</td>
<td>235</td>
<td>275</td>
<td>300</td>
<td>230</td>
<td>185</td>
<td>280</td>
<td>295</td>
<td>270</td>
<td>258.8</td>
</tr>
<tr>
<td>4</td>
<td>290</td>
<td>270</td>
<td>300</td>
<td>270</td>
<td>285</td>
<td>270</td>
<td>300</td>
<td>285</td>
<td>283.8</td>
</tr>
<tr>
<td>5</td>
<td>300</td>
<td>250</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>270</td>
<td>300</td>
<td>300</td>
<td>290.0</td>
</tr>
<tr>
<td>6</td>
<td>255</td>
<td>270</td>
<td>290</td>
<td>255</td>
<td>297</td>
<td>275</td>
<td>275</td>
<td>295</td>
<td>276.5</td>
</tr>
<tr>
<td>7</td>
<td>270</td>
<td>255</td>
<td>265</td>
<td>285</td>
<td>285</td>
<td>285</td>
<td>270</td>
<td>285</td>
<td>275.0</td>
</tr>
<tr>
<td>8</td>
<td>242</td>
<td>237</td>
<td>110</td>
<td>286</td>
<td>209</td>
<td>266</td>
<td>286</td>
<td>264</td>
<td>237.5</td>
</tr>
<tr>
<td>mean</td>
<td>265.1</td>
<td>266.5</td>
<td>269.4</td>
<td>274.4</td>
<td>267.4</td>
<td>280.1</td>
<td>289.5</td>
<td>280.5</td>
<td>274.1</td>
</tr>
</tbody>
</table>
Confidence Ratings for Hypothesis Questions, Experiment 3

Multifactor Analysis of Variance

<table>
<thead>
<tr>
<th>SOURCE</th>
<th>df</th>
<th>SS</th>
<th>MS</th>
<th>F</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Effects</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>15</td>
<td>15</td>
<td>0.019</td>
<td>.8937</td>
</tr>
<tr>
<td>P</td>
<td>1</td>
<td>425</td>
<td>425</td>
<td>0.525</td>
<td>.4798</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>3150</td>
<td>3150</td>
<td>3.889</td>
<td>.0542</td>
</tr>
<tr>
<td>Subject</td>
<td>7</td>
<td>19397</td>
<td>2771</td>
<td>3.420</td>
<td>.0046</td>
</tr>
<tr>
<td>First-Order Interactions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NxP</td>
<td>1</td>
<td>329</td>
<td>329</td>
<td>0.405</td>
<td>.5340</td>
</tr>
<tr>
<td>NxT</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>0.009</td>
<td>.9279</td>
</tr>
<tr>
<td>PxT</td>
<td>1</td>
<td>108</td>
<td>108</td>
<td>0.133</td>
<td>.7209</td>
</tr>
<tr>
<td>Residual</td>
<td>50</td>
<td>40511</td>
<td>810</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>63</td>
<td>63942</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Factors:  
N = Module Name  
P = Preview Statement  
T = Typographic Signaling
Appendix N

Response Times to Hypothesis Questions, Experiment 3

Response Time Data (sec)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>235.5</td>
<td>175.8</td>
<td>172.6</td>
<td>129.9</td>
<td>274.4</td>
<td>207.4</td>
<td>126.8</td>
<td>203.3</td>
<td>190.7</td>
</tr>
<tr>
<td>2</td>
<td>519.3</td>
<td>315.2</td>
<td>381.8</td>
<td>318.1</td>
<td>207.6</td>
<td>356.1</td>
<td>227.9</td>
<td>343.6</td>
<td>333.7</td>
</tr>
<tr>
<td>3</td>
<td>224.3</td>
<td>171.2</td>
<td>129.4</td>
<td>137.0</td>
<td>94.2</td>
<td>94.6</td>
<td>170.3</td>
<td>182.8</td>
<td>150.5</td>
</tr>
<tr>
<td>4</td>
<td>257.2</td>
<td>219.0</td>
<td>215.2</td>
<td>200.4</td>
<td>114.0</td>
<td>240.1</td>
<td>121.3</td>
<td>68.0</td>
<td>179.4</td>
</tr>
<tr>
<td>5</td>
<td>330.6</td>
<td>195.0</td>
<td>130.8</td>
<td>226.6</td>
<td>65.8</td>
<td>86.8</td>
<td>84.4</td>
<td>81.3</td>
<td>150.2</td>
</tr>
<tr>
<td>6</td>
<td>175.6</td>
<td>119.5</td>
<td>119.7</td>
<td>187.7</td>
<td>138.7</td>
<td>53.9</td>
<td>102.3</td>
<td>116.3</td>
<td>126.7</td>
</tr>
<tr>
<td>7</td>
<td>144.5</td>
<td>270.9</td>
<td>74.3</td>
<td>184.9</td>
<td>132.1</td>
<td>90.1</td>
<td>201.3</td>
<td>59.4</td>
<td>144.7</td>
</tr>
<tr>
<td>8</td>
<td>113.7</td>
<td>63.5</td>
<td>171.5</td>
<td>74.5</td>
<td>62.8</td>
<td>59.4</td>
<td>97.8</td>
<td>37.3</td>
<td>85.1</td>
</tr>
<tr>
<td>mean</td>
<td>250.1</td>
<td>191.3</td>
<td>174.4</td>
<td>182.4</td>
<td>136.2</td>
<td>148.6</td>
<td>141.5</td>
<td>136.5</td>
<td>170.1</td>
</tr>
</tbody>
</table>
Appendix N (Cont.)

Response Times to Hypothesis Questions, Experiment 3

Multifactor Analysis of Variance -- LOG(sec)

<table>
<thead>
<tr>
<th>SOURCE</th>
<th>df</th>
<th>SS</th>
<th>MS</th>
<th>F</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Effects</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>1.1651</td>
<td>1.1651</td>
<td>8.531</td>
<td>.0052</td>
</tr>
<tr>
<td>P</td>
<td>1</td>
<td>1.1338</td>
<td>1.1338</td>
<td>8.302</td>
<td>.0058</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>0.8829</td>
<td>0.8829</td>
<td>6.465</td>
<td>.0141</td>
</tr>
<tr>
<td>Subject</td>
<td>7</td>
<td>9.3841</td>
<td>1.3406</td>
<td>9.816</td>
<td>.0000</td>
</tr>
<tr>
<td>First-Order Interactions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NxP</td>
<td>1</td>
<td>0.0178</td>
<td>0.0178</td>
<td>0.130</td>
<td>.7236</td>
</tr>
<tr>
<td>NxT</td>
<td>1</td>
<td>0.0016</td>
<td>0.0016</td>
<td>0.012</td>
<td>.9141</td>
</tr>
<tr>
<td>PxT</td>
<td>1</td>
<td>0.1486</td>
<td>0.1486</td>
<td>1.088</td>
<td>.3019</td>
</tr>
<tr>
<td>Residual</td>
<td>50</td>
<td>6.8285</td>
<td>0.1366</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>63</td>
<td>19.5625</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Factors:  
N = Module Name  
P = Preview Statement  
T = Typographic Signaling
Appendix O

Incorrect Answers to Location Questions, Experiment 3

Incorrect Answers (0..3)

<table>
<thead>
<tr>
<th>Subject</th>
<th>1(n)</th>
<th>2(p)</th>
<th>3(t)</th>
<th>4(n,p)</th>
<th>5(n,t)</th>
<th>6(p,t)</th>
<th>7(n,p,t)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>total</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>36</td>
</tr>
</tbody>
</table>

Chi-Square Tests Of Differences

<table>
<thead>
<tr>
<th>Independent Variable</th>
<th>$\chi^2$</th>
<th>df</th>
<th>N</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module Name</td>
<td>0.14</td>
<td>1</td>
<td>192</td>
<td>p=.71</td>
</tr>
<tr>
<td>Preview Statement</td>
<td>2.19</td>
<td>1</td>
<td>192</td>
<td>P=.14</td>
</tr>
<tr>
<td>Typographical Signaling</td>
<td>1.23</td>
<td>1</td>
<td>192</td>
<td>p=.27</td>
</tr>
</tbody>
</table>
Appendix P

Types of Errors to Location Questions, Experiment 3

<table>
<thead>
<tr>
<th>Subject</th>
<th>1 (n)</th>
<th>2 (p)</th>
<th>3 (t)</th>
<th>4 (n,p)</th>
<th>5 (n,t)</th>
<th>6 (p,t)</th>
<th>7 (n,p,t)</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>total</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>36</td>
</tr>
</tbody>
</table>

Chi Square Tests Of Differences

<table>
<thead>
<tr>
<th>Independent Variable</th>
<th>$\chi^2$</th>
<th>df</th>
<th>N</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module Name</td>
<td>2.18</td>
<td>1</td>
<td>192</td>
<td>$p &gt; .05$</td>
</tr>
<tr>
<td>Preview Statement</td>
<td>0.55</td>
<td>1</td>
<td>192</td>
<td>$P &gt; .05$</td>
</tr>
<tr>
<td>Typographical Signaling</td>
<td>0.55</td>
<td>1</td>
<td>192</td>
<td>$p &gt; .05$</td>
</tr>
</tbody>
</table>
Appendix Q

Response Times to Location Questions, Experiment 3

Response Time Data (sec)

<table>
<thead>
<tr>
<th>Subject</th>
<th>1 (n)</th>
<th>2 (p)</th>
<th>3 (t)</th>
<th>4 (n,p)</th>
<th>5 (n,t)</th>
<th>6 (p,t)</th>
<th>7 (n,p,t)</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>93.8</td>
<td>66.8</td>
<td>102.8</td>
<td>149.8</td>
<td>107.5</td>
<td>54.0</td>
<td>79.6</td>
<td>54.3</td>
</tr>
<tr>
<td>2</td>
<td>591.6</td>
<td>365.8</td>
<td>407.5</td>
<td>386.6</td>
<td>298.5</td>
<td>40.4</td>
<td>43.7</td>
<td>155.8</td>
</tr>
<tr>
<td>3</td>
<td>413.3</td>
<td>78.9</td>
<td>69.0</td>
<td>160.7</td>
<td>55.8</td>
<td>174.8</td>
<td>61.5</td>
<td>101.4</td>
</tr>
<tr>
<td>4</td>
<td>166.8</td>
<td>194.1</td>
<td>122.5</td>
<td>482.0</td>
<td>323.3</td>
<td>517.0</td>
<td>138.5</td>
<td>341.4</td>
</tr>
<tr>
<td>5</td>
<td>104.5</td>
<td>90.5</td>
<td>86.4</td>
<td>163.3</td>
<td>39.5</td>
<td>49.1</td>
<td>53.7</td>
<td>70.2</td>
</tr>
<tr>
<td>6</td>
<td>392.0</td>
<td>229.2</td>
<td>273.9</td>
<td>195.1</td>
<td>69.3</td>
<td>252.9</td>
<td>83.1</td>
<td>286.7</td>
</tr>
<tr>
<td>7</td>
<td>44.3</td>
<td>93.1</td>
<td>206.8</td>
<td>352.7</td>
<td>40.3</td>
<td>27.0</td>
<td>182.6</td>
<td>26.4</td>
</tr>
<tr>
<td>8</td>
<td>113.3</td>
<td>205.4</td>
<td>53.7</td>
<td>105.9</td>
<td>59.9</td>
<td>69.9</td>
<td>73.3</td>
<td>51.1</td>
</tr>
<tr>
<td>mean</td>
<td>239.9</td>
<td>165.5</td>
<td>165.3</td>
<td>249.5</td>
<td>124.2</td>
<td>148.1</td>
<td>89.5</td>
<td>135.9</td>
</tr>
</tbody>
</table>
Appendix Q (Cont.)

Response Times to Location Questions, Experiment 3

Multifactor Analysis of Variance -- LOG(sec)

<table>
<thead>
<tr>
<th>SOURCE</th>
<th>df</th>
<th>SS</th>
<th>MS</th>
<th>F</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Effects</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>1.6928</td>
<td>1.6928</td>
<td>4.030</td>
<td>.0501</td>
</tr>
<tr>
<td>P</td>
<td>1</td>
<td>2.7481</td>
<td>2.7481</td>
<td>6.543</td>
<td>.0136</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>0.3578</td>
<td>0.3578</td>
<td>0.852</td>
<td>.3703</td>
</tr>
<tr>
<td>Subject</td>
<td>7</td>
<td>13.7157</td>
<td>1.9594</td>
<td>4.665</td>
<td>.0004</td>
</tr>
<tr>
<td>First-Order Interactions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NxP</td>
<td>1</td>
<td>0.7723</td>
<td>0.7723</td>
<td>1.839</td>
<td>.1812</td>
</tr>
<tr>
<td>NxT</td>
<td>1</td>
<td>0.0074</td>
<td>0.0074</td>
<td>0.018</td>
<td>.8960</td>
</tr>
<tr>
<td>PxT</td>
<td>1</td>
<td>0.0439</td>
<td>0.0439</td>
<td>0.104</td>
<td>.7513</td>
</tr>
<tr>
<td>Residual</td>
<td>50</td>
<td>21.0010</td>
<td>0.4200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>63</td>
<td>40.3390</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Factors:  
N = Module Name  
P = Preview Statement  
T = Typographic Signaling