Effective and Appropriate Use of Controlled Experimentation in Software Development Research

Mark Alan Johnson
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Computer Sciences Commons

Let us know how access to this document benefits you.

Recommended Citation

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.
THESIS APPROVAL

The abstract and thesis of Mark Alan Johnson for the Master of Science degree in Computer Science were presented October 28, 1996 and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

Warren Harrison, Chair

Richard Hamlet

David M. Raffo
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL:

John McHugh, Chair

Department of Computer Science

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by ______________________ on 11-6-96
ABSTRACT

An abstract of the thesis of Mark Alan Johnson for the Master of Science in Computer Science presented October 28, 1996.

Title: Effective and Appropriate Use of Controlled Experimentation in Software Development Research.

Although there is a large body of research and publication on software development, only a limited amount of this work includes empirical demonstration of its effectiveness. Yet, it is this empirical understanding which will help move software development from a craft to an engineering discipline. Of the empirical methods for research, controlled experiments are the most commonly thought of in scientific studies, and yet the least used to study software development.

This thesis begins with a brief review of the different empirical methods commonly used to study software development. This review provides a quick introduction to each empirical method, compares the main advantages and weaknesses of each method, and provides a context for how controlled experimentation compares to other empirical methods for studying software development.
Using empirical methods to study software development is not easy or straightforward. There are limitations which appear to be inherent in the nature of software and issues due to the improper understanding or application of empirical methods. These limitations and issues are identified, specifically for controlled experiments, and approaches for dealing with them are proposed.

A controlled experiment was designed and conducted to demonstrate the method and explore the limitations and issues for empirical research in software development. This experiment and its results are presented. This example experiment demonstrates that conducting even a simple experiment in software development is challenging. Lessons learned from this experience are reported.

Controlled experiments require that the researcher have a high degree of control over the environment where the experiment is carried out. This can be costly and difficult to achieve. This thesis concludes by discussing how controlled experiments can be used effectively in studies of software development.
EFFECTIVE AND APPROPRIATE USE OF CONTROLLED EXPERIMENTATION IN SOFTWARE DEVELOPMENT RESEARCH

by

MARK ALAN JOHNSON

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE in
COMPUTER SCIENCE

Portland State University
1996
# TABLE OF CONTENTS

## CHAPTER

1. INTRODUCTION ........................................................................................................... 1

2. EMPIRICAL METHODS FOR SOFTWARE DEVELOPMENT RESEARCH .................................. 7
   2.1 Anecdotal Reports .................................................................................................... 10
   2.2 Surveys .................................................................................................................. 11
   2.3 Case Studies ........................................................................................................... 15
   2.4 Quantitative Modeling with Simulation .................................................................... 18
   2.5 Controlled Experiments ........................................................................................... 21

3. ISSUES WITH CONTROLLED EXPERIMENTS IN SOFTWARE ............................................ 24
   3.1 Limitations due to the Nature of Software Development .......................................... 26
   3.2 Issues in using Controlled Experiments ................................................................... 37

4. AN EXAMPLE EXPERIMENT ......................................................................................... 48
   4.1 The Experiment ....................................................................................................... 48
   4.2 Comparison to the Limitations and Issues ................................................................ 74
   4.3 Lessons Learned ...................................................................................................... 78

5. EFFECTIVE USE OF CONTROLLED EXPERIMENTS IN SOFTWARE ................................. 81
   5.1 Studies Appropriate for Controlled Experiments ..................................................... 82
   5.2 Effective Studies using Student Subjects ................................................................... 85
   5.3 Improving ‘Cost-Effectiveness’ and Generalization .................................................... 89
   5.4 Summary .................................................................................................................. 92

BIBLIOGRAPHY ............................................................................................................. 94

Controlled Experimentation in Software Development .................................................. 94
Related References ........................................................................................................ 102

APPENDIX

A: REFERENCES FOR CONTROLLED EXPERIMENTS .............. 106

B: MATERIALS AND DATA FROM THE EXAMPLE EXPERIMENT ...................................... 109
LIST OF FIGURES

Figure 1: Cognitive Paradigms from Curtis [25].................................................. 37
Figure 2: Word Count Measures for Program Listings................................. 57
Figure 3: Distribution of Time to Complete Questionnaire............................ 65
Figure 4: Distribution of Scores on Questionnaire.......................................... 66
Figure 5: Students' Background Summary...................................................... 68
Figure 6: Averages for GPA................................................................................. 69
Figure 7: Average Number of Programs Written............................................. 69
Figure 8: Average Score on Questionnaire.................................................... 70
Figure 9: Average Time to Complete Questionnaire...................................... 71
Figure 10: Combined Sections, Average Time and Score, Students
           Ranked 4 or 5 on CS Classes or Programs Excluded............................ 72
Figure 11: Non-Function Style Program.......................................................... 110
Figure 12: Function Style Program................................................................. 111
Figure 13: Informed Consent Form.................................................................. 112
Figure 14: Experiment Questionnaire.............................................................. 113
Figure 15: Background Questions and Instructions....................................... 114
Figure 16: Researcher's Instructions Script.................................................... 115
Figure 17: Survey Data Set (part 1): Background Information....................... 116
Figure 17: Survey Data Set (Part 2): Questionnaire Results......................... 117
1. INTRODUCTION

How can an engineer or manager find believable information regarding development processes and models, methods, tools, product and process metrics, and all the other software development subjects? As one progresses through their career in software development, one develops personal techniques. For instance:

- **Rules of Thumb from Personal Experience**: When trying to deal with the ambiguities of work in software development, engineers and managers develop rules of thumb, like “The testing cycle for a maintenance release always takes three months from engineering release to shipment. It doesn’t seem to matter what the original schedule said.” These are specific to particular sets of circumstances and environments, but serve a useful purpose.

- **Consult with Colleagues**: Another technique learned is which coworkers seem to be able to pick the ‘right’ new technologies or methodologies from the current literature. Through the trial and error process of listening to advice from different people, then getting burned or getting lucky, engineers develop a model of ‘who to listen
to. Unfortunately this is like the financial investments caution: “Past performance is no guarantee of future success.”

- **Sales Materials and Salespersons:** Although the information received often seems firm and quantitative, it is in fact highly biased. Since the point of sales is to sell things, the best one can hope for is that positive information about a product is true. Negative information is more likely to be accurate about a competitor’s products than the salesman’s own.

- **Scanning the Software Development Literature:** The ideas presented in the literature sound nice, but few have hard evidence of actual use. If there is an example of use it is frequently a small demonstration project, often carried out at an academic or corporate research lab. In a recent personal communication (Dalio [27]), the observation was made that ideas in software development seem to have a life of about three years. In the first year, an idea is initially reported and articles start to appear describing it and why it is good. In the second year more articles are published, and it seems that everyone is trying this idea. Then in the third year, a few people are still trying to keep the idea alive, but much of the focus in the literature has moved on to other, newer ideas. In a few cases such as formal inspections, ideas live on, but most have become potential subjects for a “What ever happened to?” column in the trade press.
Unfortunately, none of these methods for gaining information produce consistent or quantified results.

Compare this state of affairs with the electronics field: At the higher levels are general circuit analysis rules and numeric device specifications that allow quick, ball-park estimates for fan-in or fan-out loading of an integrated circuit input or output pin. At a more detailed level, there are models of transistor operation that allow calculation of the flow of holes and electrons across junction boundaries. And if needed for full understanding, device physics can be studied at the quantum mechanics level.

Certainly, the hardware development world isn't perfect, but how is it that it seems so much more orderly? How has the information used in hardware development become quantitative?

The answer is through a lot of empirical groundwork. By developing a quantitative model or theory based on observation, then doing empirical studies to confirm the model, and ultimately understanding the application and limit of that model, hardware engineering has built a solid base of quantitative information at multiple levels of abstraction. This empirical work collects, analyzes, and interprets observations – to paraphrase Tichy et al. [49].
In the relatively young field of software development, there is a much smaller body of modeling and empirical research. Solid work has been done using modeling and empirical studies for the technical areas of computer science such as algorithms, compilers, operating systems, and computer architectures. However, in the areas relating to how software is developed, much of the published research findings appear to be 'analytical advocacy.' This is defined by Fenton et al. [30] as "if sufficient brilliance and analysis were put into conceiving a technique benefit would surely follow."

As found by Tichy et al. [49], much of the time there is no empirical support for published software research. In 1995, they conducted a survey of research articles in computer science. They found that over 40% of the articles discussing design and modeling methods have no empirically validated results. They found so few articles reporting controlled experiments that they lumped them in with reports of other empirical studies. The resulting category was a bare 10% of the articles.

There are a number of different methods of empirical research which are in use for studying software development. While these methods are outlined in chapter 2, the most seldom used is for me the most intriguing. The controlled experiment involves setting up an environment where many aspects of that environment are held constant while the researcher varies
one or a few stimuli and measures to see if this results in a change. The key advantage to this method of empirical study is that a fairly direct cause and effect relationship can be established, if the researcher has good control of the environment and makes the proper measurements. This is why in general scientific research, "Experiments are, by far, the most common and commonly thought of empirical evaluation method" as stated by Adelman [17].

If controlled experimentation is a method frequently used in other areas of science and engineering, and it provides the most direct cause and effect understanding, why is it so little used for studying software development? How can a working engineer or manager know if the result reported from an experiment is reasonable or applicable to them? How might they go about running their own experiments?

To answer these questions, I reviewed the various empirical methods in general, and controlled experiments in particular. Then I designed a controlled experiment to reinforce my understanding of the method and to try and answer a simple question in teaching software development. After conducting the experiment, I found that even a simple experiment in software development is challenging. This document describes:
• The common types of empirical methods used in studying software development.

• A discussion of the limitations and issues of using controlled experiments in software development studies.

• My experience running a controlled experiment.

• My conclusions concerning the effective use of controlled experimentation in studying software development.
2. **EMPIRICAL METHODS FOR SOFTWARE DEVELOPMENT RESEARCH**

This chapter briefly reviews several empirical methods which can be used to study software development. This review provides a quick introduction to each empirical method, compares the main advantages and weaknesses of each method, and provides a context for how controlled experimentation compares to other empirical methods for studying software development.

Hale et al. [32] define empirical research as "research in which data about software ... is collected and evaluated." The distinguishing characteristic of empirical methods is the focus on the collection and analysis of data from observations. This is in contrast to analytical methods where the focus is on logical evaluation independent of specific activities.

Studies using empirical methods can be grouped into three general classes:

- **Examples and Experience reports** are generally a single use of a method, tool or model by a single group. An example is a situation
created or selected specifically to demonstrate how a method, tool or model works or might be applied. An experience report describes an instance of usage of a method, tool or model and the results achieved by this usage. Both types of studies are focused on enhancing the understanding of an idea and its application.

- **Information gathering studies** are used to collect and analyze information to be able to describe or model some aspect of software development. Information is gathered about the specific aspect of software development being studied, and about the particular environment(s) where the study is conducted. This allows the researcher to make a statement on the aspect of software development they were studying, and to qualify under what circumstances this statement should be valid.

- **Comparative studies** are designed to compare two or more alternative methods or tools, or to compare the predictions of a theory or model to an actual situation in software development. In a comparative study, a first measurement, called the baseline, is taken before any change is made. Then a change is made in the environment, and the performance of the subjects is again measured. The goal is be able to establish that any change in the measurements is due to the controlled change made in the environment. A control serves the same purpose
as a baseline, but instead of measuring the same subjects before and after, a separate set of subjects is measured, using the original environment, while another group uses the changed environment.

There are a number of different empirical methods for collecting and evaluating quantitative information in software development research.

- **Surveys:** The collection of a large set of data on a software development topic from a wide variety of sources. Surveys are generally used for information gathering studies, although they can also be used for comparative studies.

- **Case Studies:** The trial of some change in a typical software development environment. Frequently, case studies are single instances of usage of an idea, so they are experience reports or examples of use. They can also be designed to gather information or do comparisons of ideas.

- **Modeling with Simulation:** Creating a quantitative model of a software development environment and simulating changes in the environment by changing values used in the model. Models with simulation are generally used for comparative studies (alternatives) but can also be used for describing one aspect of software development in terms of how that aspect is effected by variations in other aspects.
• **Controlled Experiments**: Systematically applying changes in a tightly controlled environment and observing for effects which can be statistically tied back to the changes. Controlled experiments are classically used for comparative studies. They can also be used for information gathering, especially for building descriptive models.

The key difference between the categories of empirical method is "the degree of control the researcher exerts over conditions that influence the objects he or she is studying" as stated by Hale et al. [32]. This control is the ability to control factors which could affect the performance of the subjects of the study.

2.1 **Anecdotal Reports**

Anecdotal studies or reports are a limited use of one of the empirical methods. Adelman [17] describes these as a "one-group posttest-only study." These are studies where only one group of subjects is studied, and they are studied using only a single method, tool, or measurement. They can be effective as an example of use, or as a report of experience.

An anecdotal report becomes a problem when the purpose of a study is information gathering or comparison. As an information gathering study, the limited exposure of an anecdotal study means that the only statement that
can be made about the results is “In this particular case, with these particular subjects, this particular result was achieved.” As a comparative study, an anecdotal report lacks a baseline or control, so there is no way to justify any claim that a method, tool or measurement is “better” or “different.”

Anecdotal reports are common in the software literature because many new ideas are continually being developed, and new factors identified which could be measured. For each of these new ideas, it is easiest to do a single example of use to demonstrate the idea. However, these single examples have limited value in broadening our understanding of software development. They need to be followed up with extended studies to develop a solid base of understanding of how these new ideas fit in with existing knowledge of software development.

2.2 Surveys

Surveys are defined as “the collection and analysis of data from a wide variety of projects” by Kitchenham et al. [39]. The data is collected either as a special event, or as part of routine operations, such as project management information in project plan documents. The study is designed either before the data is collected, such as for an opinion survey by phone, or after the data is collected, such as considering what could be learned from historical records of project management. These two factors of special event
and pre- or post-design can affect the quality and consistency of the data collected, and in turn, the precision of the conclusions reached during the analysis of the data. There is a large body of literature on the design and execution of surveys, which should be consulted before undertaking a survey.

The data collected by surveys is generally from real-life situations rather than laboratory settings. Often, surveys cover multiple sites, such as multiple divisions of a company, or multiple companies, as defined by Kemerer [35]. This increases the applicability of the results of the survey to different organizations which is called the generalization or external validity of the results. Generally, surveys collect a large quantity of data points on the same topic to improve the reliability of the statistical analysis of the data. However, there can be limitations in the accuracy of this data, which reduce the statistical reliability.

The problems which can affect the accuracy of the data are related to the proper understanding by the subjects of what information is being requested, their motivation to provide accurate information, and proper interpretation by the researcher of the information provided. For example, in a phone survey, the person making the phone calls can answer questions and provide clarification on what information is being requested. At the same
time, a phone survey could be viewed as an annoying interruption by the subject contacted, reducing the subject's motivation to take the time to provide complete or accurate information. As another example, in a survey of historical project management information, the researcher has to interpret the information contained in project plans. If the authors of these documents are no longer available, it is possible that the interpretation may be inaccurate. It is also possible that over time the information provided in project plans evolved, which may not be clear from reading the plans.

A final issue with surveys is the selection of subjects. To provide good statistical accuracy it is important that the subjects be selected randomly from the overall population. This can be difficult to achieve. For example, in a phone survey, not all the people contacted will participate. Or in the example of a historical survey of project plans, not all projects may have completed or filed plans. In these cases, the survey needs to account for these dropouts in its design or analysis.

As an example, Card et al. [23] report on a survey performed on data collected from a set of projects at NASA, and recorded in the Software Engineering Laboratory's database. They selected 22 projects from the database, all from the same spacecraft flight dynamics software domain. The database contained information on project characteristics, such as size,
software changes and errors, programmer effort from labor tracking information, and computer utilization statistics. Card et al. attempted to identify the possible impacts of eight different software development methodologies, such as structured coding and code reading, on the performance of these projects. In this survey, Card et al. provide a good example of confirming that the data was defined and collected consistently, and selecting a consistent set of projects to study. They ran into problems when they overanalyzed the data and drew causal conclusions that may not have existed.

The main advantages for surveys are:

1. Surveys are reasonable to do with professional programmers in industrial environments, giving data from real-life situations.

2. A large data set can be obtained, which allows statistical analysis, at a relatively low cost compared to other empirical methods.

3. A survey with a large data set from a broad set of subjects and sites can provide good external validity.
The main weaknesses of surveys are:

1. The low level of control of the subjects and the data collection process by the researcher can lead to high levels of variability and subjectivity in the data.

2. Sometimes the large quantity of data leads to over-analysis and the researcher loses sight of the limitations inherent in the data.

3. The researcher is often not directly involved in the environments surveyed, and therefore may not observe other events affecting the surveyed activities.

2.3 Case Studies

Case studies are conducted in a normal software development environment. For example, a project team might conduct a case study by using a new specification language on their next development project. If the case study consists of a single use of this specification language by a single group, it is an experience report. If the use of the new specification language is repeated across multiple project teams or across multiple organizations (defined as a field study by Hale et al. [32]) then the information gathered can be generalized, and this is a form of information gathering study. For a case study to be part of a comparative study, the results from using the new
specification language need to be compared to an earlier project done using the previous method. Or a similar project would be done by a similar team to provide the comparison.

Case studies are generally done using full-scale projects which makes them costly and time consuming. Therefore, case studies are often done only once or a few times. This makes case studies the most likely empirical method to be an anecdotal report, in spite of the time and effort that may have gone into the study, and the amount of data that may have been collected. Also, case studies are generally not repeated enough to do a statistical comparison.

Because the researcher is working in the real world during a case study, they can discover unexpected insights from unplanned occurrences, and adjust their research to match this clearer understanding, as stated by Potts [45] in his detailed evaluation of the use of case studies in research on the requirements definition process.

Lubars et al. [40] provides an example of a case study. In this study, the three researchers worked with a software development team at a large aerospace firm, developing a portion of the preflight engagement software for cruise missiles. They were studying object oriented design techniques. The
source information for the design was a 529 page Navy document describing the operating procedures. They also attended a class given for missile operation planners. They compared a top-down design technique to a bottom-up technique for deriving objects. They found that on systems this complex, both methods were required to successfully specify the needed objects.

The main advantages of case studies are:

1. Case studies are reasonable to do with professional programmers in industrial environments, giving data from real-life situations.

2. Because the researcher is actively involved observing the subjects, the researcher can see things occur that are beyond the specific planned data collection. This can help with interpretation of the data collected, and also provide insights into aspects outside the original research focus.

3. Case studies can be used in situations where the researcher does not have the control to enforce the random assignment of subjects to tasks and other stimuli.

4. Because there is less control of the subjects and objects, case studies generally have a lower cost than a controlled experiment in same environment.
The main weaknesses of case studies are:

1. The low level of control over the subjects and the environment can lead to high levels of variability in the resulting data.

2. A case study from a single environment has limited external validity in application to other, different environments. Field studies, of multiple different environments, can help reduce this weakness.

3. Generally, case studies do not have enough separate samples to allow statistical conclusions.

2.4 Quantitative Modeling with Simulation

There are a variety of quantitative models used for prediction and simulation of software development. A quantitative model is a numerically based set of relationships which describe one or more aspects of something from the real-world. An example would be the COCOMO software project lifecycle model developed by Boehm [20]. This model is intended to provide an estimate of the effort and schedule required to develop a software project, based on a set of characteristics for the project and its environment. Using a model for simulation involves changing the value of one or more of the characteristics the model is based on and then re-calculating the result.
Tichy et al. [49], in their survey of the use of empirical methods reported in the computer science literature, contend that not all simulations are true empirical methods. In their view a simulation has to either (1) be a simulation where the model is used to generate data which is the input to another empirical method, such as a case study (for instance, generating multiple 'what-if' scenarios for evaluation against one or more scenarios from the real-world), or (2) be a simulation where the model uses real-world data traces as input and is performed in a realistic set-up, such as simulating job scheduling in an operating system using a captured job-stream. They were surveying the reports of use of empirical methods in computer science literature, and wanted to make a clear distinction between the actual use of a quantitative model for empirical research, versus an example which simply shows how the model works.

As an example of quantitative modeling used for simulation, Kellner and Hansen [34] describe “software process modeling” as a quantitative model used for simulation in “a technique that encompasses a representation approach, comprehensive analysis capabilities, and the capability to make predictions regarding the effects of changes to a process.” Once a model is created and calibrated to an environment, variables can be manipulated as with a controlled experiment, and the effects measured. This can give operational guidance to allow the assessment of potential process changes
before they are made (Raffo [46]). Once a series of trials has been completed to evaluate alternative courses of action, the preferred alternative would be selected and applied to the actual environment, with the resulting effect studied by one of the empirical methods discussed previously, to validate the model's prediction.

The main advantages of quantitative modeling with simulation are:

1. The model allows the researcher to conduct repeated "what-if" scenarios to find a preferred solution at a very low cost per trial, compared to conducting actual case studies or controlled experiments equivalent to each "what-if."

2. The model provides insights into the operation of the modeled process and the interaction of its variables.

3. A model based on a real-world environment, and properly calibrated, can provide good external validity to similar environments, or be re-calibrated for new environments.

4. For stochastic simulation, the outputs are in the form of statistical distributions which enables the generation of confidence intervals, showing the inherent uncertainty associated with the modeled process.
The main weaknesses of quantitative modeling with simulation are:

1. Often the model must be calibrated to each specific environment, reducing overall external validity.

2. After running a series of 'what-if' scenarios and the selection of a preferred 'what-if' scenario, the predicted results still need to be validated by a separate empirical study in a real-world environment.

2.5 **Controlled Experiments**

Controlled (or formal) experiments are studies where the researcher has a high degree of control over the environment. Changes, also called stimuli, are systematically applied in a properly designed, random fashion in repeated trials for both a control and one or more experimental situations. Subjects are randomly assigned to one or more of these trials. A consistent set of measurements is taken during each trial. The randomization and repetition allow for a statistical analysis of the results to determine if a cause and effect relationship exists between the changes made and the results measured. See Adelman [17], Basili et al. [18], Hale et al. [32], or Pfleeger [44] for a more detailed description of controlled experiments.

The distinguishing characteristics of controlled experiments are the degree to which the researcher can control extraneous factors and the
application of stimuli to subjects in a random fashion. This degree of control provides a benefit through tight statistical control, but is expensive to achieve in real-world situations. As a result, controlled experiments are often conducted in specialized or laboratory environments, using small, easily controlled objects, and limiting the numbers of subjects. These limitations on the experiment reduce the actual generality of the results measured. Since much of the software development world is relatively uncontrolled, the high degree of control imposed by a controlled experiment can perturb any measured result.

An example controlled experiment conducted as part of this thesis is presented in Chapter 4. Additional examples of controlled experiments in software development research are described in the annotated section of the Bibliography.

The main advantages of controlled experiments are:

1. High degree of control by the researcher over the application of stimuli and collection of data can lead to reduced variability.

2. Because of the high degree of control, statistically significant results should be more readily obtained than from other empirical methods.
3. The random application of stimuli and selection of subjects in controlled experiments should allow them to have greater generality to other environments than other equivalently conducted (same number of subjects, size of objects, environment, etc.) empirical studies.

The main weaknesses of controlled experiments are:

1. The high degree of control over subjects and data collection in a controlled experiment can create an artificial environment.

2. Controlled experiments have a higher cost than other equivalently conducted (same number of subjects, size of objects, environment, etc.) empirical studies.

3. The frequent use of small objects and novice subjects in controlled experiments (usually to control cost of the experiment) limits actual external validity of experimental result.
3. ISSUES WITH CONTROLLED EXPERIMENTS IN SOFTWARE

Controlled experiments can provide the most direct understanding of causal relationships between changes in an environment and their possible effects. To gain the benefits of controlled experimentation, it is important to understand two areas:

- **How to properly carry out a controlled experiment:** Providing a detailed tutorial on designing and carrying out controlled experiments is beyond the scope of this thesis. However, Chapter 4 provides an example controlled experiment and Appendix A provides references to additional sources of information on controlled experiments.

- **The limitations and issues of using most empirical methods, including controlled experiments:** All of the empirical methods discussed in Chapter 2 have a related set of limitations due to the nature of software development, and issues in their proper use when studying software development. To a greater or lessor degree, the effectiveness of each empirical method is impacted by these limitations and issues.
In "No Silver Bullets: Essence and Accidents of Software Engineering" [21], Brooks asserts that there are two categories of limitations which have held back the advancement of software engineering. The first category is what he calls 'essential' limitations. These are limitations inherent to the nature of software, and there is no way to solve them. Instead, they must be managed for and will continue to make software development a complex and difficult task. The second category is what Brooks calls 'accidental' limitations. These are not inherent in the nature of software, but instead are due to the methods and techniques we have developed. These accidental limitations can be overcome as we develop better methods and tools for software development.

Similarly, in performing empirical studies of software development, there are 'essential' limitations which appear to be inherent in the nature of software, and 'accidental' issues due to improper understanding or application of the empirical methods. This chapter presents these limitations and issues, specifically for controlled experiments, and proposes approaches for dealing with them.
3.1 Limitations due to the Nature of Software Development

This section discusses the limitations encountered when using empirical methods, especially controlled experiments, to study software development. These limitations are due to the complex nature of the software being developed as well as the associated processes of software development, and the small base of theories and models we have today on which empirical studies can be built. Most of the questions we are trying to answer about software development require extended studies because of this complexity. This complexity leads to a multitude of factors which can affect the outcome of a study, and can be difficult separate. For many of these factors it can be hard to know what to measure, or how to measure it. Finally, because software development is a human intellectual activity, the variation in performance between individuals can have a greater impact on a study than the effect of a change in a method or tool.

These limitations can have a profound impact on a software development study. At a minimum, they need to be carefully accounted for in the design and execution of a study. They can mean that a study needs to start with work to build a sufficient foundation of understanding before the actual question to be answered can be addressed. And they may even make a study impossible to carry out.
3.1.1 Understanding of Software Development

Software development is a large, multidimensional, highly complex and interrelated subject. Much of what occurs during software development is directly related to human thinking processes and the social interactions of project teams, while our current empirical understanding is limited to a few specific points and technical areas. We do not have a strong empirical model (or theory) for the basic building blocks of software development. Studies are limited to single experiments exploring cause and effect relationships in specific situations or at best building small points of theory on how software development occurs.

Without a theory or empirical model to provide a base for understanding the fundamentals of software development, it is very difficult to effectively study many of the problems we take on in software development research. It is easy to ask "Will using specification method X improve my software quality?" It is very hard to answer this question, because we do not have a precise definition or measure for quality. Further, how software is specified is only one element among many in the software development process that affects the quality of the finished product. Until a set of fundamentals is determined, many such questions appear to be unanswerable. And even with a substantial investment of time and effort to explore them, these questions may be intractable.
3.1.2 Studying Complex Problems

With the complexity of software development, it can take several small experiments just to understand what should really be measured and controlled, or to determine if additional study is feasible. Because software development is composed of so many interrelated factors, experimenters need to explore the possible interactions, what Basili calls the "critical mass" of factors which may be required to have an effect.

Research needs to start with a broad vision of what needs to be understood, and what environments need to be considered. This vision includes planning a sequence of experiments to make a complete study of a research area. To accomplish this goal, the support of large and well funded academic or industrial research organizations is required. Further, these organizations need to have a long-term, big-picture view of their role in advancing the understanding of software development.

3.1.3 Confounding of Factors

"Confounding" is the inability to determine which factors actually influenced an outcome, sometimes showing up during analysis of data from a study when two or more factors cannot be separated. A more insidious form of confounding occurs between a factor captured in the study and another factor that was not recognized. In this case, an observed effect may be
attributed to the studied factor, when that effect is actually due to the unrecognized confounding factor.

Two separate effects cause confounding to occur in software development:

- Every environment contains many, many factors which are very difficult to quantify or control between groups in a study. When a difference is found in the results of a research project, there is the possibility that the difference is due more to some uncontrolled factor, rather than the change under study. For example, Pfleeger [44] mentions the confounding that can occur when evaluating a tool which implements a particular methodology. If the methodology is not accounted for in the experiment's design, the researcher cannot know if it was the tool or the methodology which made a difference.

- The evolutionary nature of software development. Software development takes place in a lifecycle of multiple phases. Mohamed et al. [42] point out that the effect of a method or tool may be spread over several of these phases, therefore delaying the timescale of the overall experiment. For example, an improved requirements analysis method may reduce the number of errors which show up during user testing. Simply measuring the effect of this new requirements analysis
method during the requirements gathering phase would not show its full impact.

The best protection against the confounding of results in a study is careful consideration during the design. In the example of possible confounding between a tool and a method, if a new tool is to be the subject of a controlled experiment, and the methodology behind the tool is not in current use by the subjects, then the researcher should design the sequence of experimental trials to first find the effect from the method, and then find the effect from the tool. Otherwise, the researcher cannot conclude whether an effect was from the tool, the method, or shared between the two.

Any change in a software development process impacts more than just the phase of the software development lifecycle in which it is introduced. Therefore, all phases of the lifecycle in use in the experimental environment should be measured and understood. For example, a new requirements analysis tool will impact the requirements gathering phase. Changes it creates in the resulting requirements may in turn create changes, good or bad, for the rest of the development process, production, and use of the developed software. The impact of this new requirements tool must be measured at all phases of software development and in the product's use by
customers to see the full benefit of the new tool and to understand any hidden costs of the new tool.

To reduce the overall exposure to confounding of results in any one experiment, the experiment should be replicated in multiple environments. In this way, unrecognized confounding of factors specific to one environment will be discovered by the difference in results in the replication in another environment.

3.1.4 Surrogate Measures

When performing empirical studies in software development, one of the biggest problems is that many of the attributes we wish to study do not have generally agreed to methods of measurement. To overcome the lack of a measure for an attribute, some factor which can be measured is used instead. This alternate measure is presumed to be related to the actual attribute with which the study is concerned. These alternate measures are called surrogate measures. The use of surrogate measures is most important in controlled experiments, because of the statistical analysis that is a key element of a controlled experiment.

For example, there is no clear definition of what 'quality' means. Frequently some measure of 'lack of defects' or 'more defects were found
during the development lifecycle’ (therefore fewer defects are presumed to remain) is used as a surrogate. Studies where ‘lack of defects’ is used as a surrogate for quality rarely take the additional step of collecting reports of failures during customer use to corroborate the presumption that the quality was better. As pointed out by Fenton et al. [30] from a study by IBM, many faults take years of operation to show up as actual failures for users.

As another example, Haas and Hassell [31] run through a whole sequence of surrogate measures used for program understanding, and then point out difficulties in getting valid data for each measure. For example, in the case of ‘program reconstruction’ the authors report they have found that “many subjects have an intense dislike for the reconstruction measure.” Their concern is that “such a negative reaction can easily influence an individual’s performance on reconstruction tests and thus can bias the results of experiments using them.”

Problems with the use of surrogate measures occur most often when the surrogate measure is used in place of a desired attribute to determine if a change in the software development environment impacts that attribute. If the surrogate measure used has no empirically demonstrated relation to the intuitive meaning of the attribute being studied, any result found by the study
can only be directly related to the surrogate measure, and not the desired attribute.

Studies that involve an attribute of software development having no clearly demonstrated measures available should be avoided. Alternatively, a preliminary study can be done to validate a relationship between the surrogate measure used and the attribute being studied. For example, if a researcher uses "the number of faults found during development" as a measure of the goodness of a testing method, the research should follow the product into production use and measure the impact this testing method has had on the failures that customers report. For a valid comparison, the researcher needs to also follow the product into production use as part of the baseline or control sections of the study.

In the long term, software development research should identify measures with demonstrated relationships and the conditions under which these relationships hold. These measures would become part of the base of software development knowledge, reducing the total effort of studying other issues in software development. The works of Brooks [22] and Haas and Hassell [31] to identify proper ways to measure program understanding are examples of extending the base of knowledge.
3.1.5 Subject Variability

There is tremendous variation in the performance of individuals on tasks studied in software development research. Researchers who study this variation report ranges often as much as 20 to 1 between the best and worst performers. For example, Curtis [24] reported a range of 23 to 1 on a study of debugging data from GE. Frequently, the variation among the subjects of a controlled experiment can be greater than the variation between the control and experimental changes. This can make analysis of the experimental data difficult and can reduce the statistical confidence in the conclusions. Worst of all, research has not found any easily measured variables which can help to control for this variation. Brooks [22] reports no less variability when using students with apparently similar backgrounds (measured by specific courses or grade levels) instead of using professional programmers. DeMarco and Lister [28] report that in their 'coding war games' they have found no relation between actual performance and the traditional measures of professional programmer performance such as 'years of experience' or 'salary.'

Curtis [25] reports that in spite of decades of work to identify measures for subject variability, only by careful analysis of specific job skill requirements has any progress been made, and that is restricted to the specific environments studied. Related to this, Potts [45] stresses that understanding of the specific application domain can be a significant factor in
performance, and needs to be considered similarly to novice versus professional levels of experience.

Subject variability is a central problem in software development research. Part of the problem is that most research has tended to focus on methods and tools, and not the cognitive processes of the people using them. A few researchers have looked at the behavioral science aspects of software development. Shneiderman [48] does an excellent job of presenting a cognitive model of how people process information between short-term memory, working memory, and long-term memory. Keeping in mind that long-term memory is where semantic and syntactic knowledge is stored, this model helps explain performance differences between novice and experienced programmers. von Mayrhauser and Vans [50] have reviewed several additional cognitive models, and propose their own ‘meta-model’ built up from all of them. They also point out that there is very little empirical research to support any of these models or to demonstrate how they can be applied.

Until clear controls for subject variability are identified and understood, there are some things that can help address subject variability in software development research. First, Curtis [25] contends that all researchers consciously or unconsciously select one of five “cognitive paradigms” he
identifies. These are viewpoints that the research takes in considering the scope and depth of the impact of the human element on the research. In each cognitive paradigm, different classes of variables are controlled or measured to understand the human aspects of the study. Figure 1, below, presents these paradigms. By understanding which paradigm is being used, it is possible to ensure that the measures being used are appropriate to the assumptions being made about the human element's impact.

Second, in addition to considering the subject's general experience level in software development, the study should consider the specific experience with the exact domain of the environment and objects. Both the application domain and the environment should be identified; For example, spacecraft control software and the type of workstation used. The subjects should be evaluated for these characteristics and then assigned to the experiment and control situations. The experiment and control groups should have a balanced level of domain experience assigned to each situation.

Finally, experiments should be designed as within subject studies when practical. This means that an individual's performance is measured on both the control and the experiment situations. In this way, the performance of a subject on the control can be compared to the same subject's performance on the experiment situation. This reduces the impact of
variation from individual to individual, but is still subject to variations in individual performance from situation to situation.

**Figure 1: Cognitive Paradigms from Curtis [25]**

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Definition</th>
<th>Dynamic Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual Variation</td>
<td>Predict individual performance empirically</td>
<td>Difference between individuals</td>
</tr>
<tr>
<td>Group Behavior</td>
<td>Social behavior of groups</td>
<td>Group structure and interaction</td>
</tr>
<tr>
<td>Organizational Behavior</td>
<td>Behavior of groups of teams in the larger structure</td>
<td>Group and organization structure and interaction</td>
</tr>
<tr>
<td>Human Factors</td>
<td>Relation between stimuli and response of individuals</td>
<td>Stimulus objects</td>
</tr>
<tr>
<td>Cognitive Science</td>
<td>How knowledge is acquired, represented in memory and used in solving problems</td>
<td>Differences in how individuals learn and apply information to varying stimuli</td>
</tr>
</tbody>
</table>

### 3.2 Issues in using Controlled Experiments

In addition to the limitations due to the complexity of software development and our understanding of it, described in the previous sections, there are common issues in properly using empirical methods for software development studies. These proper use issues are generally due to not fully understanding how an empirical method should be applied, or to simply
failing to properly plan or carry out the study. A difficult usage issue can be the need to treat all cases and subjects in a study as similarly as possible so that any effect measured can be tied to the question being studied. The learning curve associated with a new method or tool being studied can affect the results of a study, and may need to be measured through repeated cycles of use. The size of a software project can have a substantial impact on the outcome of a study and needs to be appropriate to the question to be answered. The subjects selected to participate in a study need to be match the type of subjects the result is expected to apply to. Empirical methods, and especially controlled experiments, involve the analysis of quantitative data. It is important that the proper statistical techniques are used, and that the interpretation of the results matches these statistical techniques.

Although these issues will not prevent a software development question from being answered, they must be properly handled for a study to achieve accurate results.

3.2.1 Maintaining Equivalence Across Treatments

The term treatment refers to each different method, tool or other condition which will be studied an experiment. For example, in evaluating a new debugging tool, the treatments might be the old debugging tool and the new debugging tool. One of the hardest tasks of the researcher using
controlled experimentation in software development is to establish truly
equivalent situations for both the experiment and the control. In studying any
type of software development methodology, the control group will use some
alternate method to solve the problem assigned. There is no such thing as
'no' methodology, so the researcher should provide some alternative. In this
way, the researcher will know how the experimental method compares to the
alternative, rather than how it compares to multiple alternatives the subjects
chose for themselves.

Closely related to the issue for control treatments is the training and
general encouragement (interest) a researcher will often give to the
experimental group. If similar attention is not paid to the control group, there
may be motivational differences between the two groups, which Curtis [25]
has identified as a key factor in individual performance.

Basili and Reiter [2] is an example of an experiment which has both of
these problems. They desire to show that a disciplined team using a defined
set of software engineering methods will perform better than teams or
individuals using ad-hoc (undefined by the researchers) methods. Because
they give the control (ad-hoc) groups no instruction, there is really nothing to
compare their defined software engineering methods to. And, they provide
training and attention to the experimental group, while doing nothing
equivalent with the ad-hoc groups. It is very possible that this made the ad-hoc groups less motivated, and therefore their performance was impacted.

To properly design a controlled experiment, there are several steps a researcher should take to maintain the equivalence of all treatments. First, the researcher needs to explicitly identify the baseline or control against which the experimental treatment will be compared. Next, the researcher should use whatever methods and measures possible to evaluate the materials, training, and instructions used with each treatment, to assure that they are as equivalent as possible.

Motivation of the subjects is a key reason why equivalence across treatments is important. Frequently, a software development researcher is looking to demonstrate that their new method or tool is 'better.' The researcher's enthusiasm and active support can be a confounding factor in determining the actual benefit of the new method if it is focused exclusively or primarily on the subjects using the new method. Therefore, the study should be designed so that both the experimental and control treatments receive the same attention, such as training and support.
3.2.2 Cycles of use

Complex changes in technology and new methodologies are rarely learned and fully understood with a single exposure through training or use. Controlled experiments need to account for this ‘learning curve’ for all but simple changes. Fenton et al. [30] present the example of a NASA program which made the transition from FORTRAN to ADA. During the first project done in ADA, productivity and quality were actually below that previously achieved in FORTRAN. It was not until the third project done in ADA that the benefits of the technology change were clearly demonstrated.

If a controlled experimental study is being conducted for a new method or technique, it is critical that the method be used for several cycles. Repeating the study allows researchers to understand the learning curve and gives the subjects time to fully learn the nuances of the new method. It may also be important to provide training or usage support multiple times during this learning period, and not expect that a simple, one-shot class will be sufficient. A baseline of the subject group's performance with their previous method should be established before starting the new method's installation. Then, with each usage cycle, a new set of measurements should be taken to determine the length and incline of the learning curve for this method.
3.2.3 Size of the Problem

Fenton et al. [30] state succinctly: "just as software-development-in-the-small differs from software-development-in-the-large, research-in-the-small may differ from research-in-the-large." The object selected for the study needs to match the goals of the study. If the intent of a controlled experimental study is to demonstrate the value of a new methodology in large scale software development, as is generally carried out by industrial organizations, then the study cannot be done with 100 or 500 line program examples. It must be done with large scale programs, often 50,000 lines to 500,000 lines, or more.

For example, in Shneiderman's et al. [12] studies of the effectiveness of flowcharts, only small programs were used. Shneiderman, in his conclusions, points out that similar studies should be conducted with larger programs (over 1000 lines of code) to see if an effect would be demonstrated. Curtis [25] echoes this by saying "Unfortunately, most of the research has been performed on small batch-like systems, rather than on the large, interrupt driven, embedded systems in which control flow is much more complex and often involves a temporal component."

It can be valuable to start with a small object to calibrate the experimental materials and measures for cycles with a larger object.
However, the researcher should be aware that a whole different set of factors can come into play as the object of the study becomes 'software-in-the-large.' As Curtis [25] points out "novices comprehend a program based on its surface structure... Experts, however, analyze a program by the solution or algorithmic structure of the program." With this in mind, the object used in a study needs to be sufficiently large that it cannot simply be solved by 'brute-force.' Both the cognitive processes individuals use and the impact of social and organizational issues change as the size of the object moves from the trivial to the significant.

3.2.4 Subject Selection

The primary problem that occurs with subject selection is failing to get the proper quantity of subjects with the appropriate skill and knowledge levels. Sometimes this is due to the researcher simply not realizing that the background of the subjects and the number selected are critical factors in achieving meaningful results in an empirical study. It can be hard to enlist professional staff as the subjects for studies which will take them away from their normal work. And it can hard to convince a project manager that they should add risk to a new project by trying a new method to see if it will be an improvement. However, without subjects representative of the desired population, little can be concluded regarding that population. As pointed out by Moher and Schneider [43], subjects are frequently selected based on
availability to the researcher -- students. They also point out that studies have shown that novices and experts use different methods for solving problems, as supported by the cognitive processing model of Shneiderman [48] and the 'programming plans' and 'rules of programming discourse' studied by Soloway and Ehrlich [15].

Even when subjects from the apparently same population of professional programmers are used, problems can occur. DeMarco and Lister [28] have found an overall variation of 10 to 1 among the professional programmers participating in their 'coding war games.' However, the variation between individuals from the same companies is only about 20%. This indicates that there may be limitations on generalizing the results of studies from a single organization.

In some cases, students provide a reasonable population to draw subjects from. If the problem being studied is independent of the experience and expertise of the subjects, then either novice (student) or experienced subjects may be used. As an example, the study reported by Sinha and Vessey [13], looks at a cognitive fit theory which they state does not depend on experience or expertise, and so they use student subjects. However, if they had wished to validate the cognitive fit theory for professional
programmers, then they would have needed subjects who were at the professional level.

Dealing with variations from one industrial environment to another can be more difficult. The normal solution would be to replicate an experiment in multiple industrial environments. If a working manager wished to bring a method into their environment which had been reported to work well elsewhere, they could replicate the experiment (or use another empirical study method such as a case study) to validate that a similar result would be achieved in their organization.

3.2.5 Proper Statistical Analysis

The use of an appropriate statistical analysis method can be an issue in analyzing the data from an experiment. The analysis method should match the purpose of the experiment. For example, if an experiment is conducted to compare the defect detecting efficiency of two testing methods, performing a correlation analysis would be inappropriate. The proper statistical test would be a comparison of the means for the two methods. The analysis method should also match the design of the experiment in terms of the number of different situations considered, and the number of groupings (called levels) of the variables measured. For example, if there are two groups of data to be compared in a test of means, Student's t-test could be used. If there are more
groups of data, the F statistic would be used as part of a more general analysis of variance between the groups of data.

Pfleeger [44], in her December 1995 article, provides a detailed discussion of how to choose the proper analysis method. If the experiment's purpose is to confirm a theory, evaluate the accuracy of a model, or compare two or more methods or tools, then a statistical test, such as analysis of variance, should be performed to see if there is a statistically significant difference between the alternatives. If the experiment's purpose is to test for a relationship between two or more factors, or to validate a proposed measure, then a correlation analysis should be used. If the purpose is to develop an equation that describes the relationship, then a regression method would be used.

3.2.6 Proper Interpretation of Results

The most common issue with interpreting the results of a controlled experiment is over generalizing the results by applying them to other environments. Researchers should resist the desire to 'increase' the value of their work by making unsupported statements about its applicability to situations beyond what they have actually controlled for. And those reading research results should be skeptical of broad statements of applicability.
Another issue with interpreting results involves a proper understanding of the meaning of the statistical analysis. There is sometimes confusion between the statistical statement 'could not disprove the null hypothesis' and the natural language statement 'there is no relationship.' When the data analysis does not determine that an experimental situation had an effect, it may simply be that the study was designed poorly. For example, if an experiment is expecting a small effect, using too few subjects could give a 50/50 chance that any particular replication of the experiment would show a statistically significant result. In this case, a single experiment showing no result simply means that the experiment needs to be replicated multiple times until enough subjects have been used.
4. AN EXAMPLE EXPERIMENT

During the Fall of 1995 I conducted a controlled experiment to gain first-hand knowledge about this empirical method. The experiment was carried out at Portland State University using students from 2 sections of CS 161. CS 161 is the first introductory course for new Computer Science majors. The experiment was designed to compare the understandability of two program coding styles for these novice students. Program understanding is defined by Haas and Hassel [31] as an "individual's ability to draw inferences about a program."

The development and execution of the experiment, along with the analysis of the results, is covered in detail in Section 4.1. In Section 4.2 I review my experiment against the limitations and issues presented in Chapter 3.

4.1 The Experiment

The details of program coding are traditionally taught by starting with a very simple program, such as the "hello, world" example in Kernighan and Ritchie [37]. Simple operations, such as assignment and arithmetic, and
control structures, such as WHILE loops, are then added to create basic programs. At some point, structuring elements such as functions or procedures are added to introduce the concepts of structured programming. When teaching coding to novice students, educators question the timing for introducing the concepts of structured programming and structured program elements such as functions. Is it better to start students with some basic program operations and delay the introduction of program structuring through functions, as with the traditional teaching methods? Or is it better to immediately start out with the slightly more complex subject of functions and therefore introduce structured programming as one of the first concepts novice students learn?

The question of when to introduce structured programming relates to the question “Is it really harder for novice students to understand the use of functions in program coding?” A search of the literature did not find any empirical evaluation of the question of novice students understanding functions or procedures. The controlled experiment described here attempts an empirical evaluation of this subject.

This controlled experiment is divided into several tasks: First finding a program that can easily express two coding styles; Second, corralling a representative set of subjects; Third, defining an objective, yet accurate
method of measuring subject understanding; Fourth, creating the materials to be used; Fifth, executing the experiment; Sixth, validating the data collected; Seventh, determining the proper statistical techniques; Eighth, analyzing the experiment results; Lastly drawing conclusions.

4.1.1 Objects

In experiments, the term ‘objects’ refers to the items which have the experimental change applied to them. For example, a program listing written in two code styles. To test whether novice students have difficulty understanding functions, samples of the two coding styles were needed. One sample was written in the basic ‘all in one main program’ style (called ‘non-function style’ in the remainder of this thesis) where any similar blocks of code are repeated as needed within the program. The second coding style (called ‘function style’) used functions to structure the program and avoided the repetition of blocks of code within the program. To have the most consistency between the two code style samples, different versions of the same program listing were used.

To find a program listing representative of what students learn in their first programming class, a search was done of a selection of introductory programming/software engineering texts. The program selected was from Jones [33]. One of the reasons this program was selected is that Jones
presented it in both coding styles: First, the non-function style when basic programming was introduced; And secondly, the function style when procedures were introduced in the text. Another reason the program was chosen was that it was one page in length, a typical length for programs the students in CS 161 had seen up to that point. Keeping the program length to one page avoided possible complications from the students having to flip back and forth between pages of a program during the experiment. The programs were translated from Pascal into C++, since C++ was the language used in the classes from which the subjects were drawn. The two versions of the program are presented in Appendix B as Figures 11 and 12, non-function and function style.

4.1.2 Subjects

The subjects for this experiment were drawn from the two sections of CS 161 taught during the Fall quarter, 1995, at Portland State University. CS 161 is the introductory course for CS majors who have no prior computer experience. Use of these students was intended to provide a relatively uniform set of novice programmer subjects.

In order to conduct an experiment under the auspices of Portland State University, in which people are the subjects, the rules of the Human Subject Research Review Committee (HSRRC) must be reviewed and
followed. Because this experiment involved no direct impact on the subjects, and the data collected could not be linked back to specific individuals, a waiver from full committee review was applied for and approved. The terms of this waiver required that:

- Only volunteer subjects could be used
- Volunteers must be at least 18 years of age
- Participation in the experiment had no affect on the volunteer's grade in the course
- The course professor have no knowledge of who participated
- The experimental questionnaire not be called a test (the term 'survey' was used in all materials and interactions with the subjects)
- That a signed consent form (the consent form having been pre-approved by the HSRRC as part of the application process) be obtained from each subject that participated in the experiment.

The approved informed consent form is included in Appendix B as Figure 13.
4.1.3 Measures

To measure the ease with which the students could understand the code styles, a questionnaire was developed. Brooks [22] reviews different techniques for measuring programming tasks, and concludes that the use of question answering is an effective method to measure program understanding. He also lists two issues in creating and scoring a valid questionnaire. First, when open-ended or short answer questions are used, creating the questions is straightforward, but scoring can be difficult and subjective. Second, when multiple choice questions are used, the alternatives to the correct answers need to be reasonable answers, ones that a subject would likely find if his/her understanding of the program is somewhat off. With these issues in mind, a 'fill-in-the-blank' type of questionnaire was created, using questions with simple numeric answers to keep scoring objective.

Haas and Hassell [31] provide a more detailed discussion of the potential types of questions to measure program understanding. 'Forward reasoning questions' are those where the subject starts with input to a program and is asked what output would be produced. 'Backward reasoning questions' are those where the subject is given output from the program and asked what input would create it. For this questionnaire, three forward reasoning questions (1, 2, and 4) and two backwards reasoning questions (3
and 5) were asked. 'Special questions' are those that require the subject to have a deeper understanding of the overall operation of the program. Question (6) was asked as a special question. See Appendix B, Figure 14 for the questions used in this experiment.

To provide a secondary measure of the ease with which the students' understood the coding styles, the subjects recorded the start and finish time for each question on the questionnaire. Both Brooks [22] and Haas and Hassell [31] caution about the use of time to complete a task as a measure of performance, primarily because it can be confounded by other factors when the subject spends time during the experiment not performing the actual experimental task. However, researchers such as Scanlan [10] have found that the time needed to complete a task can be a critical factor in detecting an experimental effect. Therefore, the experiment was designed to complete processes such as instructing the subjects before the actual experimental task timing began, and to provide time (by collecting start and stop times on each question) between parts of the task where the subject might normally pause.

To measure differences between the students that might affect their scores on the questionnaire, several items of background information were collected. These included: 'number of previous computer or programming
classes,' 'number of programs written,' 'college GPA (or high school GPA if
less than two terms of college),' and 'college experience in credit hours and
years.' As mentioned in Chapter 3 in the section Subject Variability, these
measures do not provide an exact characterization of performance
differences between individual subjects, but can provide a basic set of
measures to allow stratification of the subjects into one or two classes, and
can eliminate subjects as 'too experienced.'

The experiment was designed as a simple comparison of the two code
styles. Subjects were randomly assigned to one code style or the other by
interleaving the experimental survey packets (described below in the section
Materials) so that the two code styles alternated in a stack of questionnaires.
The packets were then passed out without informing the subjects that there
were two versions of the program listing. The students were generally
observed to sit in groups with friends, so this interleaving of the
questionnaires provided an approximately equal assignment of students from
each group of friends to each code style. When students completed the
questionnaires, the questionnaires were marked as to which section the
subject was in, to allow for possible grouping by class section, in case this
had an impact on their performance on the questionnaire.
The specific statistical questions asked in this experiment, called the hypotheses, were:

- **Null Hypothesis**: There will be no statistically significant difference in average scores of subjects on the program understanding questionnaire between the functionally block structured style and the in-line structured style treatments.

- **Alternative Hypothesis**: There will be a statistically significant difference in average scores of subjects on the program understanding questionnaire between the functionally block structured style and the in-line structured style treatments.

### 4.1.4 Materials

An initial set of materials for the experiment were created and tested in a pilot run of the experiment. As mentioned earlier, in the section Object, two program listings were created for the same program, using one or the other of the code styles. To ensure that the two versions of the program were equivalent, they were checked to ensure that each statement with the same purpose was identical from program to program, with the exception of the differences due to use of functions in one listing. Standard word count statistics were run on each program to assure that the listings were equivalent, and are shown in Figure 2. The only significant difference is in
the word and character counts, because the non-function version of the program repeats some statements that are functionally blocked in the other listing. As a final check of equivalence, a professional technical editor was consulted to provide a subjective evaluation of the two program listings, and stated that they were "visually equivalent for a reader."

**Figure 2: Word Count Measures for Program Listings**

<table>
<thead>
<tr>
<th>Measure</th>
<th>Non-function style</th>
<th>Function style</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pages</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Words</td>
<td>299</td>
<td>244</td>
</tr>
<tr>
<td>Characters</td>
<td>1146</td>
<td>984</td>
</tr>
<tr>
<td>Paragraphs</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>Total Lines</td>
<td>54</td>
<td>55</td>
</tr>
<tr>
<td>Blank Lines</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Based on the function of the program, understanding questions were created as stated in the previous section. In its initial form, the questionnaire was simply a fill in the blank format. To test the questionnaire, the CS 161 grader was asked to perform the experiment while being observed. Afterwards, the grader was asked about the process and the questionnaire, especially in light of her experience grading assignments turned in by the students from these classes.
The grader was an undergraduate student, at the Junior level in the Computer Science program at Portland State University. She did not have extensive previous programming experience. She was considered significantly advanced in comparison to the expected profile of the students from the SC 161 sections, but still a reasonable test of whether the materials would be too simple or too hard for the students. In this pilot of the experiment, the grader had answered 3 of the 6 questions correctly, and taking a total of 16 minutes. Her observations were:

- Although the CS 161 students had already covered integer arithmetic, they would have difficulty understanding the impacts of truncation during integer division, which was key to determining proper operation of the program.

- It was somewhat awkward to not have anywhere to record the cycles the program went through when working out an answer to the questionnaire.

- She commented that the program’s use of literals for constant values was not typical of the work currently being done by the students.

- She had asked what was meant by ‘How many guesses...’ in the questions on the questionnaire, which indicated this might not be clear to some students.
After this pilot, the questionnaire was changed significantly to improve its ease of use in the actual experiment. For each question, a small table was created to capture the intermediate steps of the student in answering that question. This eliminated the need for explaining how to count 'How many guesses..." and would provide some insight into the process used by students in answering each question. Based on the grader taking 16 minutes to complete the experiment it was felt that the typical subject would take most of the available class period (50 minutes, minus starting time delay for late arrivals of less than 5 minutes and about 5 minutes of instruction), since the grader was presumed to be substantially more experienced than the students. See Appendix B, Figure 14 for the final format of the questionnaire as actually used in the experiment.

The 'Background Questions' were converted to table format answers, with the first two questions being made into multiple choice. On the bottom of the Background Questions page, written instructions were added to explain the basic operation of the program, how integer division works with truncation, how to record start and stop times for each question, and an example of how to answer the questions on the questionnaire. These written instructions were added because, as pointed out by Moher and Schneider [43], some subjects respond better to verbal instruction, and some to written.
In Appendix B, Figure 15 shows this Background Questions and Instructions form.

For the actual experiment, packets were made up and distributed to the students. These were identical, except for the version of the program listing. There was no way for the subject to determine that there were differences in the packets they received, short of taking two packets and comparing the program listings. The packet consisted of:

- The Informed Consent form
- The Background Questions and Instructions sheet
- One of the two versions of the program listing
- The Program Understanding questionnaire

4.1.5 Execution

To ensure consistent execution of the experiment, the following steps were taken. First, a detailed script was created to introduce the materials to each class section, as shown in Appendix B, Figure 16. Second, the verbal instructions were as close to identical as possible when given to each section of the class. To aid in this, overhead transparencies were used instead of writing on the chalkboard. The example of how to answer questions was worked identically for each section using the transparency and a marking
pen. Third, the sequence of instructions was the same: start with reading and signing the Informed Consent Form, then step through the Background Questions, and lastly, once the subjects were ready, the instructions were reviewed and the example question worked.

On Monday, November 13, 1995, I was introduced to the course sections during their normal meeting times. It was stated that the course professor would not be at the class meeting the following day, and instead the experiment would be run. The students were informed that participation was voluntary, but they were encouraged to participate. The following day, I attended the class sessions at their normal times and locations, and administered the experiment.

In both sections of the class, when reminded that participation was voluntary, some students got up and left. I then asked the students if any were under 18 years of age. In the second section of the course, one student was excused from participation for this reason. The survey packets were distributed and the students were stepped through the informed consent process, background questions, and experiment instructions.

The students then began the questionnaire. In the first section, after a surprisingly short period of time, the first few subjects turned in their surveys
and left. During a quick review of these survey forms, the researcher noted that the first couple of subjects to return their surveys had done fairly well. After that, there were a number of subjects who seemed to have rushed through, guessing answers, and in a few cases seemed to have simply given up and left early. The researcher got the feeling, although no one mentioned it, that for some of the subjects, once the first few had finished, they were then under pressure to finish early also. In the second section, this behavior was repeated. In both sections, all but one of the subjects were done before the end of the class period.

4.1.6 Data Validation

As survey packets were turned in by students completing the experiment, the questionnaires were given a quick review. The questionnaire was checked for the reasonableness of the times listed for each question given the current time. With the format in which the questions were answered (having the subject list the sequence of guesses made by the program and responses from the user), it was possible during grading to determine if a student had properly understood the program and questions.

To score the questionnaires, a grading sheet and binary tree of the sequence of guesses the program would make were produced. To check for possible errors due to rounding up instead of truncating during integer
arithmetic, a second grading sheet and binary tree were created using rounding. These grading sheets were then turned over to an independent party to do the actual grading of the questionnaires. Although some subjects wandered well off from the correct answers, none of them appeared to round up instead of truncating.

Processing the data from the survey packets showed that some subjects had not answered all of the questions on the questionnaire, had not provided start and stop times for all questions, or had not provided all of the background information. Haas and Hassell [31] recommend that participants with partially completed questions or information be excluded from the analysis to maintain consistency. Four surveys were excluded because they lacked a GPA, and 11 surveys were excluded because of incomplete questions or times. The remaining data was entered into a spreadsheet for further analysis. In Appendix B, Figure 17 displays data set from the questionnaires, with the incomplete surveys removed.

4.1.7 Statistical Techniques

To select the proper statistical analysis techniques for the results of this experiment, three factors about this experiment were considered.
• Scale of the Measurements: As discussed by Fenton [29], there are five different scales of measurement, nominal, ordinal, interval, ratio and absolute. It is important to know the scale of data, because only with ratio and absolute data can the more sophisticated statistical methods, such as parametric tests like Student's t-test, be used.

• Distribution of the Data: Each of the parametric statistical methods assumes a specific distribution for the data being analyzed. The normal distribution (characterized by the 'bell shaped curve') is probably the most well known distribution and is assumed by tests like Student's t-test.

• Purpose and Design of the Experiment: The type of statistical analysis performed should be based on the purpose and design of the experiment in terms of the number of different situations being compared, and the type of comparison desired. For example, Student's t-test provides a simple test to see if two sets of data can be considered statistically similar or different.

The two measures used to evaluate ease of understanding the code styles are ratio scale. Ratio scale requires that data indicate ordering, the distance between values to be significant, and that a zero value be part of the measure to indicate none of that measure is present. The Score on the
understanding questions, and the Time to complete each question meet these requirements, so that a parametric statistical test could be used.

The distributions of the two measures used to evaluate ease of understanding the code styles are shown by the histograms in Figures 3 and 4. Although not normal distributions, they are regular in shape and have a single central tendency. As mentioned by Kirk [38] on page 493, "In general, however, parametric tests are robust with respect to departures from the assumptions of normality and homogeneity. Thus it is common practice in the behavior sciences to use the more powerful parametric tests even though the assumptions are only approximately fulfilled." With this in mind, the I decided that non-normality of the these distributions should not prevent the use of a parametric tested based on the normal distribution.

Figure 3: Distribution of Time to Complete Questionnaire
Given this experiment's purpose of comparing two code styles, and its design with one data set for each code style, the Student's t-test was an appropriate parametric statistical test to be used in evaluating the results.

To test for bias in the assignment of students to one or the other of the code styles, GPA, number of programs written and number of CS classes from the background questions were used in the analysis. GPA is a ratio scale measure, and Student's t-test can be used for comparison of groups of data.

The number of programs and number of classes measures were assigned a ranking of 1 through 5, but the distance between values is not significant which makes them interval scale, and therefore non-parametric data. Wonnacott and Wonnacott [51], on page 529, present a method of ranking non-parametric data and then using a parametric test as an analysis.
4.1.8 Analysis

To begin analysis of the results of this experiment, the background information on the students was reviewed to determine if they truly met the desired 'novice' programmer profile. Figure 5 presents a summary of this background information. CS 161 is intended to be an introductory course for undergraduate computer science majors with no prior computer experience. So it was assumed that the population of students in CS 161 would be primarily new freshman, with little computer or college experience. In fact, as can be seen from the student background summary in Figure 5, there is a large variation in programming experience, number of computer science classes, and the general college background of the students. Anecdotally at the end of one class session I talked to one student who reported over five years of professional programming experience. When I asked why this person was taking the course and not challenging it or requesting to substitute a more advanced course, he answered was "Well, I thought I would see what it was like, and it should be an easy A." This large variation in experience and college maturity of the students will be considered as a factor during analysis of the wide variation in performance.
To begin the analysis of the data, checks were made to test for bias between groups of students, both across the code styles, and across sections of CS 161. Figure 6 presents the average GPA of the subjects, by course section and overall across the two code styles, and the results of a t-test on this data. The t-scores do not show a statistically significant difference for GPA values, at the 90% confidence level, indicating that there was no unintended bias in the assignment of students to one code style or the other based on subject GPA.
Figure 6: Averages for GPA

<table>
<thead>
<tr>
<th></th>
<th>Function Style</th>
<th>N</th>
<th>Non-Function Style</th>
<th>N</th>
<th>t-score</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Section 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S D</td>
<td>3.1 (0.7)</td>
<td>10</td>
<td>3.3 (0.5)</td>
<td>14</td>
<td>-0.47</td>
</tr>
<tr>
<td><strong>Section 2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S D</td>
<td>3.2 (0.4)</td>
<td>8</td>
<td>3.3 (0.5)</td>
<td>8</td>
<td>-0.35</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S D</td>
<td>3.2 (0.6)</td>
<td>18</td>
<td>3.3 (0.5)</td>
<td>22</td>
<td>-0.58</td>
</tr>
</tbody>
</table>

SD = Standard Deviation

Figure 7 repeats the test for bias in the assignment of students by using the average ranked number of Programs completed by each subject. Again, there is no bias shown between code styles, at a 90% confidence level.

Figure 7: Average Number of Programs Written

<table>
<thead>
<tr>
<th></th>
<th>Function Style</th>
<th>N</th>
<th>Non-Function Style</th>
<th>N</th>
<th>t-score</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Section 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S D</td>
<td>3.6 (1.2)</td>
<td>10</td>
<td>3.9 (1.3)</td>
<td>14</td>
<td>-0.64</td>
</tr>
<tr>
<td><strong>Section 2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S D</td>
<td>3.3 (1.5)</td>
<td>8</td>
<td>2.6 (1.1)</td>
<td>8</td>
<td>0.97</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S D</td>
<td>3.4 (1.3)</td>
<td>18</td>
<td>3.5 (1.4)</td>
<td>22</td>
<td>-0.02</td>
</tr>
</tbody>
</table>

SD = Standard Deviation
Having detected no bias in the assignment of students to code styles, the formal hypotheses regarding a difference in average Score or average Time to complete the questionnaire between the code styles were tested. The results of these analyses are shown in Figures 8 and 9, below. The t-scores do not show a difference between the code styles, at a 90% confidence level, indicating that the null hypothesis (that there is no difference in the average score or time to complete between the treatments) cannot be rejected.

<table>
<thead>
<tr>
<th></th>
<th>Function Style</th>
<th>N</th>
<th>Non-Function Style</th>
<th>N</th>
<th>t-score</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Section 1</strong></td>
<td>4.0 (1.3)</td>
<td>10</td>
<td>4.3 (2.0)</td>
<td>14</td>
<td>-0.42</td>
</tr>
<tr>
<td>SD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Section 2</strong></td>
<td>3.6 (1.9)</td>
<td>8</td>
<td>4.3 (1.8)</td>
<td>8</td>
<td>-0.67</td>
</tr>
<tr>
<td>SD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td>3.8 (1.6)</td>
<td>18</td>
<td>4.3 (1.9)</td>
<td>22</td>
<td>-0.80</td>
</tr>
</tbody>
</table>

SD = Standard Deviation
The intent of this experiment was to test novice programmers for a difference in ease of understanding between two code styles. To get closer to a novice base of subjects, any student ranked 4 or 5 for Number of CS Classes or for Number of Programs was removed from the data set. This removed a total of 22 subjects, 10 from the function style and 12 from the non-function style. Removing this number of subjects from the data set reduced most of the groups, when broken out by their course sections, down to 4 students each. Groups of 4 students are too small to be statistically meaningful, so this analysis was done combining the course sections. The t-tests were then rerun, and the results are presented in Figure 10, below.

<table>
<thead>
<tr>
<th></th>
<th>Function Style</th>
<th>N</th>
<th>Non-Function Style</th>
<th>N</th>
<th>t-score</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Section 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SD</strong></td>
<td>9.9 (2.5)</td>
<td>10</td>
<td>11.6 (3.7)</td>
<td>14</td>
<td>-1.32</td>
</tr>
<tr>
<td><strong>Section 2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SD</strong></td>
<td>12.3 (6.2)</td>
<td>8</td>
<td>10.4 (3.2)</td>
<td>8</td>
<td>0.76</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SD</strong></td>
<td>10.9 (4.5)</td>
<td>18</td>
<td>11.1 (3.5)</td>
<td>22</td>
<td>-0.15</td>
</tr>
</tbody>
</table>

SD = Standard Deviation
Table: Combined Sections, Average Time and Score. Students Ranked 4 or 5 on CS Classes or Programs Excluded

<table>
<thead>
<tr>
<th></th>
<th>Function Style</th>
<th>N</th>
<th>Non-Function Style</th>
<th>N</th>
<th>t-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td>3.3</td>
<td>8</td>
<td>3.5</td>
<td>10</td>
<td>-0.25</td>
</tr>
<tr>
<td>SD</td>
<td>(1.6)</td>
<td></td>
<td>(2.5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>10.9</td>
<td>8</td>
<td>11.4</td>
<td>10</td>
<td>-0.24</td>
</tr>
<tr>
<td>SD</td>
<td>(5.5)</td>
<td></td>
<td>(3.4)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SD = Standard Deviation

Removing the more experienced students reduced the average score on the questions and had no effect on the average time to complete questions. The low value on the t-test for Score and Time again indicate that the null hypothesis cannot be rejected, even when only the novice students are examined.

4.1.9 Conclusions

The average scores for the understanding questionnaire were not significantly different between the two code styles. This suggests that the ease of understanding of each code style is not significantly different. There are several possible explanations for this result:

- It is possible that there is some difference in ease of understanding between these styles, but that this experiment did not sufficiently test the subjects, due to factors such as keeping the program listing to a single page, or making the questionnaire form too simple.
• It is possible that the subjects did not have sufficient motivation to seriously participate in this study. Many seemed to ‘give up and leave’ part way through the experiment, or simply guess at answers. In reading older experimental studies where students are the subjects, participation in the experiment is often considered for part of their course grade. With the HSRRC process, this is no longer possible, and alternative methods of motivation in addition to encouraging participation to further our understanding may be required.

• It is possible that there is no difference in ease of understanding of these styles for novice programmers.

An incidental finding from this experiment is that many of the students in the CS 161 course have a greater computer science, programming and/or college background than originally presumed. This information could be used the next time the CS curriculum is updated to separate the truly novice students from the more experienced students. Finally, there was a high degree of variance in the performance of the subjects in this experiment. Attempting to control for computer science class or programming experience reduced the questionnaire scores slightly, but did not affect the degree of variation in performance (see Figure 10, above).
Future experiments could be conducted to see if these results can be replicated. In a future experiment, consideration must be given on motivating the active and concentrated participation of the students. One approach would be to pay each subject a base amount to participate, say $5.00, and add on $1 for each correct answer on the questionnaire. Also in a future experiment, the program listings should be several pages long (perhaps 3 pages) to test whether there is a threshold for ease of understanding between these code styles and to reduce any 'brute-force' solution of the problem by the students.

4.2 Comparison to the Limitations and Issues

This section provides an evaluation of this example experiment against the limitations due to the nature of software development and issues in using controlled experiments, presented in Chapter 3.

4.2.1 Limitations

This example experiment shows that even for a study as simple as this, the limitations due to the nature of software development have an impact.

- Understanding of software development: This experiment was intended to be a straight-forward example. It was not intended to
require a substantial theoretical base or to provide a significant expansion of the understanding of software development. Determining if there is a difference in the ease with which novice programmers understand code styles is a reasonable study to take on with today's knowledge of software development.

- **Studying Complex Problems:** Since this was intended to evaluate a fairly straight-forward question in computer science education, the size of this study seemed reasonable during planning. As mentioned in the Conclusions section above, it turns out that even a study as this could be expanded to gain further insights into areas such as the proper motivation of student subjects.

- **Confounding of Factors:** The only apparent potential sources of confounding in this experiment would be subject motivation or length of the code style listings. Further studies should consider how these could be addressed. The experiment did not deal with a 'software development lifecycle' so there is not a problem of phase to phase interactions.

- **Surrogate Measures:** The program understanding questionnaire method, which was used as a surrogate measure of ease of understanding code styles, is a method that has been used and recommended for over a decade. Both Brooks [22] and Haas and
Hassell [31] recommend using properly designed questions as a measure for program understanding.

- **Subject Variability**: As previously discussed, subject variability in terms of performance was very high in this experiment. In this case, using the subject characteristics of previous programming experience and previous computer science courses as controls made no difference in the subject variation. As discussed above in confounding factors, motivation of the subjects may have been a contributing factor in the variation.

### 4.2.2 Issues

This example experiment was designed to avoid the issues in using controlled experiments.

- **Maintaining Equivalence Across Treatments**: As discussed in the section on execution of the experiment, significant efforts were taken to maintain equivalence across both code styles and course sections involved in the experiment.

- **Cycles of Use**: This experiment did not involve learning a new technique or method, since the students had previously seen both function and non-function styles of code in the CS 161 course. In
addition, the experiment was interested in the impact on novice programmers of the understanding of the two code styles.

- **Size of the Problem:** The program used as the object in this experiment was of ‘toy’ size. However, this is representative of the first programs presented to novice student programmers. It was intentionally small in size to fit on one page and not be overwhelming to a truly novice programmer.

- **Subject Selection:** The subjects selected were from the intended population of students in a beginning computer science class. However, the high variation in their backgrounds brings into question the idea that students in CS 161 are getting their first introduction to programming and software engineering.

- **Proper Statistical Analysis:** This experiment was designed to test the basic hypothesis that there would be a difference in scores on the understanding questionnaire between the two code styles. This calls for a comparison of the means of the two groups. Although the data collected was not normally distributed, it was regular in shape, with a single central tendency. As is common in studying the behavioral sciences, a Student's t-test of the means was used, even though the t-test's assumptions were only approximately met.
Proper Interpretation of Results: As covered in the analysis section, the results of this experiment were statistically inconclusive. The null hypothesis was not disproved, so nothing conclusive can be said about whether one of the code styles is different from the other.

4.3 Lessons Learned

The lessons learned from this example experiment are at two levels. The first level is the detailed lessons which could be applied to a future follow-on or related study. The second level is the generalizations that could be applied if considering the use of controlled experiments to study an aspect of software development in an industrial environment.

Detailed lessons learned: If I were conducting another empirical study such as this controlled experiment, I would focus on three key areas. First, I would identify what assumptions I was making about the subjects, such as their prior experience, and then pre-screen them to determine if these assumptions were correct. As part of this pre-screen, it would be valuable to interview a few of the potential subjects in detail to see if there were other areas of their backgrounds which might impact the experiment. Based on the results of the pre-screen, I would decide if I really had the proper subjects for my proposed study. Second, I would consider how the subjects could be more directly
motivated to participate and put their full efforts into the experiment. As part of the pre-screening of subjects, ideas for motivation could be tested. Third, I would conduct additional reviews of the experiment’s design and materials with a critical audience. Then I would conduct additional pilots of the experiment to ensure that the materials and measures were properly refined.

- **Generalized lessons learned**: If I were considering conducting a controlled experiment in an industrial environment, I would focus on two key areas. First, I would ensure that I had a clear question to answer. For example, a controlled experiment could be effective to compare the defect finding effectiveness of two testing methods. See Section 5.1 for the types of studies where controlled experiments will be most effective. Second, I would be sure that I fully understood the environment where the experiment was to take place, and had found measures that were validated for detecting the effect I was looking for in the experiment. Before designing and executing the experiment at the center of my study, I would have done a variety of exploratory evaluations using a mix of empirical methods. In the example of comparing two testing methods, I would start by doing a survey to determine how the subjects perform testing today. From this I would determine the type and duration of training and practice required as
preparation for the experiment. See Section 5.3 for a more detailed description how different types of empirical methods could be combined to build understanding of the environment and factors to be controlled and measured.
5. EFFECTIVE USE OF CONTROLLED EXPERIMENTS IN SOFTWARE

Controlled experiments require that the researcher have a high degree of control over the environment where the experiment is carried out. To get this high level of control, it is best if the difficulty of establishing and maintaining this control is low. Experiments also depend on repeated trials to build enough data for statistical validity. To aid in getting sufficient replication, it is desirable that the cost of replicating the trials be low. These factors directly affect the types of studies and environments where controlled experiments will be the most effective research method, from both their ability to provide results, and the cost of gaining those results.

In this chapter I present my conclusions on how controlled experiments can be used effectively to study software development. The key factors in this effective use are: Pick appropriate problems to study with controlled experiments. Consider the use of student subjects to start the exploration of a problem, if appropriate. Use controlled experiments as part of a larger set of empirical methods in developing an understanding of the subject area of a study and determining the proper environmental factors to control and measures to use.
5.1 Studies Appropriate for Controlled Experiments

Although controlled experiments could be used to study almost any subject within software development, because of the relatively tight control required and the repetition of trials as mentioned above, they are most commonly used to study smaller objects in more laboratory-like settings. The key benefits of controlled experiments are in providing answers to specific questions or where a clear cause/effect relationship is to be determined, with other factors ruled out. With these factors and benefits in mind, here are several areas of software development where controlled experiments are an effective empirical tool.

- **Theories:** One of the best uses for controlled experimentation is to validate theories. Once a theory has been proposed based on exploratory studies of some problem area, the theory needs to be validated and potentially refined. If the theory makes specific predictions that are of interest to the researcher, these can be stated as hypotheses and controlled experiments constructed to confirm or reject these hypotheses. Of course, it is very possible to have a theory that requires this validation to take place in an environment where conducting a controlled experiment is expensive and difficult to control. For example, a theory of how professional versus novice programmers recognize elements of programs (as individual tokens,
or as sets of tokens in 'plans.' See Soloway and Ehrlich [15]) requires
the participation of professional programmers, who are generally hard
to enlist as subjects. The need to conduct controlled experiments in
industrial environments can be mitigated by the fact that if a theory
makes detailed predictions, they can be individually validated in
specific experiments. Therefore, no one experiment in that industrial
environment need be extremely large.

• Models: Like theories, models need to be validated and possibly
refined. As with theories, controlled experiments are an effective
method for validating a model because the specific predictions can be
used as the basis for the formal hypotheses, which can then be
confirmed or rejected. For example, if a company had been using the
COCOMO model (Boehm [20]) calibrated for their environment and a
particular type of projects, when they got a contract for a different type
of projects they could conduct a controlled experiment comparing the
existing calibration of COCOMO to the actual results for their first few
projects under the new contract. If the results of the new projects
differed from the COCOMO model by a statistically significant amount,
they would know that the COCOMO model needed to be recalibrated
for the new type of projects. They might also be able to use the data
from the experiment with the new projects to perform regression
analyses and recalibrate the COCOMO model for this new project type. The same issues for validating theories apply to validating models of software development in industrial environments.

- **Metrics:** A metric claims to measure some attribute of software or the software development process. The specific claims of a metric can be validated as a hypothesis in a controlled experiment. As mentioned under 'Surrogate Measures' in Section 3.1.4 above, an important area of on-going research could be the validation of surrogate measures for commonly desired software development attributes such as 'software complexity' or 'software quality.' For example, if a metric is proposed that claims to measure software complexity, a series of experiments could be conducted to determine how the new metric compares to existing 'complexity' metrics to understand what potential attributes of complexity it captures in relation to the existing metrics.

- **Methods and Tools:** Controlled experiments can be used to compare, evaluate, and understand different methods and tools. To initially explore a new method or tool, an empirical method such as a case study may be more cost effective. See Section 5.3 below for more on this subject. However, to explore specific details, such as the trade-offs of different designs for the formal inspection process versus cost (in resource and schedule) and defect finding effectiveness (see
Porter et al. [9]), a controlled experiment can provide effective control of extraneous factors and provide detailed data on these interactions.

5.2 Effective Studies using Student Subjects

The use of student subjects in controlled experiments is generally desired because they are more readily available (for academic researchers), their time is less expensive than that of professional programmers, and they can be easier to enlist for an experiment and to motivate during the experiment. There are several situations where the use of student subjects in a controlled experiment is appropriate.

- **Studies appropriate for student subjects:** The most obvious situation where students are appropriate subjects is one where the performance or learning processes of novice programmers is under study. A second situation is where the level of expertise or experience of the subjects is not a factor that needs to be controlled, because the purpose of the study is independent of these factors. A third situation is the use of graduate-level students to initially explore an area, providing an understanding of some of the possible interactions between variables, and helping to determine the appropriate measurements. The study would then be taken to an industrial environment and replicated using professional programmers.
- **Developing students as surrogate subjects**: An area for future research is better understanding when students might be effective as 'substitute' subjects for professional programmers in experiments. In the case of project management experiments, there has been some research showing that graduate level students may be acceptable substitutes for professional managers (Bettenhausen [19] and Remus [47]). In fact, these studies found that the use of graduate students actually reduced the variability normally experienced from environment to environment using industrial organizations. If similar circumstances can be determined for software development, the use of student subjects could actually be a benefit by providing the researcher with better control of subject characteristics.

As pointed out by Curtis [25], a key issue in subject performance is the "relevant knowledge possessed" by the subject. Curtis also points out that the reason programming tests have generally not been successful as a measure of programmer performance (for example, as a hiring criteria) is that a sufficient job analysis was not done of the actual skills and cognitive requirements of different software development activities, such as translating design into code or debugging. And, as Curtis and Iscoe [26] and Potts [45] point out, this relevant knowledge needs to include domain-specific information. With these
considerations in mind, it could be possible to create situations where students are actually more effective subjects than professionals.

The general design of this new model of using student subjects would include sufficient training and skill development through practice to create the ‘relevant knowledge’ equivalent to that of a professional in a specific target area of study, including software domain. Because this training period would most likely have to be months long, and it would be best for the subjects to have general maturity in software development, graduate students would be best. Having created this pool of student subjects with the proper relevant knowledge, the researcher could then have them perform the experimental tasks on significantly sized problems.

For example, to study the difference in ADA versus C++ for some aspect of object-oriented programming, two groups of graduate students could be selected. If possible, students already familiar with one of the two languages would be placed into the group to use that language (for example, a student who used C++ in their undergraduate studies would be placed into the C++ group). Then, the students would be run through an intensive training and practice period on the language for their group and the specific domain of the problems to be studied in the experiments. This would develop their
'relevant knowledge.' During and at the end of the training period, some type of specifically focused testing of the 'relevant knowledge' would be used to confirm that they had indeed developed the desired skills, and if necessary, some subjects could be removed from the study. Then the actual experiments could be conducted, using realistic sized problems, such as actual (or equivalent) programs from the software domain area.

Although setting up this elaborate training program would be a major effort on the part of the researcher, once it was in place, it could be maintained in operation over multiple years, with new students being added as they arrived to replace graduating students. With this trained pool of subjects, a variety of related problems could be efficiently studied. For their effort, the researcher would gain several advantages:

- The subjects from this pool would actually provide better control for variation in 'relevant knowledge' and skill than randomly selected professional programmers from one or more industrial environments.

- Once this program was going, the researcher could likely study more problems in a given period of time than they would be able to
using actual industrial settings because the setup and management would be fully under the researchers control in the proposed environment.

- The program would bypass the problems where industry simply doesn't have the time, resources, or interest in participating in research.

5.3 Improving ‘Cost-Effectiveness’ and Generalization

Making controlled experimentation more cost effective and improving the generalization of results go hand in hand. In addition to the cost of performing an initial experiment is the cost of replication to re-validate results or expand generalization. To help reduce the total cost of controlled experimentation and still provide more generalized results, there are two strategies which can be used in combination:

- **Combine empirical methods in a larger study:** Look at ways to combine empirical methods, including controlled experiments at the proper point, to explore the area of study. The exact sequencing of different empirical methods would be dependent on the specific study. For example, where experiments could be used at the end of the study, a series of case studies might be performed to gather background and understand the larger picture related to the
effectiveness of a specific software development methodology. Once it appeared that the parameters of the methodology's use were understood and cause/effect relationships had been hypothesized, controlled experiments could be designed and conducted to test the specific hypotheses and gain further insight into the meaning of the observations from the case studies.

As an example where controlled experiments might be used first, a set of controlled experiments could be conducted to determine the fault detecting capabilities and related factors governing the use of several software testing methods. This set of experiments would likely all take place in a specific laboratory or development environment. To generalize the results to other environments, and to see if some additional environmental factors might impact fault detecting capabilities in production use, a series of case studies could be conducted in multiple environments. Case studies would be easier to arrange and conduct than experiments, although they would lose some precision in their findings. The case studies could provide results that would indicate if the original experimental findings hold across multiple environments, and if they did not hold, might indicate the factors that confounded the results.
• **Carefully tune the controlled experiment(s):** It is not really possible to entirely leave out an aspect of controlled experiments and still have a controlled experiment. For example, if you are not able to repeat the control and experiment situations enough times, a valid statistical analysis cannot be done. However, your study could still make a very good case study. There are specific steps which can be taken to reduce the overall cost of performing a controlled experiment and to make its execution more efficient. These steps involve doing pre-work in planning the experiment, so that the 'minimum' amount of experimentation is required for the desired outcome, and carefully monitoring the experiment as it is executed.

Perhaps the best way to manage the cost of a controlled experiment is to carefully consider the goals of the study, and determine if there are ways to narrow down the focus for the actual experiment. For example, estimate the likely variation in the data, so that a more accurate determination of how many data points are needed to provide the desired level of confidence. In this way the number of subjects, objects, or trials might be reduced. Can the problem be carefully examined in advance and factor combinations or treatment levels be eliminated because they do not appear to be as valuable to the final results? For example, the removal of one level from a
treatment is multiplied by the number of levels of the other factors it is
crossed with and therefore can have high leverage in reducing the
overall cost of an experiment.

The efficiency of a sequence of experiments can be improved during
its execution by analyzing partial results as they become available and
adjusting or canceling the subsequent experiments. For example, if
the data collected part way through has had less variability than
anticipated, and the desired confidence level has been reached for
some portion of the experiments, it may be possible to stop that set of
experiments early. If a specific treatment is not showing the desired
result, it can be abandoned, or the experiment restarted if a cause for
the lack of result is suspected.

5.4 Summary

The software practitioner needs to understand how software
development can be done in the most effective manner. One way to gain this
understanding is to perform empirical studies, including controlled
experiments. When considering the use of controlled experiments, it is
important to:
• Review the types of research where controlled experiments are most effective, and determine if a controlled experiment is the correct empirical method for the study.

• Consider ways that overall cost can be reduced and efficiency improved by the combination of different types of empirical methods for different parts of the study.

• Consider ways a controlled experiment can be designed and carried out that minimize the effort involved.

• Plan and manage for each of the limitations and issues.

• Present a full report of their study, including enough details of the design and execution so that others can evaluate their results, and consider replicating the experiment.

Controlled experimentation is a powerful research tool. It is one of a set of empirical methods which can be used to further the understanding of a scientific area through quantitative data collection and analysis. Controlled experiments have been one of the bases for progress in many scientific fields. Appropriate use of controlled experiments can provide a major improvement in software development research.
This bibliography is divided into two sections. The first section contains annotated references to published reports of controlled experiments studying software development. The second section contains references related to empirical studies and controlled experimentation. References in the used in the text of this thesis may refer to either section of this bibliography.

**Controlled Experimentation in Software Development**


33 graduate students acted as ‘project managers’ in a simulation of a software development project. They were divided into three treatment groups, each group given an initial productivity estimate below, equal to, or above that actually built into the simulation. They were asked to track and report the progress of the project based on periodic reports they received on resource consumption and work accomplishment during the simulation. They were told that based on the productivity estimates they reported back, adjustments would be made to resources and schedule (in reality, their input was ignored, and in all cases the simulation stuck to the fixed productivity value). The goal of the experiment was to see if, over the course of the simulation, they would change their productivity estimates to close in on the value built into the simulation. The results were
that the subjects stuck to the original productivity estimates they were given, and did not converge on the actual value.


45 students from two courses were formed into 1 person or 3 person teams and each completed the same software development project. 7 of the 3 person teams were specifically instructed to use a top-down design approach, with code reading and walkthroughs, using a chief programmer team organization. The remaining teams 1 or 3 person teams were left to create perform the task as they saw fit (ad-hoc). 35 variables were measured to identify a possible effect of the methodology treatment versus the ad-hoc treatment. Of these, approximately 1/2 detected no effect between the three groups of subjects. In addition to many detailed observations, the researchers feel the results support the claim that disciplined methodology effectively improves both the process and product of software development.


42 advanced students and 32 professional programmers applied one of three techniques: (1) code reading, (2) functional testing, or (3) structural testing to unit sized programs. They found that for professional programmers, code reading was most effective at both fault detection and the rate of fault detection. With the students, fault detection and detection rate were less distinguished across the three methods. They also found results varied for different types of code and types of faults.
36 professional programmers performed three maintenance tasks on three different programs (one on each). The three programs varied from the unstructured original, to a partly restructured intermediate (long jumps were eliminated, and control structures were limited), to a fully functionally block structured program. They found that overall, the fully restructured program was fastest to maintain, although not fastest for each individual task. They also found that in making the maintenance changes, the subjects made the fewest 'primary' errors in the original unstructured program, but also created the most 'ripple-effect' errors in it.

80 undergraduate students were placed in teams of five and asked to create test specifications for two different systems, once using manual methods, the other time using the GDSS tool. The subjects developed the same number of individual ideas (working independently at the start of the task), but groups using the GDSS tool ended up with more tests, and their tests were judged better (by three independent judges). The subjects also reported preferring the automated tool to the manual method.

A series of six experiments were conducted of the cloze testing method (where portions of a program listing are deleted and the subject is asked to 'fill in the blanks'). From 36 to 44 students (upper division to graduate level) were used per
experiment. The first two experiments (1 and 2) produced ‘comprehensibility’ ratings for two programs (‘easy’ and ‘hard’) using a program understanding questionnaire. The test score and time to complete were statistically significant between the ‘easy’ and ‘hard’ program versions. The next two experiments (3 and 4) used the same programs, but with the cloze ‘fill in the blank’ as the test instead of a questionnaire. Experiment 3 again showed that that scores and time for the ‘easy’ program version were statistically higher than those for the ‘hard’ version. Experiment 4 did not show a statistically significant difference in score or time between the ‘easy’ and ‘hard’ versions. This was attributed to removing 1 in 5 lines during experiment 3, and 1 in 3 lines during experiment 4. A closer examination of the 1 in 3 removal’s affect showed that information required to understand that specific program had been removed. The last two experiments (5 and 6) were conducted on programs with a 1 in 3 removal, but where which lines were removed was adjusted to minimize the removal of program specific information. For these last two experiments, the difference between ‘easy’ and ‘hard’ was statistically significant. The results show that cloze testing can be used as a program understanding measure, but that care must be taken in deciding which lines are removed from the program listing.


12 upper-division undergraduate students participated in 4 design and coding tasks each. The first two tasks were to familiarize the subjects with the design and coding standards, and a third as a pre-test task to provide a baseline. Then the experimental group was given instruction on abstract data types, and both experimental and control groups performed a fourth design and coding task. The subjects using the experimental treatment were found to have taken (statistically) significantly longer to complete the fourth task, although their resulting programs were no better than those of the control group, based on the measures used. The experimenters then suggest that this result supports their work on an automated code generator for abstract data types, but this is not tested anywhere in the experiment.

48 graduate students participated in one of two replications (24 in each replication) of an experiment where each individual participated in two inspections, one each of two specifications, using one of three defect detection methods (ad-hoc, checklist, or scenarios). They found that scenarios were superior to the other methods in fault detection rate, and help inspectors focus on specific fault classes (without missing other fault classes). They also found that the checklist method (which is the most common practice in industry) was no more effective than ad-hoc, and that on average, the fault collection meetings added nothing to fault detection effectiveness.


11 experienced developers have conducted 34 (of 64 planned) code inspections, varying the number of inspectors, number of teams, and whether defects found in a first inspection are repaired prior to the second inspection. They are measuring several variables to understand the trade-offs between cost (resources and time) of the different inspections and their defect finding effectiveness. To date they have found that only 15% of the defects found by inspectors during preparation would compromise the functionality of the delivered system. Although they found the defect reporting meeting added 33% to the defects found, they got an even better gain at equal or lower cost by having a second inspection. Finally, they found that 2 person inspections were almost as effective as 4 person inspections.
82 junior through graduate level students answered questions to test their understanding of a simple, an intermediate and a complex algorithm using a structured flowchart or pseudocode. They were then re-tested using whichever (pseudocode or flowchart) information source they had not previously used. The experiment was carefully structured and carried out using facilities and help from the psychology department, and analyzed with assistance from the statistics department. Scanlan found that for all three levels of complexity, time to complete the test was (statistically) significantly shorter using flowcharts than pseudocode. In the study by Shneiderman et al. [12], time was not a variable with was controlled, which could be one factor accounting for the difference in results.

27 graduate students were organized into nine 3-person teams to perform n-fold (n-different teams inspect the same product, merging their results together through a common moderator) inspections on a 22 page requirements document with 99 faults injected into it. Their results were that on average, a single team found about 35% of the injected faults, and when the work of all 9 teams was combined, 78% of the faults had been found. Although they classified the faults into 6 different types, they were not able to show (statistically) that any particular fault type was easier or harder to detect.
A series of experiments using groups of from 53 to 70 beginning to intermediate students to measure for an effect in using flowcharts as an aid to program composition, understanding, debugging, and modification. There was not a statistically significant difference in performance of the tasks by students with or without flowcharts. Although they conjectured that detailed flowcharts may be a redundant source of information, they also recommended that future experimentation was needed with macro and structure flowcharts to see if these would have an effect, and with larger programs (over 1000 source lines). Partly because they had started out to demonstrate the value of flowcharts, this study contributed to the decline in the use of flowcharts for documenting algorithms. See Scanlan [10] for a counterpoint to this study.

34 LISP and 48 Pascal undergraduate students participated in 2 experiments where they solved two recursive problems in each. In the first experiment one problem was presented recursively and the other iteratively, and the subjects could choose either recursive or iterative constructs for the solution. In the second experiment, both problems were presented either recursively or iteratively, and the subject was required to solve them with a specific method. The study found that the programming language's support for a control construct had greater impact than the cognitive fit of the construct to either the problem representation or solution.
39 novice, 90 intermediate, and 48 advanced students were asked to write a plan for solving a data reading and calculation problem. They were then asked to write a program to solve the problem in either regular Pascal or Pacsal-L (which provides a exit from the middle of a loop). A Read/Process strategy was preferred for both the plans and the programs, although as the student's experience increased, they increasing moved to the Process/Read strategy required by WHILE structures in languages like Pascal.

Two experiments were conducted. In the first, 94 novice and 45 advanced student programmers were asked to fill in a blank line in one of two versions of a program. In the second, 41 professional programmers were asked to recall, verbatim, one of the two versions of the program. The program versions varied in that one was "plan-like" following normal programming conventions, and the other was unplan-like. The studies found that experience programmers perform much better with the plan-like structured programs. On the unplan-like program, the experienced programmers performed no better than the novice programmers.

6 students were observed applying one of three different system analysis methods (structured, Jackson System Design, and object-oriented) to three separate problems. Over the three applications, their performance was measured to check for a learning curve effects. The results reported were that the students were best able to apply the structured methodology, and least able to apply the object-oriented methodology. Also,
significant learning was detected only for the structured methodology over the course of the three applications.

**Related References**


[27] B Dalio, Director of Engineering Standards, Mentor Graphics Corporation, a private communication regarding the life cycle of ideas in software development, March 1996


A: REFERENCES FOR CONTROLLED EXPERIMENTS

It is important for a working engineer or manager considering using a controlled experiment to know what the proper steps are in designing and carrying out a controlled experiment. If these steps are not properly followed, the validity of the results will be reduced, and could even lead to incorrect conclusions.

There are a small number of articles which discuss how to perform controlled experiments in the software development field. These include Basili et al. [18] which provides a good framework, especially the detailed breakdown of elements in what they call the Definition phase. However, they do not provide a detailed explanation of how to actually apply this framework. They also provide an exhaustive listing of articles reporting controlled experiments from the late 1970s through 1985, which can serve as a source of examples. Mohamed et al. [42] provide another framework, translating the elements that make up a controlled experiment into the language of software engineering, and discussing the application of controlled experiments in a software development environment. Like Basili, they do this at a terminology level, and do not provide the details to allow actual application. Adelman [17]
presents a basic description of controlled experiments for use in evaluating computational Decision Support Systems, along with a good analysis of the types of statistical validity which need to be considered, and how to deal with threats to them. Fenton et al. [30] provides a good summary of the possible problems that can occur in using empirical methods to study software development. In a series of articles Pfleeger [44] takes the approach of presenting a basic how-to guide on using controlled experiments and analyzing the results in software. Of the articles listed here, Pfleeger's set of articles provide the most thorough explanation for how to actually conduct a controlled experiment.

To provide backup to these articles from the software development literature, there are a large number of text books which cover statistics and experiment design, along with the proper analysis techniques for the data from experiments. General statistical texts, such as Wonnacott and Wonnacott [51], provide a good introduction to the statistics needed for experiments, including non-parametric analysis methods. Statistical texts focused on experimental applications in behavioral sciences, such as Kirk [38], provide a balance of statistical depth for understanding with clear application examples to studies involving human behavior studies.
The use of experiments in psychology research has matured significantly over the past several decades. There are a number of handbooks in the psychology literature, such as Martin [41] and Keppel et al. [36] which discuss in detail how to design and carry out controlled experiments. Martin does an excellent job of explaining many details of designing and conduction controlled experiments, with simple, easy to understand examples. Keppel et al. focus on the statistical analysis of experimental results, and the proper handling of the different types of designs of experiments. These and other books from psychology provide an excellent model for how controlled experiments can be applied to software development studies.
B: MATERIALS AND DATA FROM THE EXAMPLE EXPERIMENT.

The following pages contain the materials and results from the example experiment.


```cpp
// GuessNumber!
// Program that will guess a user picked number between 1 and 100
#include <iostream.h>

int lowbound = 1;
int highbound = 100;
int guess = (100 + 1) / 2; // integer division truncates
char answer;

int main()
{
    // explain the game and ask to begin
    cout << "I will try to guess a number between 1 and 100 "
         << "that you have picked." << 'n';
    cout << "Press any key when you have picked a number." << 'n';
    cin >> answer;

    // make first guess, get answer, verify answer is c, l, or h
    cout << "My guess is " << guess << " How did I do?" << 'n';
    cout << "Reply: 'c' = correct; 'l' = low; 'h' = high" << 'n';
    cin >> answer;
    while (answer != 'c') && (answer != 'l') && (answer != 'h')
    {
        cout << "Reply: 'c' = correct; 'l' = low; 'h' = high" << 'n';
        cin >> answer;
    }

    while (answer != 'c')
    {
        // adjust guess based on user's "low" or "high"
        if (answer == 'h')
        {
            highbound = guess;
            guess = (highbound + lowbound) / 2; // integer division truncates
        }
        else
        {
            lowbound = guess;
            guess = (highbound + lowbound) / 2; // integer division truncates
        }

        // display guess, get answer, verify answer is c, l, or h
        cout << "My guess is " << guess << " How did I do?" << 'n';
        cout << "Reply: 'c' = correct; 'l' = low; 'h' = high" << 'n';
        cin >> answer;
        while (answer != 'c') && (answer != 'l') && (answer != 'h')
        {
            cout << "Reply: 'c' = correct; 'l' = low; 'h' = high" << 'n';
            cin >> answer;
        }
    }

    cout << "Thanks! That was fun." << 'n';
    return 0;
}
```
Figure 12: Function Style Program

```c
// GuessNumberF
// Program that will guess a user picked number between 1 and 100
#include <iostream.h>

int lowbound = 1;
int highbound = 100;
int guess = (100 + 1) / 2; // integer division truncates
char answer;

int main()
{
    // explain the game and ask to begin
    cout << "I will try to guess a number between 1 and 100:"
    << "that you have picked." << "\n";
    cout << "Press any key when you have picked a number:" << "\n";
    cin >> answer;

    // make first guess and get answer
    checkGuess();
    while (answer != 'c')
    {
        changeGuess();
        checkGuess();
    }

    cout << "Thanks! That was fun." << "\n";
    return 0;
}

void checkGuess()
{
    // display guess, get answer, verify answer is c, l, or h
    cout << "My guess is " << guess << " How did I do?" << "\n";
    cout << "Reply: 'c' = correct; 'l' = low; 'h' = high" << "\n";
    cin >> answer;
    while (answer != 'c') && (answer != 'l') && (answer != 'h')
    {
        cout << "Reply: 'c' = correct; 'l' = low; 'h' = high" << "\n";
        cin >> answer;
    }
}

void changeGuess()
{
    // adjust guess based on user's "low" or "high"
    if (answer == 'h')
    {
        highbound = guess;
        guess = (highbound + lowbound) / 2; // integer division truncates
    }
    else
    {
        lowbound = guess;
        guess = (highbound + lowbound) / 2; // integer division truncates
    }
}
```

INFORMED CONSENT FORM

I, ____________________________, agree to take part in this research project on Programming Style’s Impacts on Program Comprehension.

I understand that the study involves providing background information on my computer science and programming experience, reading a small program listing, and answering a series of questions about the program.

Mark Johnson, principle investigator, has told me that the purpose of this study is to learn whether programming style differences can make a difference in the ability of a reader to comprehend the actions of a program.

I will not receive any direct benefit from taking part in this study. However, this study may help increase knowledge that may help others in the future.

Mark Johnson has offered to answer any questions I have about the study and what I have to do.

He has promised that all information I give will be kept confidential to the extent permitted by law, and that all of the people in the study will remain anonymous.

I understand that I do not have to take part in this study, that the professor teaching this course will not know about my participation or non-participation in the study, and that this will not affect my course grade or my relationship with Portland State University.

I have read and understand the above information and agree to take part in this study.

Date: _________________ Signature: ________________________________

If you have concerns or questions about this study, please contact the Chair of the Human Subjects Research Review Committee, Research and Sponsored Projects, 105 Neebenger Hall, Portland State University, (503)725-3417.
Figure 14: Experiment Questionaire

Program Understanding Survey

1. List the guesses the program would make to guess the number 13.
Start Time:
Program's Guess
User's Response (h, l, c)
Finish Time:

2. List the guesses the program would make to guess the number 81.
Start Time:
Program's Guess
User's Response (h, l, c)
Finish Time:

3. There are 2 numbers that will be found in exactly 2 guesses. List the program's guesses to each.
Start Time:
Program's Guess
User's Response (h, l, c)
User's Response (h, l, c)
Finish Time:

4. List the guesses the program would make to guess the number 28.
Start Time:
Program's Guess
User's Response (h, l, c)
Finish Time:

5. List 1 number that will be found in exactly 6 guesses.
Start Time:
Program's Guess
User's Response (h, l, c)
Finish Time:

6. List the maximum number of guesses the program will require for any number in the range from 1 to 100
Start Time:
Maximum Number of Guesses
Finish Time:

113
Figure 15: Background Questions and Instructions

Programming Style's Impacts on Program Comprehension

Background Questions

1. Number of previous computer or programming courses (check 1 box)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3-4</td>
<td>More than 4</td>
</tr>
</tbody>
</table>

2. Number of programs written (check 1 box)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>2-3</td>
<td>4-6</td>
<td>7-12</td>
<td>More than 12</td>
</tr>
</tbody>
</table>

3. List GPA (if first or second term of college list high school GPA)

<table>
<thead>
<tr>
<th>College GPA</th>
<th>High School GPA</th>
</tr>
</thead>
</table>

4. List amount of college completed

<table>
<thead>
<tr>
<th>Credit Hours</th>
<th>Years</th>
</tr>
</thead>
</table>

Instructions for Survey

Attached is a program listing and a survey to check your understanding of the program.

The program is a simple number guessing game. It asks the user running it to pick a number between 1 and 100. Then the program makes a guess at the number that the user has picked. After the program prints its guess, the user is asked to reply with "I" to indicate the program's guess was lower than the number picked by the user, or "h" to indicate the guess is higher, or "c" to indicate the guess is correct. If the user responds "I" or "h", the program changes its guess and tries again. When the user responds "c", the program ends.

Please read the program and then answer the survey. While answering the survey, please note:

- The program uses integer division, which truncates answers. For example: 101 / 2 = 50.5, which is truncated to 50

- Write down your start and stop time for each question, to the nearest minute, in the boxes provided. Use the clock in class or your watch. It is important that you use the same clock or watch each time, so that the difference between start and stop times indicates the time it took you to answer a question. See the example below.

- Use the space in the boxes to record each guess made by the program and the user's response (h = high, l = low, c = correct). For example, to answer "List the guesses the program would make to guess the number 75?" you would fill in the answer boxes as follows:

<table>
<thead>
<tr>
<th>Start Time:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program's Guess</td>
</tr>
<tr>
<td>User's Response (h, l, c)</td>
</tr>
<tr>
<td>Finish Time:</td>
</tr>
</tbody>
</table>
Introduction Script

1. Who are you?
   - Mark Johnson of Mentor Graphics
   - Collaborating with Warren on study to make programs easier to understand

2. Excuse anyone who is under 18 years old

3. Explain Informed Consent
   - Participation is voluntary
   - Will not affect grade one way or the other
   - Need them to complete form
   - When turn in survey packet, remove form and place in box

4. Explain the program
   - Simple number guessing game
   - User (running program) picks a number 1 to 100 (but don't tell program)
   - Program will make a guess
   - If guess is lower than number picked, user tells program 'l'
   - If guess is higher than number picked, user tells program 'h'
   - Program guesses another number
   - When guess is finally correct, user tells program 'c'
   - Program thanks user and ends

5. What to do with Packet
   - Complete informed consent
   - Fill out background questions
   - Read instructions
   - Scan through program listing
   - Example question: "List the guesses the program will make to guess the number 75"
## Figure 17: Survey Data Set (part 1): Background Information

<table>
<thead>
<tr>
<th>Class Section</th>
<th>Program Version</th>
<th>Number of CS Classes/Rank</th>
<th>Number of Prags/Rank</th>
<th>College GPA</th>
<th>High School GPA</th>
<th>Selected GPA</th>
<th>Credit Hours</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
<td>3.33</td>
<td>3.50</td>
<td>3.33</td>
<td>140</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>4</td>
<td>4</td>
<td>2.00</td>
<td>2.00</td>
<td>8</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>2</td>
<td>4.45</td>
<td>3.50</td>
<td>3.45</td>
<td>38</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>2</td>
<td>4</td>
<td>3.30</td>
<td>3.30</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>3</td>
<td>4.25</td>
<td>2.50</td>
<td>2.50</td>
<td>200</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>5</td>
<td>4.60</td>
<td>3.90</td>
<td>4.00</td>
<td>30</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
<td>3.82</td>
<td>3.97</td>
<td>3.82</td>
<td>27</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
<td>3.80</td>
<td>3.80</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>1</td>
<td>3</td>
<td>3.60</td>
<td>3.60</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>3</td>
<td>5</td>
<td>3.90</td>
<td>3.90</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>2</td>
<td>3.25</td>
<td>3.25</td>
<td>185</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>4</td>
<td>5</td>
<td>3.70</td>
<td>3.33</td>
<td>3.70</td>
<td>48</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>1</td>
<td>3</td>
<td>3.70</td>
<td>3.50</td>
<td>3.70</td>
<td>45</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>3</td>
<td>5</td>
<td>2.70</td>
<td></td>
<td>2.70</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>3</td>
<td>5</td>
<td>3.40</td>
<td>3.40</td>
<td>13</td>
<td>0.3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>1</td>
<td>3</td>
<td>3.70</td>
<td>3.90</td>
<td>3.70</td>
<td>200</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>1</td>
<td>5</td>
<td>2.50</td>
<td></td>
<td>2.50</td>
<td>111</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>2</td>
<td>5</td>
<td>3.20</td>
<td>3.20</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>1</td>
<td>2</td>
<td>3.50</td>
<td>3.50</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>5</td>
<td>5</td>
<td>3.00</td>
<td>2.40</td>
<td>3.00</td>
<td>25</td>
<td>0.5</td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>1</td>
<td>2</td>
<td>3.00</td>
<td>3.49</td>
<td>3.00</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>5</td>
<td>5</td>
<td>2.50</td>
<td></td>
<td>2.50</td>
<td>85</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>2</td>
<td>3.41</td>
<td>3.41</td>
<td>182</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>3</td>
<td>4</td>
<td>3.00</td>
<td>3.00</td>
<td>180</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>1</td>
<td>4</td>
<td>3.50</td>
<td>3.50</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>1</td>
<td>1</td>
<td>4.00</td>
<td></td>
<td>4.00</td>
<td>128</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>1</td>
<td>2</td>
<td>2.80</td>
<td></td>
<td>2.80</td>
<td>110</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>1</td>
<td>5</td>
<td>3.00</td>
<td>3.50</td>
<td>3.00</td>
<td>120</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>3</td>
<td>5</td>
<td>2.80</td>
<td>3.00</td>
<td>2.80</td>
<td>90</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>1</td>
<td>3</td>
<td>3.30</td>
<td></td>
<td>3.30</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>I</td>
<td>1</td>
<td>3</td>
<td>2.30</td>
<td>2.30</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I</td>
<td>2</td>
<td>3.60</td>
<td>3.91</td>
<td>3.60</td>
<td>45</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I</td>
<td>2</td>
<td>3.70</td>
<td></td>
<td>3.70</td>
<td>90</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I</td>
<td>1</td>
<td>2</td>
<td>3.50</td>
<td>3.50</td>
<td>3.50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I</td>
<td>4</td>
<td>5</td>
<td>3.58</td>
<td>3.58</td>
<td>200</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I</td>
<td>4</td>
<td>3</td>
<td>2.80</td>
<td>2.80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I</td>
<td>5</td>
<td>2</td>
<td>3.50</td>
<td>3.50</td>
<td>3.50</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I</td>
<td>4</td>
<td>2</td>
<td>3.47</td>
<td>3.91</td>
<td>3.47</td>
<td>45</td>
<td>1.5</td>
</tr>
</tbody>
</table>
Figure 17: Survey Data Set (Part 2): Questionnaire Results

<table>
<thead>
<tr>
<th>Class Section</th>
<th>Program Version</th>
<th>#1 Time</th>
<th>#1 Answer</th>
<th>#2 Time</th>
<th>#2 Answer</th>
<th>#3 Time</th>
<th>#3 Answer</th>
<th>#4 Time</th>
<th>#4 Answer</th>
<th>#5 Time</th>
<th>#5 Answer</th>
<th>#6 Time</th>
<th>#6 Answer</th>
<th>Total Time</th>
<th>Total Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>4</td>
<td></td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>4</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>4</td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>9</td>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>9</td>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>3</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>4</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>4</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>7</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>8</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>8</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>8</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>8</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>7</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>6</td>
<td></td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

117