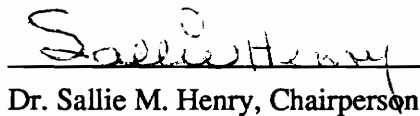
# Comparison of an Object-Oriented Programming Language to a Procedural Programming Language for Effectiveness in Program Maintenance

by

Matthew Cameron Humphrey

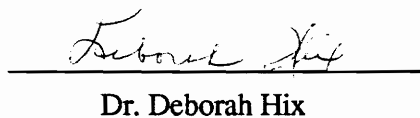
Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master of Science
in
Computer Science and Applications

APPROVED:

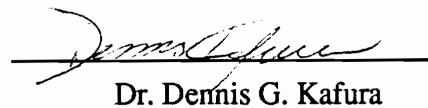


Dr. Sallie M. Henry, Chairperson

Dr. H. Rex Hartson



Dr. Deborah Hix



Dr. Dennis G. Kafura

Blacksburg, Virginia

LD
5655
V855
1988
H956
c.2

# Comparison of an Object-Oriented Programming Language to a Procedural Programming Language for Effectiveness in Program Maintenance

by

Matthew Cameron Humphrey

Dr. Sallie M. Henry, Chairperson
Computer Science and Applications

(ABSTRACT)

New software tools and methodologies make claims that managers often believe intuitively without evidence. Many unsupported claims have been made about object-oriented programming. However, without rigorous scientific evidence, it is impossible to accept these claims as valid. Although experimentation has been done in the past, most of the research is very recent and the most relevant research has serious drawbacks. This study attempts to empirically verify the claim that object-oriented languages produce programs that are more easily maintained than those programmed with procedural langauges. Measurements of subjects performing maintenance tasks onto two identical programs, one object-oriented and the other procedure-oriented show the object-oriented version to be more maintainable.

# Acknowledgements

I would like to thank my advisor, Dr. Sallie Henry, for her continued support throughout this research. Her assistance and insistence through the process of instigating a new course, in arranging for new software, in teaching "Object-Oriented Software Engineering" and in relentlessly collecting every scrap of data have been immensely appreciated. I would also like to thank my other committee members Dr. Rex Hartson and Dr. Dennis Kafura. A very important appreciation goes to Dr. Debby Hix who repeatedly pointed out the proper methods for experimentation and provided the guidelines for analysis. A heartfelt thanks to Dan Mercantile, Bob Koons and Colby Perkins of the statistics consulting lab. Their help has been invaluable in the analysis of the data.

A special thanks goes out to Cal Selig for his assistance with various computers, machines and data formatting and formatting guide for the thesis. I also want to thank Joy Davis, Bryan Chappell and Mary Kafura for various assistance they have given. I am also grateful to Robert and Beverly Williges for their professional advice and experience on this research.

I especially want to thank some very close friends and family for their emotional support: Scott Gosik, James Dunson, Denise Rich, Rose D., Andy Shoemaker, John Hodges, Geoff Knobl, Tessa Dalton, Doris, Elbert, Greg, Bert and many others. Words won't tell them how important they are. And lastly, my great-grandmother Mary Young Humphrey, who taught me how to quilt, without saying a word.

# Table of Contents

**Chapter 1 Introduction to Software Engineering Experimentation** 1
Introduction . . . . . . . . . . 1
Previous Studies . . . . . . . . . . 6
Conclusion . . . . . . . . . . 9

**Chapter 2 Maintenance and Enhancement** . . . . . . . . . . 10
Introduction . . . . . . . . . . 10
Object-Oriented vs. Procedural Development . . . . . . . . . . 11
Laboratory Experiment vs. Real World . . . . . . . . . . 14
Conclusion . . . . . . . . . . 14

**Chapter 3 Experimental Methods** . . . . . . . . . . 15
Introduction . . . . . . . . . . 15
Procedure . . . . . . . . . . 16
Subjects . . . . . . . . . . 19
Tasks . . . . . . . . . . 18
Materials . . . . . . . . . . 22
Conclusion . . . . . . . . . . 22

**Chapter 4 Data Collection** . . . . . . . . . . 25
Introduction . . . . . . . . . . 25
Student Data . . . . . . . . . . 26

Raw Student Data . . . . . . . . . . 29
Task Data . . . . . . . . . . 31
Conclusion . . . . . . . . . . 39

**Chapter 5 Results** . . . . . . . . . . **40**
Introduction . . . . . . . . . . 40
Significant Values for Student Data . . . . . . . . . . 40
Significant Values for Task Data . . . . . . . . . . 41
Faults in the Data . . . . . . . . . . 45
Conclusion . . . . . . . . . . 45

**Chapter 6 Conclusions** . . . . . . . . . . **46**
Introduction . . . . . . . . . . 46
Task Fault in the Experiment . . . . . . . . . . 52
Supporting the Hypothesis . . . . . . . . . . 56
Future Work . . . . . . . . . . 58
Conclusion . . . . . . . . . . 59

**Bibliography** . . . . . . . . . . **60**

**Appendix A. Rules of the Experiment for Subjects** . . . . . . . . . . **67**
Rules for Doing Assignments Handout . . . . . . . . . . 68
Task Instructions Handout . . . . . . . . . . 69
Task Details Handout . . . . . . . . . . 71

**Appendix B. Raw Data** . . . . . . . . . . **72**
Student Background Data . . . . . . . . . . 73
Task Data . . . . . . . . . . 74

**Appendix C. Questionnaires** . . . . . . . . . . **86**
Subject Background Questionnaire . . . . . . . . . . 87
Task Questionnaire . . . . . . . . . . 90
Post Class Questionnaire . . . . . . . . . . 92

**Appendix D. Task Specifications** ........................ **93**
Warm-up Task Specification ........................ 94
Task 2 Request ........................ 95
Task 3 Request ........................ 96
Task 4 Request ........................ 97

**Appendix E. Original Program Specifications** ........................ **98**

**Vita** ........................ **105**

# List of Tables


| | | |
|---|---|---|
| Table 1. | Order for Implementing Tasks | 23 |
| Table 2. | Summary of Student Data Dependent Variables | 27 |
| Table 3. | Means of Student Data | 30 |
| Table 4. | Task Data Dependent Variables | 32 |
| Table 5. | Raw Task Data Averaged Between Groups | 36 |
| Table 6. | Raw Task Data, Averaged Between Languages | 37 |
| Table 7. | Raw Task Data, Averaged Between Tasks | 38 |
| Table 8. | ANOVA of Student Data over Groups | 42 |
| Table 9. | Results of ANOVA on Task Data Variables | 43 |
| Table 10. | ANOVA on Task Data Variables Omitting Task 2 | 44 |
| Table 11. | Synopsis of Conclusions from the Task Data | 48 |
| Table 12. | Summary of Task Data Variables Omitting Task 2 | 53 |

# Chapter 1 Introduction to Software Engineering Experimentation

## Introduction

New software tools and methodologies make claims that managers often want to hear. "Language X cuts design time" or "This Computer Aided Software Engineering package improves maintainability". Most professionals recognize hype when they see it and treat it accordingly. Many managers and software engineers have only an intuitive feeling for the accuracy of these claims because there is no hard scientific evidence, only "warm fuzzy feelings". Intuitive feelings have little place in the scientific discipline of Software Engineering.

This experiment empirically tested some of the claims that have been made about object-oriented programming languages. Subjects completed simulated maintenance requests on two programs, one written in an object-oriented language and the other written in a procedure-oriented language. The purpose was to determine if the implementation language helped or hindered the enhancement task. By using a scientific study to answer questions about program maintenance, this experiment attempts to broaden the scope of knowledge in Software Engineering.

Software Engineering is "the technical discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost

estimates" [FAIR85]. Software products are not only the delivered code, but also any documentation, design notes, user manuals and personnel training that constitute the deliverables. To accomplish these goals, various methodologies have been developed to control the software development process.

Many organizations publish their own in-house software development paradigms which they use for all code developed internally [STEW81] [NAKY78] [SYNJ78] [DAVC76] [KRAK78]. There are also plenty of partial methodologies and software development techniques that apply to particular types of software, such as concurrent or distributed programs, mathematical programs or user interface programs [ALFM77] [MATY78]. Most methodologies propose to make software easier to write, maintain, modify and understand. They do this by partitioning the problem into smaller pieces that are easier to manipulate. The differences among the methodologies is how that decomposition is achieved. Instead of presenting all of these, only the three most prevalent program design techniques are presented here. The techniques are Structured Design, Jackson's Structured Programming and Object Oriented Design. Notice that these are not methodologies, but just techniques for developing programs. They are compatible with other design techniques, such as top-down design, bottom-up design, functional decomposition, stepwise refinement, iterative development, information hiding and others [ROSD77] [PARD79] [FAIR85] [SWAG78] [BROF82] [GANC77] [YOUE82].

Structured design divides a system into modules such that each module has a high binding strength while the coupling dependencies between modules is low [STEW74] [STEW81] [STAJ76] [PRIJ82] [FAIR85] [SWAG82] [GANC77] [YOUE82]. Structured design is often used directly with top-down decomposition and stepwise refinement. "The benefits of structured design result from the independence of the modules" [STEW81]. Typically, modules are defined that have the highest binding possible first, and then the resulting system structure is arranged to minimize the coupling. For most modern programming languages, a module is synonymous with procedure or function. With object-oriented programming, it means "method", an operation on an object. Binding is the relationship among the items within a module. Coupling is the relationship among modules in a system.

Jackson Structured Programming as a design technique maps the problem structure into a program structure that solves the problem [JACM75]. It works best with systems that have definite structure to the input and output data, as is typically found in data processing

facilities [FAIR85]. The designer describes the structure of the input data in terms of abstract structures, records and fields. The output data, usually a report, is described using the same hierarchical data constructs. The program transforms the input structure into the output structure. "The greatest strength of the Jackson method is the almost mechanical process of creating a program structure from the data structures" [FAIR85]. However, the method is less useful when there is a "structure clash", which occurs when the input is very different from the output in structure or information form. Jackson Structured Programming also has problems when the system requires to be able to "look-ahead" at the input data. There is no way to represent look-ahead in the input structure.

Object-oriented design is a new technique that uses the good aspects of top-down design and abstract data types combined with the modularization and separation of structured design. In object-oriented design, "the decomposition of a system is based on the concept of an object. An object is an entity whose behavior is characterized by the actions that it suffers and that it requires of other objects." [BOOG86]. Object-oriented design has several definable characteristics. Object-oriented programming directly supports these characteristics. The four attributes are Encapsulation, Messaging, Inheritance and Polymorphism.

Encapsulation forces objects to be visible from the outside only. They are encapsulated so that only their interfaces show. The interface of an object is the names and formats of the messages to which it can respond. Design information concerning the structure of the object is hidden within the object. Note that while encapsulation is usually considered an implementation detail by forbidding programmers from using private data (as in Pascal) [JACJ87], it is also a design detail since it allows the interface to an object to be described and used without knowledge of the inner workings. It can also be used to constrain programmers by using the interface as the detailed design, requiring the programmers to implement the necessary methods, but with the freedom to implement any way they choose, so long as the interface remains the same [BROF75] [RENT82] [BOOG86].

Messages accomplish the processing in an object-oriented system. In a purely object-oriented system (such as SmallTalk), all processing is accomplished via messages. In hybrid systems, like 'Objective-C', messaging exists simultaneously with procedure calls to accomplish processing. In slightly object-oriented systems, like Ada, Pascal and C, procedure calls substitute for actual messages. Messages break dependencies inherent in the "caller-callee" model because " a message is a request of what the sender wants with no hint

or concern as to what the receiver should do to accommodate the sender's wishes" [RENT82]. The receiver and the sender are separate and distinct units. No assumptions are made by the sender as to "how" nor any assumptions by the receiver as to "why". This enforces low coupling between modules [RENT82] [BOOG86].

Inheritance is the collective property of sharing and adaptation. "Sharing makes for a usable system by facilitating factoring, the property of one thing being in only one place. Successful factoring produces brevity, clarity, modularity, concinnity and synchronicity..." [RENT82]. Additionally, adaptation allows individual objects to be different. One object may share (inherit) the properties of another, then add or delete properties that are exclusive to itself. For example, a taxi is like a car, except that a taxi has a built-in driver and a meter. Both are still have doors and can go places, but they are distinct. Inheritance yields the most capability for code reusability, understandability and simplicity [RENT82].

Polymorphism is useful, but not a necessary characteristic of object-oriented design. It refers first to the separation of objects and variables. Variables are names used in programs to denote particular objects at execution time. The type of a data item is associated with the object that is the data item, rather than with the variable. The variable is merely a handle for the object. This allows for code that functions regardless of the type of object on which it is operating. For example, Stacks of objects or Lists of objects do not know what kind of objects are in the list, but it is not necessary for them to know.

Polymorphism has its disadvantages as well. Gannon showed that languages that allowed dynamic typing retained errors longer than those languages that did not [GANJ77]. This is because the programmer is required to remember the types of the objects. The most logical solution is to ban polymorphism, but the construct is useful. While the object-oriented language CLU alleviates the problem by implementing static typing, CLU also allows for a variable of type "any", which essentially produces a polymorphic variable [LISB81]. Since uncontrolled polymorphism exhibits longer lasting errors, it remains a useful, but optional characteristic of object-oriented programming.

The three required characteristics listed above are Encapsulation, which insures that only the interface of objects are visible; Messaging, which implements "call by desire" and provides low coupling and Inheritance, which allows both sharing of common code and adaptation of code to new, individual purposes [RENT82]. Polymorphism formally separates variables

from objects, but is optional and does not affect the design.

When designing a program using the object-oriented method, it is necessary to:

> "Identify the objects and their attributes
> Identify the operations suffered by and required of each object
> Establish the visibility of each object in relation to other objects
> Establish the interface of each object" [BOOG86]

This is similar to the basic analysis required by all other methodologies.
All methodologies have features that have long been recognized as good software engineering. Methodologies attempt to ease software design, construction, maintenance and comprehension by reducing complexity. Complexity is reduced by partitioning the problem into more manageable units. The smaller pieces are usually called modules and they enforce a hierarchy in which the most powerful, most abstract entities are on the top and the most concrete and primitive entities are on the bottom while the design decisions between the layers are well hidden.

While this is true of all methodologies, of particular interest is the object-oriented methodology. Object-oriented methodologies and object-oriented languages have put forth many unsupported claims. Object oriented approach cuts development time [BROF86], makes software "resist both accidental and malicious corruption attempts" [BOOG86], is more maintainable [BOOG86] [MEYB81], more understandable [BOOG86], has greater clarity of expression [BROF86], supports the buy vs. build software trend [BROF86][COXB86], is easier to make enhancements to [COXB84] [COXB86] [RENT82], better prototyping and iterative development [BASV75] [COXB84], reduced value-type errors because of uniformity of objects [MACB82] [COXB84] [COXB86] [RENT82].

While these claims have a qualitative "rightness" there is little supporting quantitative data. Boehm-Davis claims that object-oriented design are the hardest to modify, but a procedural language was used in the experiment [BOED86]. Gannon claims that dynamically typed operands (polymorphism) results in more errors, but in that experiment, programmers were required to keep track of the structure of the data themselves which violates the principle of information hiding [GANJ77]. The programmer should not have to care how the object is represented. Holt found that object-oriented programs are most difficult for subjects to recognize and understand, but again a procedural language was used with an object-oriented

design [HOLR87].

There are other problems with experimentation in software engineering. Many experiments are conducted on trivial programs which are only a dozen statements long [GANJ77]. Other experiments using student subjects made unreasonable conclusions about professional programmers.

This experiment justifies the claim that systems developed with object-oriented languages are more maintainable than those developed with procedural languages. Subjects determine the maintainability of two languages by performing maintenance tasks on two identical large programs, one written in each language. Maintenance times, error counts, change counts and programmer's impressions are collected. The analysis of the data shows that systems using object-oriented languages are indeed more maintainable than those built with procedural languages.

However, this conclusion is only one aspect of the many sides to software engineering. The goal of software engineering is produce better software systems. It is only possible to test this goal by controlled experiment and analysis. This experiment is another piece in the software engineering "mosaic". "The primary goals of software engineering are to improve the quality of software products and to increase the productivity and job satisfaction of software engineers." [FAIR85]. By this software engineering strives to reduce software cost, increase reliability and increase robustness, among other things. The goal of this experiment is to expand the foundations of software engineering so that those who work with software can make intelligent choices when building and maintaining systems.

## Previous Studies

Only recently has there been much empirical experimentation in computer science [ATWM78] [BARM69] [BARM77] [BOIS74a ][BOIS74b] [CARJ70] [CARE77] [CURB79] [DUNH77] [ELSJ76] [GANJ76] [GANJ77] [GOUJ75] [KNUD71] [LEEJ78] [LOVT77] [LUCH74] [MYEG78] [SAAH77] [SCHN77] [SHNB76] [SHNB79] [YOUE74]. Within the curriculum of software engineering and software maintenance there is great difficulty in obtaining appropriate materials and subjects. The greatest difficulty is training the subjects in

the use of a methodology or a language or a technique. Teaching methodologies can take from months to years. These problems deter many researchers from doing thorough studies. However, two experiments conducted by Deborah Boehm-Davis and Robert Holt do present some preliminary background information in the field of software maintenance. Their studies are now summarized.

In the Boehm-Davis experiment, 18 professional programmers built programs using one of three different design methodologies: Jackson Structured Design, Object Oriented Design and Functional Decomposition Design. The programs were small, requiring a minimum of only 100-200 lines of code. The results collected showed that "the data did not provided any clear answers regarding their relationship to future maintainability" [BOED84]. However, it is still possible to suspect that object-oriented code is more maintainable, since "the complexity ratings again favor the Jackson and object-oriented methodologies as they show lower complexity ratings than the other solutions" [BOED84].

The study did show that "The complexity ratings again favor the Jackson and object-oriented methodologies, as they show lower complexity ratings than the other solutions. Low complexity ratings are important since they are argued to be the key to more maintainable code" [BOED84]. They also found that "There was no correlation between percent complete and years of programming experience, and the solutions generated by the experienced programmers were no more alike than the programs generated by the less experienced programmers" [BOED84]. This second finding has the greatest impact, since it supports the use of students in programming experiments as being relatively as useful as professional programmers. This helps alleviate the problem of lack of professional programmers on which to experiment.

In the end, this study provided some ideas, but no concrete information for maintainability. Additionally, there were some flaws with the study. The Jackson and Object-oriented methodologies did equally well, but "The programmers using the Jackson program design methodology had a great deal more experience in programming and with the design methodology, than did the programmers using either the object-oriented or functional decomposition approach". Therefore, the experiment was biased towards programmers who had used the Jackson method. In spite of the bias, though, they did only as well as the programmers who used the object-oriented methodology with little or no training. The large differences in the experience of the users of the different methodologies makes the results

unreliable.

Additionally, not all of the data was complete when it was collected. Incomplete solutions were subjectively judged as to the degree of completeness and given a percent rating accordingly. The programs were not of realistic size, reducing the effectiveness of the methodologies, which work best when the project cannot be easily handled by a single person. The definitions of the methodologies were very vague and do not seem to be supported by any external documentation. The study does not say if the programmers were provided with methodology definitions on how to effectively use the methodology.

Another experiment by Boehm-Davis and Holt [HOLR87] attempts to make statements about software maintenance based on the methodology, the experience of the programmer, the complexity of the task and the type of the program. Eighteen professional programmers and eighteen student programmers were asked to make simple or complex modifications to several of three different programs, each built according to three different methodologies. Each subject performed one task on each of the three different types of problems. The results indicate that functionally decomposed code is the easiest to modify and object-oriented code is the most difficult to modify. This is the exact opposite of what the researchers expected and what one intuitively expects since the functionally decomposed code had little structure or modularization while the object-oriented code was the most modular.

This study also has several problems. The size of the programs being modified was not indicated, but a minimum of only 100-200 lines of code is likely, since these problems are the same as the programs that appear in the 1984 Boehm-Davis experiment. All of the programs were written in Pascal, which is not an object-oriented language. To use Pascal as an object-oriented language requires adding a great deal of complexity.

There were three subjects for each of the possible treatment levels of the experiment, yielding a very small sample size. Additionally, direct comparisons of subjects with other subjects, called a "between subjects" experiment, was used for several parts of the experiment, which is highly unreliable when dealing with small numbers of programmers [BROR80].

The methodologies tested in the experiment were not well defined. Additionally, the subjects were not necessarily practiced in the various methodologies. How is it possible to test the effect of methodology when the subjects are not familiar with it?

The experiment does not say if the modifications were presented as "change variable X in line Y", which is an implementation modification or if the change is to the specification: "Make the program do this now, change how it does this now." A specification change would be the same for a program regardless of the methodology while an implementation change depends only on the code.

Two different kinds of changes were used, which is a good way to see how the complexity of the change affects the performance. A simple change is defined as being a change in only one location and a complex change is defined as a change in many locations. However, unless the change is defined as "change this line" how is it possible to guarantee what a simple change is. Modification tasks written from the implementation point of view are not indicative of the maintenance task. While it indicates that changes localized in one area of the code require less time than those modifications requiring changes in many locations, those happen to be the definitions of simple and complex modifications.

The evidence presented in the first experiment is not statistically significant to be certain of the results. The bias towards Jackson structured programming also interferes with the results. Only the second experiment can make any significant claims about maintainability of object-oriented code. But for that experiment, there are a large number of errors in the construction of the experiment that weaken the results. The small sample size, the vagueness of the methodology definitions and the problems of how the modification tasks are presented (implementation or specification) indicate that a more rigorous experiment is needed to verify the results.

## Conclusion

Although experimentation has been done in the past, most of the research is very recent and the most relevant research has serious drawbacks. The experiment presented in this thesis attempts to solve problems had by other experiments to provide more relevant data to the research community.

# Chapter 2 Maintenance and Enhancement

## Introduction

Software typically passes through the life cycle stages of requirements, design, implementation, testing and maintenance. The experiment described in this thesis concentrated on the maintenance portion of the software life cycle. Maintenance can be divided into three sub-activities: "corrective maintenance (performed in response to the assessment of failures); adaptive maintenance (performed in anticipation of change within the data or processing environment); and perfective maintenance (performed to eliminate inefficiencies and enhance performance or improve maintainability)". [LIES78]. This research concentrates only on perfective maintenance which is also called "enhancement".

Improving the maintenance phase of the software life cycle promises the best reduction of software cost. Many researchers have concluded that most of the cost of software is spent on maintenance, between 50% and 75% [LEHM80] [FAIR85]. Lehman has estimated that the United States, government and commercial institutions, collectively spent more than $50 billion to $100 billion on software in 1977, or more than $25 billion to $75 billion on maintenance [LEHM80].

Software enhancement is the largest portion of maintenance. Fairly estimates that 60% of all maintenance money is spent on perfective maintenance [FAIR85]. That's equivalent to 42% of the total software cost being spent on enhancements after the product is delivered. A small

improvement in this area of maintenance yields large returns.

Besides cost effectiveness, studying maintenance reduces the effects of requirements, design and testing from the grand scale to a small scale. There are many methodologies for the early part of the life cycle, and studying a small issue in the design area brings with it the problems of comprehending requirements, understanding the whole system, building the whole system and testing methodologies. While the enhancement task must nevertheless be understood, coded and tested, analyzing the single task is more accurate than analyzing the entire system.

Additionally, experimenting on a large real world system yields more realistic results. Many studies are flawed by having subjects write very short programs to test language features [GANJ77]. The simple programs do not adequately represent the real world design and coding effort. "...writing a large system is not just a matter of scaling up the manpower required for a small system.." [BROR80]. Studying maintenance of realistically sized programs reduces the artificiality of the experiment.

## Object-Oriented vs. Procedural Development

In this experiment, subjects performed maintenance tasks on two functionally identical programs. One program was written in 'C' and the other was written in 'Objective-C'. The purpose of using 'C' and 'Objective-C was to measure the effects of maintenance on a system developed using an object-oriented language as compared to a system using a "classical" procedural language. These two languages minimize the effects of using different languages. To begin, clear definitions of object-oriented and procedural must be given.

An object-oriented program, in the roughest sense, uses data objects rather than control flow to decompose the system into manageable pieces. However, this definition is too vague and really refers more to the design of the program than the implementation. These issues of program design are not of interest; only the language of the implementation is being considered.

For this experiment, a language is object-oriented if it provides the following facilities: encapsulation, messages and inheritance. These facilities must be intrinsic to the language. Smalltalk has the smoothest instance of the object-oriented paradigm, but 'Objective-C' was

chosen as the object-oriented language for the experiment. 'Objective-C' has many other features, such as polymorphism, but they were not required.

Many have claimed Ada is object-oriented, but it was rejected from use in this experiment because Ada is not a true object-oriented language. The definition of object-oriented put forth previously in this paper describes why Ada is not object-oriented. To be object-oriented, a language must support the following features: encapsulation, messaging, inheritance and optionally, polymorphism. Ada does not directly support all of these, so it is not object-oriented. Each characteristic is discussed in turn.

Ada provides encapsulation of data types. Packages, with private interfaces allow Ada to create data types that are entirely hidden within modules. Only the interface portion is visible to other packages that wish to use the data types. Additionally, the project library insures consistency of type checking across procedure calls without allowing private data to "leak out".

Ada provides no messaging facility. Ada uses a purely procedure call paradigm in which the name of the procedure exactly identifies the subroutine to be called. The connection is static. Overload operators can be used, in a limited sense, to allow the procedure call to vary with the type of the parameter, but this is equivalent to the use of type-switch tables. This is the same problem that Jacky's object-oriented Pascal methodology. It serves only to make the code more complex.

Ada provides little capability for inheritance. The key to proper sharing of resources rises from the fact that when a new class inherits methods from a pre-existing class, those methods need not be re-implemented. The Class hierarchy is automatically traversed, finding the proper implementation of the message. In Ada, the new class must either re-implement the method or explicitly invoke the superclass. Additionally, the new class must have a component that IS the superclass rather than allowing the new class to actually be an adaptation of the superclass. Therefore, Ada has no real provision for inheritance [RENT82].

Ada is strongly typed and even makes use of the project library to insure type correctness between compilation units. While it is possible to achieve a degree of polymorphism by using generic units, this has the same problem as using overload operators for messages.

However, polymorphism is not a necessary attribute of an object-oriented language, and nothing is lost by not having it.

In general, a language must support encapsulation, messaging and inheritance to be object-oriented. Since any object-oriented program can be run on an ordinary machine, it is possible to write object-oriented programs in assembly language. But that certainly does not imply that assembly language is object-oriented. It is for this reason that a language must actively support the object-oriented paradigm to be considered object-oriented. Unless the language actively supports encapsulation, messaging and inheritance, it cannot be considered object-oriented. Ada does not support messaging or inheritance; it is not object-oriented.

Having defined object-oriented languages and why 'Objective-C' was chosen over Ada for this experiment, a definition of a "procedural" language must be given to explain why 'C' was chosen over Pascal. A procedural language is one in which procedures decompose the system into manageable pieces. Procedures abstract over control flow, rather than data or data flow. This definition, however, is still a bit weak. The procedural language used in this experiment must also have abstract data typing facilities. Pascal, of course, is an ideal choice since it is well known by the subjects, but 'C' has been chosen since it limits the differences between the object-oriented language and the procedural language just to that which makes 'C' into 'Objective-C'. It was also chosen so the subjects would have no bias from having had previous experience with one of the languages.

From the definitions of procedural and object-oriented languages arises the explanation why this experiment examines object-oriented languages at all. Object-oriented methodologies and languages have received much attention in the past few years as being the new development standard, just like "structured programming" was in the 70's.

There are claims that object-oriented languages solve some of the problems now plaguing software engineering [COXB86a] [COXB86b] [COXB84] [JACJ87] [MACB82] [RENT82] [BOOG86] [BROF86] [GOLA83] [ROMH85] [SHAM84] [LISB79]. Also there are expectations of many good things to come from objects. But very few of these claims are rigorously supported. There is almost no empirical evidence, except a handful of research which is inconclusive [BOED84] [BOED86] [HOLR87]. The purpose of this controlled experiment is to rigorously support the claim that object-oriented languages produce systems that are more maintainable than those from procedural languages.

## Laboratory Experiment vs. Real World

A controlled experiment can be used to support the claim that object-oriented languages cost less to maintain than procedural languages. A scientific approach followed by suitable statistical analysis will yield meaningful results. To do so, the dependent and independent variables must be identified, the independent variables controlled and the dependent variables measured and analyzed throughly. Only through accuracy can reliable results be guaranteed.

The real world environment lacks the controls to produce suitable data. Programmers have vastly different demands and schedules, especially at different sites and usually they cannot be spared for research work. Computers that present a uniform environment to all subjects are difficult to obtain. However, the real world has the advantage that the results have direct meaning for the subjects involved.

Artificiality is the problem of a controlled experiment, though. The results may not be extensible outside of the laboratory. However, any truths that are found in the data apply to all programmers everywhere. A goal of controlled experimentation is to control all of the unknowns, so they produce no bias. When these real world unknowns have a significant effect on the data is when the data are unreliable.

## Conclusion

The maintenance and enhancement portions of the life cycle yield the greatest cost returns for improvement. Object-oriented languages offer promise in improving the maintenance task. The claims that programs built with object-oriented languages are more maintainable than those built with procedural languages can be verified through a controlled experiment. Additionally, this experiment must be careful to remain extensible to the real world while avoiding being artificially contrived.

# Chapter 3 Experimental Methods

## Introduction

A goal of this research was to verify the claims that object-oriented design and implementation yield more maintainable systems. This goal was achieved in a controlled experiment where subjects performed enhancement maintenance on two functionally identical programs, one designed with structured design techniques using a procedural language and the other designed with object-oriented design techniques using an object-oriented language. Measuring various dependent variables when the subjects performed the task gave insight into the usefulness of object-oriented programming over structured procedural programming.

The hypothesis of this study is that systems designed and implemented in an object-oriented manner are easier to maintain than those designed and implemented using structured design techniques. Easier to maintain in this context means the programmers take less time to perform a maintenance task or that the task required fewer changes to the code. It also means that programmers perceived the change as conceptually easier or that they encountered fewer errors during the maintenance task. Maintenance is defined in terms of the variables used to measure the subjects' performance.

This experiment was arranged as a within subjects test with three independent variables. The variables were: the programming language, the subject group and the task. The programming language had two levels of treatment: procedural or object-oriented. The

system built using structured programming was written in 'C' and the system built with object-oriented techniques was written in 'Objective-C'.

The subject group also had two levels of treatment: Group A and Group B. Every subject was required to perform a modification task to both programs. Group A subjects modified the 'C' program and then modified the 'Objective-C' program before proceeding to the next task. Group B subjects did the reverse: they modified the 'Objective-C' program first and then modified the 'C' program. This counterbalancing attempted to eliminate any effect of using one language for a task before the using the other for the task.

The subjects performed four tasks. Each task was performed once on each of the two programs. All subjects performed the tasks in the same order. The tasks were all of a very similar nature and were not selected to exhibit any particular attribute. The first task was designed as a warm-up exercise and was not included in the data analysis.

## Procedure

This study was presented through a college senior-level course in software engineering entitled "Object-Oriented Software Engineering", which used the course "Introduction to Software Engineering" as its prerequisite. The object-oriented software engineering course was divided into two phases of eleven weeks each, a teaching phase and an experiment phase, such that a phase was one academic quarter. The first phase, the teaching phase, involved teaching the students software engineering techniques and the languages to be used in the study. No experimental data were collected during this segment. The second phase was the actual experiment, in which the students of the course were the subjects and they performed the tasks and data were collected. The students were enrolled for the course for both quarters.

The teaching phase encompassed three segments: software engineering, structured programming and object-oriented programming. During the first segment, general principles of software engineering applicable to all methodologies were presented, including motivation for software engineering and the need for control in development studies and experiments. These topics were presented without a bias for any programming language, methodology or

computer system.

The next segment followed with a review of structured design techniques, including general design, detailed design, structure charts, coupling and cohesion. The students were already familiar with these techniques, since the prerequisite was assumed to teach them. During this time the students were also taught the 'C' language and familiarized themselves with the VMS operating system, on which all their assignments and the experiment were given. Their programming assignments for this segment involved designing, coding and integrating their code with other students' code.

The last segment involved teaching object-oriented design and programming. They were taught the necessity of encapsulation, messaging and inheritance for accomplishing the design and implementation task. During this time also, students were taught the 'Objective-C' language which was available on the same machine as the 'C' language. The programming assignments again included design, coding and integrating new code with other student's code.

Students were also asked to each make a twenty minute presentation on topics of interest in software engineering. They were allowed to select their own topics with the approval of the instructor. Topics ranged from how software engineering techniques apply to optical disk drives to the psychology of validating methodologies with experimentation. Students were asked questions by other students and by the instructor during their presentations.

The final exam for the teaching phase involved interpreting the meaning of software engineering techniques and the use of object-oriented terminology. Students were then ranked on the basis of their scores on the programming assignments, the final exam and the presentations. The bottom two scores were selected as graders and the top two scores as pretesters for the experimental tasks. The remaining students were then randomly selected for groups A and B. This last event marked the end of the eleven week teaching phase.

The eleven weeks of the experiment phase followed. For the start of the experiment students were asked to complete a questionnaire on their programming experience. This questionnaire assessed the abilities of the subjects. This background questionnaire appears in Appendix C. They were then given a packet (Appendix A) containing information about the rules of

participation in the experiment and the two programs to be maintained. The rules of the experiment were also explained in detail in class, emphasizing that the student's performance in the experiment in no way would affect their grade. Accuracy in collecting data was stressed as more important than "good" data or "bad" data.

After the subjects completed the background questionnaire and read and understood the rules for the experiment, the first task was distributed. They were told that each task had to be completed before they would receive the following task. Additionally, they were not told that the first task 1 was a warm-up exercise, not intended to be used in the study. They were then allowed to work on the task out of class during the following week. While no deadline was assigned to any of the tasks, the subjects were told that it was imperative that they complete all of the tasks and that only exceeding the eleven week limit would endanger their grade.

Classes continued to be held once per week for two weeks, but no new material was presented. Classes afforded the students the opportunity to ask questions about the tasks or the language. Later class meetings were cancelled in favor of allowing students to simply do the tasks and questions were relegated to ample office hours scheduled for the same time slot.

No deadlines were assigned to the tasks, to prevent subjects from being pressured. However, it was made clear that every task had to be completed for credit. Some students, of course, slacked off in the first few weeks, and in a flurry of activity, completed the tasks by the only real deadline: the end of the eleven week period. Only two students were not done by that deadline. They were penalized only for turning in the tasks past the deadline. Additionally, one of the students performed the tasks out of order and therefore that student's data do not appear in the analysis.

After the subjects completed all tasks, they were asked to fill out a post-experiment questionnaire that assessed their feelings of their involvement in the experiment. They were asked to rate the productiveness of their experiences on an anchored 1 through 9 scale and then to give short descriptions of their involvement and opinion of the experiment. The post-experiment questionnaire appears in Appendix C.

## Subjects

There were 25 students enrolled in the "Object-Oriented Software Engineering" course. Two students were selected as "graders" to collect and record data from the subjects. Two other students were selected as pretesters to make sure the tasks were of reasonable complexity, had no undue complications and were of comparable magnitude. Group A had ten subjects and Group B had eleven subjects. The subject eliminated from the experiment was from Group B.

Students were used in this experiment primarily due to their availability over the twenty-two week period. The efficacy of use of students as subjects is supported for within-subjects experiments by Brooks [BROR80] and supported with empirical evidence by Boehm-Davis [BOED84]. Each student completed a background questionnaire to indicate his or her previous experience in programming and software engineering. The questionnaire appears in Appendix C.

The background questionnaire measured the students' overall Quality Credit Average (Virginia Tech's equivalent to Grade Point Average), their Computer Science Q.C.A., the number of month's experience programming in 'C', Pascal, Objective-C and SmallTalk, the number of month's experience in integrating code with other programmers code and the number of month's experience in testing software.

Additionally, all of the subjects were at approximately the same academic level. Most were graduating seniors who had been co-op students. They were all brought along together though the course "Object-Oriented Software Engineering" and had all previously completed the required course "Introduction to Software Engineering".

## Tasks

There were four modification tasks, which appear in Appendix D. A modification task was a simulated request from users to make a functional change to the system. The change was specified in terms of observable system behavior and not in terms of the implementation

code. This was to simulate a real user's request for change and isolated the task specification from the implementation language. The first task was given as a warm-up exercise and was not included in the data analysis. The remaining three tasks generated the actual data.

Both systems to which the changes were made were coded from identical specifications and user interface information. They were functionally identical so that when running it was impossible to distinguish the programs or to identify the implementation language. This was the criteria for both systems to be considered identical. The project specifications for the systems are given in Appendix E. The specifications were independent of the implementation language.

In general, the purpose of the programs was to be a sort of "laundry-list" handler. The system was not graphical, but used cursor control to maintain a formatted screen that looks like a scrap of paper with ten slots for notes. A note in the list was either a line of text or the name of a sub-list or the name of an account ledger. The line of text was simply a string and a sub-list was defined recursively through the definition of a list. An account ledger was a different data item. An account ledger was a list of purchase items and annotations. A purchase item was either a direct purchase, with a name, a category and a dollar value or a purchase item was a sub-ledger, which yields a name and a dollar value. An annotation was a line of text with no numeric content. The user was allowed to view and edit the lists and ledgers, descending as many levels as desired.

This program was chosen as the basis for the experiment because it seemed to encompass a broad range of programming techniques. It had a formatted user interface, used complex and nested data structures, was interactive, had varying control constructs, used a sizable number of procedures, functions and modules. The program was intended to be representative of typical systems.

Neither 'C' nor Objective-C had any built-in facilities that made building this program easier. Both systems were programmed from scratch, starting with the design specifications. Further details on the system are given further below in the "materials" section. As a note, both systems used 15 files (modules) each, comprising a total of approximately 4000 lines of code for each system. Lines of code here also included internal documentation.

Each task consisted of two parts. For group A subjects, the first part was to perform the task onto the 'C' system and the second part was to perform the task onto the Objective-C system. For group B subjects, the first part was to perform the task onto the Objective-C system and the second part was to perform the task onto the 'C' system. Therefore, subjects actually performed each task twice, once onto both of the systems. Performing each part of the task had to be completed before proceeding to the next task. Subjects were only allowed to work on one part of a task at a time (e.g. subjects were asked not to think about how to code the Objective-C portion of task two before completing the 'C' portion). This attempts to prevent information exchange between tasks. Additionally, subjects were not provided with the specification for a new task until both parts of the preceding task were completed. Table 1 shows the order for implementing the tasks.

Each task required that the modifications be made to an original copy of the system, as if the modification request were received with no knowledge of the other requests. This prevents the tasks from cumulatively interfering with each other. It also provides a basis of comparison for all tasks: the original copy. There is no control group for this experiment since an "optimal" or "ideal" implementation of the task does not exist. This is why the subjects' modifications are compared to the unaltered version. It is only possible to measure the difference between the original and the modified versions to determine the amount of work done.

Tasks were developed by having experienced computer programmers run the program and make comments about what new features would be handy or clever to add to the system. All tasks added new functionality to the system. Of the suggestions that were received from this informal study, four tasks were selected as "official" to be used in the experiment. These tasks were selected because they represented a broad range of programming constructs, and yet were all of the same level of difficulty. They were not chosen for any particular language features that they used. They were chosen because they seemed fairly independent of the programming languages.

A task was defined to be complete when it successfully ran with four special input data files, only one of which was available to the subject for testing. If the program did not generate a run time error, it was accepted as complete. If it did generate an error, the subject was asked to continue the task. Only two subjects on two different tasks submitted non-working

programs, which the subjects quickly repaired; all others were immediately accepted since they generated no errors.

## Materials

Subjects were given the following information:

A) Complete documented source code for the 'C' system,
B) Complete documented source code for the 'Objective-C' system,
C) The software specifications from which both systems were built,
D) Running copy of the original 'C' system,
E) Running copy of the original 'Objective-C' system,
F) One file of test data per task.

No copies of modified version (for which the provided test data would work) were provided. Likewise, design notes and external documentation and structure charts and object diagrams were not provided to subjects. These were restricted for notational reasons. This experiment is not designed to compare the differences in specific notations of design techniques. By providing only documented source code, there is an attempt to eliminate design notation as an independent variable.

## Conclusion

This experiment is designed to show that programs written with object-oriented languages are easier to maintain than systems written with procedural languages by controlling the three independent variables of language, subject group and task number and measuring the subjects by using the questionnaire that appears in Appendix C.

Additionally, the experiment used two phases, the first of which trained subjects in the techniques and languages, the second of which actually collected the data. The training phase taught the subjects general software engineering techniques, structured design techniques, the

**Table 1. Order for Implementing Tasks**

| Group A | Group B | |
|---|---|---|
| Task 1 C | Task 1 Obj-C | Warm-up Exercise |
| Task 1 Obj-C | Task 1 C | Warm-up Exercise |
| Task 2 C | Task 2 Obj-C | |
| Task 2 Obj-C | Task 2 C | |
| Task 3 C | Task 3 Obj-C | |
| Task 3 Obj-C | Task 3 C | |
| Task 4 C | Task 4 Obj-C | |
| Task 4 Obj-C | Task 4 C | |

'C' language, object-oriented design techniques and the Objective-C language. The experiment phase first collected background information on the subjects, then issued the warm-up task and then issued the three main experiment tasks and closed with a questionnaire assessing the subjects opinions of the experiment.

The subjects were students that had signed up for a course in object-oriented software engineering. They were told from the beginning both that the course taught object-oriented techniques and the 'C' language as well as the Objective-C language and that the practical portion of the course would be an experiment for collecting data. The subjects themselves were separated into two groups (A and B) by random assignment. Afterwards, background information was collected to make sure that there were no significant differences between the groups.

The tasks the subjects performed were simulated enhancement requests for an interactive list managing system. All of the tasks were the same for all subjects and each task was repeated on each of the two identical systems by all subjects. While the first task was a warm-up exercise and does not appear in the data analysis, the remaining three tasks, yielding a total of six parts, do form the real data for the experiment.

The arrangement of the procedure of the experiment, the selection and assignment of the subjects and the construction of the task and experiment materials leads to the proper design of the experiment. The data is collected from the questionnaires and analyzed for differences, which will support the hypothesis that object-oriented systems are more maintainable than structured programming systems.

# Chapter 4 Data Collection

## Introduction

This experiment collected two sets of data. The first set described the subjects and is used to show homogeniety among subjects and between groups. It is referred to as the "student" data. The second set is the actual experimental data. These data were generated by the questionnaires the subjects filled out for each task they performed. It is referred to as the "task" data. The student data is used to show that the experiment was free of bias in the subjects. The task data is used to prove claims about the abilities of the languages 'C' and Objective-C.

There are four independent variables

1) The student identifier (1 through 10)
2) The group to which the subject belonged (Group A or Group B)
3) The language used in performing a task (C or OBJC)
4) The task identifier. (2, 3 or 4)

These variables are denoted by SUBJECT, GROUP, LANGUAGE and TASK. The 10 subjects multiplied by the two groups multiplied by two languages and multiplied by three tasks, yields 120 data observations for the task data. The task data is partitioned by the four

main independent variables. The student background data is partitioned only by the student identifier SUBJECT and the subject group GROUP. Ten subjects multiplied by two groups yields 20 observations for the subject background data.

## Student Data

Two sets of data were collected on the subjects: background data and post-experiment impressions. The background data were collected to show that the two groups of students were similar and that the random assignment of students to groups produced a fair mixture. The post-experiment impressions were collected to allow students to express their opinions about the experiment and the course "Object-Oriented Software Engineering".

All background data was collected using a three page questionnaire that appears in Appendix C. Subjects were given one week to fill out the questionnaire and return the information. The questionnaires were checked for completeness and the subjects were asked to make sure that all questions were answered.

Students from the "Object-Oriented Software Engineering" course were randomly assigned to these to two groups, Group A and Group B. Random assignment was supposed to remove any bias due to differences in the subjects. However, to be confident that the groups were reasonably similar, background information was requested from each subject. A background questionnaire for the subjects collected values for thirteen dependent variables that measured the capabilities of the subjects to insure that the two groups of subjects are similar.

The following variables are from the background questionnaire, except for two subjective questions which are from the post-experiment questionnaire. The two subjective questions are SUBJTASK, how difficult the subject thought the tasks were in general, and SUBJQUES, how difficult the subject thought the questionnaires were. See table 2.

**Table 2. Summary of Student Data Dependent Variables**

| Variable | Synopsis | Pre or Post-experiment |
|---|---|---|
| QCA | Subject's overall QCA | pre-experiment |
| CSQCA | Computer Science QCA | pre-experiment |
| CURRIC | Subject's curriculum | pre-experiment |
| C | Months of 'C' experience | pre-experiment |
| PASCAL | Months of Pascal experience | pre-experiment |
| OBJC | Months of Objective-C experience | pre-experiment |
| SMALLT | Months of SmallTalk-80 experience | pre-experiment |
| INTEGR | Months experience integrating code | pre-experiment |
| TESTX | Months experience testing code | pre-experiment |
| LEVEL | Academic level | pre-experiment |
| COURSES | Number of Computer Science courses | pre-experiment |
| SUBJTASK | Task difficulty, subjective | post-experiment |
| SUBJQUES | Questionnaire difficulty, subjective | post-experiment |

The table 2 gives a brief overview of the definitions of the dependent variables for the student data. Under the "Variable" heading is the name of the variable. The synopsis describes briefly what the variable attempts to measure. The "Pre or Post-experiment" column shows if the values for the variable were measured before the experiment or after. The questionnaire used before the experiment is in Appendix C. The post-experiment questionnaire used after the experiment also appears in Appendix C.

Each variable is now explained in detail. A more accurate definition is given for each variable and examples are included where necessary. These definitions are the rigorous definitions of the dependent variables of the student data.

QCA stands for quality credit average. It is the overall grade point average of the student as appeared on his or her most recent grade card. It is on a 0.0 to 4.0 scale.

CSQCA stands for computer science quality credit average. This is the grade point average of the student computed using only computer science courses. It is on a 0.0 to 4.0 scale.

CURRIC stands for the curriculum in which the student is enrolled. Possible values are CS for computer science, EE for electrical engineering and CPE for computer engineering.

C stands for the number of months the subject has worked with the C language. These months are considered for all programming assignments, academic and commercial. Academic programming assignments are those undertaken for a class. Commercial or professional assignments are those for which the subject had been hired.

PASCAL stands for the number of months the subject has worked with the Pascal language. These months are considered for all programming assignments, academic and commercial.

OBJC stands for the number of months the subject has worked with the Objective-C language. These months are considered for all programming assignments, academic and commercial.

SMALLT stands for the number of months the subject has worked with SmallTalk-80

language. These months are considered for all programming assignments, academic and commercial.

INTEGR stands for the number of months the subject has spent integrating his code with someone elses, either at work or for an academic assignment.

TESTX stands for the number of months the subject has spent testing code either in an academic environment or in a professional environment.

LEVEL is the academic level of the subject, either sophmore, junior, senior or graduate student.

COURSES is the number of hours of computer science courses the subject has taken.

SUBJTASK is a subjective measure assessing how difficult the subject thought the tasks were. It is on an anchored scale from 1 to 9. Values were collected after the experiment.

SUBJQUES is a subjective measure assessing how difficult the subject thought the questionnaires were. It is on an anchored scale from 1 to 9. Values were collected after the experiment

## Raw Student Data

Table 3 gives the means and standard error values for the student data according to group. The data were averaged for each group and each variable. The name of the variable appears on the left followed by the mean for Group A and the mean for Group B.

**Table 3. Means of Student Data**

| Variable | Mean A | Err A | Mean B | Err B |
|---|---|---|---|---|
| QCA | 3.101 | 0.131 | 2.860 | 0.167 |
| CSQCA | 3.482 | 0.080 | 3.090 | 0.152 |
| CURRIC | | 0.000 | | 0.000 |
| MC | 4.700 | 1.184 | 5.000 | 1.085 |
| PASCAL | 27.800 | 3.169 | 33.200 | 2.308 |
| OBJC | 3.000 | 0.000 | 3.000 | 0.000 |
| SMALLT | 0.000 | 0.000 | 0.100 | 0.100 |
| INTEGR | 5.100 | 1.888 | 8.100 | 1.963 |
| TESTX | 49.200 | 7.203 | 50.900 | 6.967 |
| LEVEL | 3.800 | 0.133 | 3.900 | 0.100 |
| COURSES | 6.700 | 0.300 | 7.900 | 0.781 |
| SUBJTASK | 1.950 | 0.203 | 2.400 | 0.208 |
| SUBJQUES | 1.150 | 0.107 | 1.200 | 0.133 |

## Task Data

Two methods were used to collect the data associated with each task: questionnaires and an automatic data collection facility. While students worked on the task, they each filled out a questionnaire that recorded the amount of time they spent on the task as well as the number of errors they made. The questionnaire appears in Appendix C.

Once the subjects completed a task, they filled out the subjective portions of the questionnaire and turned in the completed forms. The computer then automatically tested their programs using a four sets of test data. Only programs that satisfied the complete test data were accepted as finished. For all the programs that passed the tests, the computer compared the subjects' source code to the original program and recorded the differences using the VMS "DIFFERENCE" facility. It also recorded the differences in sizes. After all of this was recorded in a file, the students continued to the next task.

Each task had two parts (one for each language) and there were three tasks yielding six task questionnaires per subject. Only tasks two, three and four are considered for this analysis, since the first task was a warm-up exercise. Since there are two groups of ten subjects each, there are 120 data observations. For the 120 observations, the questionnaires collected values for 21 dependent variables. Two of these variables are synthesized values, not directly measured. The remaining nineteen result from data that were collected automatically by the computer or that the subjects kept track of themselves, via questionnaires and time-use sheets. The synthesized variables are TOTERR, the total number of errors and TASKTIME, the total amount of time used to complete the task.

Table 4 which follows gives an overview of the dependent variables used in the task data. The "Variable" column lists the formal name of the variable. It is followed by a brief definition of the purpose of the variable. Finally, the variables that were collected by the computer are marked "Yes" in the last column. Those variables marked "No" were collected by subjects filling out questionnaires and "Synthesized" means automatically calculated from given information.

**Table 4. Task Data Dependent Variables**

| Variable | Synopsis | Automatically Collected |
|---|---|---|
| MODULES | Number of files changed | Yes |
| SECTIONS | Number of sections changed | Yes |
| LINES | Number of lines different | Yes |
| TOTLINES | Difference in file sizes | Yes |
| CERR | Number of failed compilations | No |
| TC | Number of compilation errors | No |
| LE | Number of linking errors | No |
| RE | Number of program crashes | No |
| LGE | Number of program logic errors | No |
| TOTERR | CERR+TC+LE+RE+LGE | Synthesized |
| STHIN | Thinking difficulty, subjective | No |
| SMOD | Modifying difficulty, subjective | No |
| STEST | Testing difficulty, subjective | No |
| SALL | Task difficulty, subjective | No |
| TTHIN | Minutes thinking | No |
| PTHIN | Percent attention thinking, subjective | No |
| TMOD | Minutes modifying | No |
| PMOD | Percent attention modifying, subjective | No |
| TTEST | Minutes testing | No |
| PTEST | Percent attention testing, subjective | No |
| TASKTIME | TTHIN+TMOD+TTEST | Synthesized |

Each dependent variable from the task data is now formally defined. More accurate variable definitions are followed by examples. These variables were collected either automatically or by the task questionnaire from Appendix C.

MODULES is the number of modules that the subject changed in order to complete the task. A module is a file that contains a collection of related functions and data definitions. Initially, both systems had the same number of modules. No task required the creation or deletion of modules and no subjected chose to create or delete any modules. Therefore, MODULES represent the number of files that required editing to complete the task. A large value for MODULES indicates that the subject edited many files.

SECTIONS refers to the number of sections changed in order to complete the task. A section is a contiguous series of code statements, all of which were changed. The number of sections changed in each module is added up for all modules and is represented by the variable SECTIONS. For some modules, if the subject edits one in three disjoint places and another in five disjoint places, then the SECTIONS variable will have a value of eight. A large number means that many different sections of code required editing; the changes were not isolated. This value is generated by the VMS "DIFFERENCE" program which compares the subjects's program to the original source code.

LINES is the number of lines different from the original module to the edited version of the module. For example, module A contains 30 lines. The subject edits six of these lines and adds 3 new ones. The LINES variable is therefore nine, since there are nine lines different from the original to the new version. This value is generated by the VMS "DIFFERENCE" program which compares the subject's program to the original source code.

TOTLINES is the difference in the module size from the original module to the edited version of the module. As in the previous example, the TOTLINES would be three, since that is the difference in the file sizes. This value is the positive difference in the size of the subjects's files compared to the original files.

CERR stands for <u>compiler errors</u>, which is the number of compiles that fail by resulting in error messages. All compiles that fail result in a message of some kind. Therefore, CERR is

the number of unsuccessful compiles.

TC stands for total compiler errors, which is the total number of compiler error messages. Every compile that fails results in a number of errors. The number of errors is output by the compiler and the end of the compilation. TC is the sum of those compiler error numbers over all failed compilations.

LE stands for the number of linking errors, which result by improperly binding the object code to the system libraries. Very few linking errors were reported and it is somewhat of an artifact of using a VAX/VMS system.

RE stands for runtime errors. This is the number of times that the program aborts when the subject tests it. A runtime error occurs when the operating system issues an error message concerning the program and halts the program. A runtime error does not occur if the program simply generates incorrect results. Only those testing runs that result in operating system interference are counted.

LGE stands for logic errors. Each test run that produces incorrect results, but does not cause interference by the operating system, is counted as a logic error. These indicate the runs for which the program must be re-edited to make work.

TOTERR is the sum of CERR, TC, LE, RE and LGE. It is a measure of the total "difficulty" had by the subject in performing the task. It is a direct arithmetic sum, with no weights.

STHIN is the subject's subjective appraisal of how difficult he or she thought the thinking portion of the task was. It is on an anchored 1 to 9 scale.

SMOD is the subject's subjective appraisal of how difficult he or she thought the modification portion of the task was. It is on an anchored 1 to 9 scale.

STEST is the subject's subjective appraisal of how difficult he or she thought the testing portion of the task was. It is on an anchored 1 to 9 scale.

SALL is the subject's subjective appraisal of how difficult he or she thought the entire task

was. It is on an anchored 1 to 9 scale.

TTHIN is the number of minutes the subject spent in the thinking portion of the task, rounded to the nearest quarter hour by the subject.

PTHIN is the percent attention the subject thought he or she was giving to the thinking task. It is a subjective measure of difficulty.

TMOD is the number of minutes the subject spent in the modification portion of the task, rounded to the nearest quarter hour by the subject.

PMOD is the percent attention the subject thought he or she was giving to the modification task. It is a subjective measure of difficulty.

TTEST is the number of minutes the subject spent in the testing portion of the task, rounded to the nearest quarter hour by the subject.

PTEST is the percent attention the subject thought he or she was giving to the testing task. It is a subjective measure of difficulty.

TASKTIME is the sum of the TTHIN, TMOD and TTEST is represents the total amount of time the subject spent on the task.

Table 5 contains the means and standard error values for the task data collected. Under the heading "Variable" is the formal name of the variable that was measured. The "mean A" column contains the mean of all values of the variable for Group A. Likewise the "mean B" column contains the mean of all values of the variable for Group B. The "stderr A" column contains the standard error that was calculated for Group A. "stderr B" is the standard error for the values of the variable for Group B.

**Table 5. Raw Task Data, Averaged Between Groups**

| Variable | mean A | mean B | stderr A | stderr B |
|---|---|---|---|---|
| MODULES | 2.21 | 1.95 | 0.12 | 0.83 |
| SECTIONS | 6.95 | 6.93 | 0.74 | 0.83 |
| LINES | 69.23 | 67.40 | 8.76 | 9.67 |
| TOTLINES | 46.67 | 48.51 | 2.67 | 4.67 |
| CERR | 2.50 | 2.41 | 0.52 | 0.29 |
| TC | 10.02 | 7.02 | 2.75 | 0.98 |
| LE | 0.18 | 0.25 | 0.06 | 0.07 |
| RE | 0.90 | 1.41 | 0.17 | 0.28 |
| LGE | 1.80 | 1.37 | 0.28 | 0.20 |
| TOTERR | 15.40 | 12.47 | 3.08 | 1.30 |
| STHIN | 2.53 | 3.77 | 0.14 | 0.21 |
| SMOD | 2.97 | 4.51 | 0.21 | 0.21 |
| STEST | 2.64 | 3.95 | 0.19 | 0.26 |
| SALL | 2.78 | 3.95 | 0.17 | 0.17 |
| TTHIN | 31.90 | 35.70 | 3.69 | 3.90 |
| TMOD | 77.00 | 67.90 | 8.05 | 6.70 |
| TTEST | 46.50 | 40.90 | 7.23 | 4.97 |
| TASKTIME | 155.40 | 144.50 | 10.60 | 10.50 |

**Table 6. Raw Task Data, Averaged Between Languages**

| Variable | 'C' | Objective-C | stderr 'C' | stderr 'Obj-C' |
|---|---|---|---|---|
| MODULES | 2.27 | 1.90 | 0.09 | 0.11 |
| SECTIONS | 8.73 | 5.15 | 0.96 | 0.46 |
| LINES | 98.93 | 37.70 | 11.40 | 2.90 |
| TOTLINES | 62.90 | 32.30 | 4.20 | 1.82 |
| CERR | 2.68 | 2.23 | 0.52 | 0.28 |
| TC | 11.85 | 5.18 | 2.77 | 0.74 |
| LE | 0.22 | 0.22 | 0.06 | 0.05 |
| RE | 0.77 | 1.55 | 0.24 | 0.22 |
| LGE | 1.55 | 1.62 | 0.24 | 0.26 |
| TOTERR | 17.07 | 10.80 | 3.15 | 1.09 |
| STHIN | 3.00 | 3.30 | 0.21 | 0.19 |
| SMOD | 3.87 | 3.61 | 0.25 | 0.22 |
| STEST | 3.35 | 3.25 | 0.25 | 0.25 |
| SALL | 3.49 | 3.38 | 0.19 | 0.19 |
| TTHIN | 32.00 | 35.70 | 3.19 | 4.33 |
| TMOD | 70.00 | 74.80 | 7.47 | 5.54 |
| TTEST | 42.50 | 44.80 | 6.42 | 6.00 |
| TASKTIME | 144.70 | 155.30 | 10.21 | 10.90 |

**Table 7. Raw Task Data, Averaged Between Tasks**

| Variable | Task 2 | Task 3 | Task 4 | Err-2 | Err-3 | Err-4 |
|---|---|---|---|---|---|---|
| MODULES | 2.10 | 2.12 | 2.02 | 0.15 | 0.14 | 0.10 |
| SECTIONS | 7.10 | 6.87 | 6.85 | 1.16 | 0.88 | 0.85 |
| LINES | 73.80 | 62.20 | 68.90 | 15.10 | 8.81 | 8.80 |
| TOTLINES | 47.45 | 44.52 | 50.80 | 7.02 | 3.10 | 2.50 |
| CERR | 3.60 | 1.92 | 1.85 | 0.75 | 0.29 | 0.34 |
| TC | 13.50 | 6.75 | 5.30 | 3.84 | 1.43 | 1.33 |
| LE | 0.30 | 0.20 | 0.15 | 0.07 | 0.09 | 0.06 |
| RE | 1.80 | 0.80 | 0.87 | 0.38 | 0.22 | 0.21 |
| LGE | 1.70 | 1.05 | 2.00 | 0.30 | 0.24 | 0.35 |
| TOTERR | 20.90 | 10.70 | 10.20 | 4.28 | 1.82 | 1.59 |
| STHIN | 3.45 | 2.97 | 3.04 | 0.25 | 0.24 | 0.23 |
| SMOD | 4.21 | 3.21 | 3.80 | 0.31 | 0.27 | 0.26 |
| STEST | 3.37 | 3.26 | 3.26 | 0.31 | 0.30 | 0.25 |
| SALL | 3.88 | 3.00 | 3.40 | 0.25 | 0.23 | 0.21 |
| TTHIN | 47.37 | 30.25 | 23.90 | 6.14 | 4.07 | 2.00 |
| TMOD | 86.10 | 66.30 | 64.90 | 10.93 | 7.91 | 7.79 |
| TTEST | 52.00 | 40.10 | 39.00 | 9.73 | 6.62 | 5.82 |
| TASKTIME | 185.00 | 136.70 | 128.00 | 14.70 | 12.60 | 9.21 |

Table 6 contains the means and standard error values for the task data collected averaged over the language variable. Under the heading "Variable" is the formal name of the variable that was measured. The "C" column contains the mean of all values of the variable for the language 'C'. Likewise the "Objective-C" column contains the mean of all values of the variable for the language Objective-C. The "stderr 'C'" column contains the standard error that was calculated for language 'C'. "stderr Obj-C" is the standard error for the values of the variable for the language Objective-C.

Table 7 contains the means and standard error values for the task data collected averaged among the three tasks. Under the heading "Variable" is the formal name of the variable that was measured. The "Task 2" column contains the mean of all values of the variable for Task 2. Likewise the "Task 3" and "Task 4" columns contain the means of all values of the variable for tasks 3 and 4, respectively. The "Err-2" column contains the standard error that was calculated for Task 2. Finally, the "Err-3" and "Err-4" columns contain the the standard error that was calculated for tasks 3 and 4, respectively.

## Conclusion

These tables present a summary of the data collected in this experiment. The actual raw data appear in Appendix B. While the means and standard error values can give an impression of the trends in the data, it is important to use statistical method to insure that unequal numbers are indeed statistically significant. The results of the statistical analysis appear in Chapter 5.

# Chapter 5 Results

## Introduction

This experiment collected two sets of data. The first set describes the subjects and is used to show homogeniety among subjects and between groups. It is called the "student" data. The second set is the actual experimental data. These data are generated by the questionnaires the subjects filled out for each task they performed. It is called the "task" data. The student data are used to show that the experiment was free of internal bias. The task data are used to make claims about the abilities of the languages 'C' and Objective-C. The student data are described first. The task data are described in the section called "Task data".

## Significant Values for Student Data

An analysis of variance (ANOVA) test was performed on each of the variables using subject identifier (1 through 10) and group (A or B) as discriminating classifications. The design provides that subject is nested within group. This yields a between subjects design over the group classification. The statistical significance of each variable is presented in table 8.

Except for CSQCA, the computer science Q.C.A of the subject, every variable in table 8 shows no statistically significant differences between Group A and Group B. The "-NA-" notation in the F-table column for variables "CURRIC" and "OBJC" mean that there was no

difference in the data at all; every value was identical. This is because every subject was a computer science student (CS curriculum) and no student had previous experience with Objective-C; they had all learned the language during the first part of the experiment.

## Significant Values for Task Data

An analysis of variance (ANOVA) test was performed on each of the above variables using subject identifier (1 through 10), group (A or B), language (C or OBJECTIVE-C) and task (2 through 4). The design provides that subject is nested within group, which is crossed with language and task. This yields a two by two by three design with ten observations per entry. The statistical significance of each variable is presented in table 9.

Table 9 shows the variables and discriminants over which statistically significant differences were found. The "Discriminant" column lists the independent variable for which a significant results was found. The "Confidence" column lists the confidence limit that the variable satisfies. The F-table(df) value is the actual ratio of the variable to the error term, which is actually the probability that the significance of the difference is due to random error in the experiment.

As can be seen, nearly every measure is dependent on some independent variable. Only LE, the number of linking errors, STEST, the subjective difficulty of testing the results of the task, and TMOD, the number of minutes used to make the modification for the task, have no significance at all. All others are significant over at least one discriminant and a few are significant over more than one discriminant.

Additionally, some variables are dependent on a combination two independent variables. This is denoted in the discriminant column by joining the two variables with an asterisk. Therefore, "SUBJECT*TASK" indicates that the combined independent variables result in a statistically significant difference in the data values. This is used later to find interactions between conditions on independent variables.

**Table 8. Results of ANOVA on Student Data Variables over Group**

| Variable | Confidence | F-table(df) | Significant |
| --- | --- | --- | --- |
| QCA | | 0.2706 | No |
| CSQCA | 5% | 0.0347 | Yes |
| CURRIC | | -NA- | No |
| C | | 0.8539 | No |
| PASCAL | | 0.1853 | No |
| OBJC | | -NA- | No |
| SMALLT | | 0.3306 | No |
| INTEGR | | 0.2853 | No |
| TESTX | | 0.8672 | No |
| LEVEL | | 0.5560 | No |
| COURSES | | 0.1686 | No |
| SUBJTASK | | 0.1395 | No |
| SUBJQUES | | 0.7730 | No |

Table 9. **Results of ANOVA on Task Data Variables**

| Variable | Discriminant | Confidence | F-table(df) |
|---|---|---|---|
| MODULES | LANGUAGE | 5% | 0.0410 |
| SECTIONS | LANGUAGE | 1% | 0.0023 |
| LINES | LANGUAGE | .01% | 0.0001 |
| TOTLINES | LANGUAGE | .01% | 0.0001 |
| CERR | TASK | 1% | 0.0013 |
| | SUBJECT*TASK | 5% | 0.0340 |
| TC | SUBJECT | 5% | 0.0415 |
| LE | NONE | | |
| RE | LANGUAGE | 5% | 0.0310 |
| | TASK | 1% | 0.0088 |
| LGE | TASK | 5% | 0.0380 |
| | SUBJECT*TASK | 5% | 0.0136 |
| TOTERR | TASK | 1% | 0.0039 |
| | SUBJECT | 5% | 0.0306 |
| STHIN | GROUP | 1% | 0.0085 |
| SMOD | GROUP | 1% | 0.0017 |
| | TASK | 1% | 0.0027 |
| STEST | NONE | | |
| SALL | TASK | 1% | 0.0039 |
| | GROUP | 1% | 0.0096 |
| TTHINK | TASK | 1% | 0.0100 |
| TMOD | NONE | | |
| TTEST | SUBJECT*TASK | 1% | 0.0084 |
| | TASK | 5% | 0.0330 |
| TASKTIME | TASK | 1% | 0.0018 |

Table 10. ANOVA on Task Data Variables Omitting Task 2

| Variable | Discriminant | Confidence | F-table(df) | Same |
|---|---|---|---|---|
| MODULES | LANGUAGE | 5% | 0.0402 | Yes |
| SECTIONS | NONE | | | No |
| LINES | LANGUAGE | 1% | 0.0058 | Yes |
| TOTLINES | LANGUAGE | .01% | 0.0001 | Yes |
| CERR | NONE | | | No |
| TC | NONE | | | No |
| LE | NONE | | | No |
| RE | NONE | | | No |
| LGE | TASK | 5% | 0.0317 | Yes |
| TOTERR | NONE | | | No |
| STHIN | GROUP | 5% | 0.0192 | Yes |
| SMOD | GROUP | 1% | 0.0027 | Yes |
| | TASK | 5% | 0.0150 | |
| STEST | NONE | | | Yes |
| SALL | GROUP | 5% | 0.0179 | No |
| TTHINK | NONE | | | No |
| TMOD | SUBJECT*TASK | 5% | 0.0484 | No |
| TTEST | NONE | | | No |
| TASKTIME | NONE | | | No |

## Faults in the Data

According to table 9, there are nine variables in which the task is statistically significant. The entries of table 7 has task 2 values that are very different from the other two tasks. This can be determined by examining the means and standard error values for table 7. Task 2 is frequently two or three standard errors from the mean of the other tasks. Since the tasks were designed to be similar, this may cloud the statistics. It is possible that task 2 is not on the same level of difficulty as the other tasks or that it is somehow flawed. Repeating the analysis of variance omitting task 2 yields the following statistics for each variable. Table 10 shows those values.

Many of the dependencies on the task have disappeared in omitting task 2. It is most likely then that task 2 contained some flaw that made it too different from the other tasks and is not a good indicator for the experiment. Of the nine variables from the ANOVA in table 9 that were dependent on TASK, only two remain in table 10 after eliminating task 2 as invalid.

## Conclusion

These tables represent the analysis of the all of the data collected in the experiment. By determining which variables are significant and noting the trends as described by the means from tables 5 through 7 it is possible to make definite conclusions about the effectiveness of Objective-C during the maintenance tasks. These conclusions appear in Chapter 6.

# Chapter 6 Conclusions

## Introduction

In order to reach meaningful conclusions, the data must first be shown to be free of any bias. First, the student data will show that there are no significant differences between the two groups of students. Next, the task data will show that there is an expected bias in the tasks that were given to the subjects. After removing this fault, the final conclusions are shown.

The ANOVA results in table 8 for all the measures taken for the student data show that only one is statistically significant between the two groups. The CSQCA is significantly different while all other variables are not. Random assignment of subjects to groups does not guarantee that all variables will be uniform. However, it does attempt to reduce the probability of a difference between the groups.

Therefore it is most likely that the random assignment of students caused one group to have a different average computer science QCA from the other group. This may cause some overall effect in the results. However, all other variables that were measured for the students report no significant difference between them. This means that differences in the computer science QCA are an exception and have probably little effect on the resulting conclusions.

The task data contain a large number of statistically significant variables as can be seen in table 9. Both the significantly different variables and the not significant variables yield useful

information about the experiment. These conclusions are presented first in a summary form and then as a detailed description of the implications of each variable.

Table 11 presents a synopsis of the conclusions for each variable. The table lists the dependent variables of the task data under the "variable" column. A review of the purpose of the variable follows under the "synopsis" column. The independent variable that causes the statistically significant results appears under the "Discrim" heading. Lastly, a brief description of the conclusion reached by the variable is presented under the "Means" heading.

Many variables are dependent on only one independent variable, such as SUBJECT, GROUP, LANGUAGE or TASK. For these variables, the actual meaning can be found by comparing the means of the task data values, averaged over the appropriate independent variable. If the discriminating variable is GROUP, examine the means in table 5. If the discriminating variable is LANGUAGE, examine the means in table 6 and if the discriminating variable is TASK, examine the means and standard error values in table 7.

No table of means is provided for the SUBJECT independent variable since no dependent variable holds a significant difference based solely on the SUBJECT. This means for all observations that no singular subject performed significantly differently from any other subject. This reinforces the fact that the subjects are homogenous.

In addition to the variables dependent on only one independent variable, some are dependent on a combination of two independent variables. This is denoted in tables 11 and 12 by conjoining the two independent variables with the word "by". Realistically, this means that the two independent variables interacted in a way that produces results different from either variable acting alone. In many cases, neither variable by itself is significant, but the two combined provide unique insight to the structure of the data.

**Table 11. Synopsis of Conclusions from the Task Data**

| Variable | Synopsis | Discrim | Means |
|---|---|---|---|
| MODULES | number of changed modules | LANG | Objective-C requires changing fewer modules |
| SECTIONS | number of changed sections | LANG | Objective-C requires changing fewer sections |
| LINES | number of different lines | LANG | Objective-C requires changing fewer lines |
| TOTLINES | number of new lines | LANG | Objective-C requires adding fewer lines |
| CERR | failed compilations | TASK | Task 2 require produces more failed compiles |
| | | SUBJECT by TASK | Subjects reacted differently to different tasks |
| TC | total of compiler errors | SUBJECT | Some subjects produce more compiler errors |
| LE | number of linking errors | -none- | |
| RE | number or runtime errors | LANG | Objective-C produces more runtime errors |
| | | TASK | Task 2 produces more runtime errors |
| LGE | number of logic errors | TASK | Task 2 produces more logic errors |
| | | SUBJECT by TASK | Subject and task interacted to produce more logic errors |
| TOTERR | total error count | TASK | Task 2 produces more total errors than other tasks |
| STHIN | subjective thinking difficulty | GROUP | Group B finds thinking about task more difficult |
| SMOD | subjective modifying diff. | GROUP | Group B finds modifying task more difficult |
| | | TASK | Subjects found task 2 more difficult to modify |
| STEST | subjective testing difficulty | -none- | |
| SALL | total subjective difficulty | TASK | Subjects found task 2 more difficult overall |
| | | GROUP | Group B finds tasks more difficult overall |
| TTHINK | minutes spent thinking | TASK | Task 2 required more time to think about than other tasks |
| TMOD | minutes modifying | -none- | |
| TTEST | minutes spent testing | SUBJECT by TASK | Subjects and task interacted to increase testing time |
| | | TASK | Task 2 required longer to test |
| TASKTIME | total time on task | TASK | Task 2 required longer than other tasks to complete |

Each dependent variable from the task data is explained here in detail. A more accurate definition is given for the purpose of the variable and the direct implications of a significant difference is explained. After this, the relationships between the conclusions of the various variables are combined into a summary of the results.

MODULES: the number of modules that were changed. Objective-C requires changing fewer modules since it is discriminated by LANGUAGE and Objective-C has a smaller mean compared to 'C'. This means that a typical request for change will cause fewer files of the program to be changed.

SECTIONS: the number of sections that were changed. Objective-C requires changing fewer sections since it is discriminated by LANGUAGE and Objective-C has a smaller mean compared to 'C'. This means that a typical enhancement request will cause fewer places in a body of code to be change. This implies that the changes are more localized, less spread throughout the program.

LINES: the number of lines that were changed. Objective-C requires changing fewer lines of code since it is discriminated by LANGUAGE and Objective-C has a smaller mean compared to 'C'. This means that a typical enhancement request requires less manipulation of code to implement for Objective-C than 'C'.

TOTLINES: the number of lines added to the file. Objective-C requires adding fewer lines of new code since it is discriminated by LANGUAGE and Objective-C has a smaller mean compared to 'C'. This means that a typical enhancement request requires less lines of code to add to implement.

CERR: the number of failed compilations. Task 2 produces more failed compilations than other task since it is discriminated by TASK and Task 2 has the highest error count. This means that Task 2 has some flaw that is apt to produce more syntax errors thereby causing more failed compilations. However, it is also true, since the SUBJECT * TASK discriminant is significant, that the subject interacted with the task to produce a higher number of compilation errors. This could be true because some subjects find their own syntax errors while others allow the compiler to find their syntax errors for them.

TC: total number of compiler errors. Some subjects cause more compiler errors than others since it is discriminated by SUBJECT. This means that some subjects habitually report many syntax errors, while others report only a few. Again, this is probably due to differing philosophies for using compilers. Some subjects allow the compiler to find their errors for them, while others find their errors first before giving their code to the compiler.

LE: the number of linking errors. Neither language produces any more linking errors than the other.

RE: the number of runtime errors. Objective-C produces more runtime errors since it is discriminated by LANGUAGE and Objective-C has a higher mean. Additionally, Task 2 produces more runtime errors since it is discriminated by TASK and Task 2 has the largest mean. These two observations probably result from different conditions, since there is no LANGUAGE * TASK interaction. Since Objective-C does run time checking of its messages (which are like procedure calls) while 'C' does not, it is possible to see that Objective-C is capable of generating a greater number of runtime errors. As for task, some flaw in task 2 probably accounts for it causing more runtime errors.

LGE: the number of logic errors. Task 2 produces more logic errors since it is discriminated by TASK and Task 2 has the largest mean. Additionally, the SUBJECT and TASK interact to produce more logic errors. This probably mean that different subjects have viewed task 2 differently, causing more logic errors. Or that some flaw in task 2 causes some of the subjects to produce incorrect code which will generate logic errors. In either case, it is probably task 2 which is causing the problem.

TOTERR: the total error count. Task 2 produces more total errors than other tasks since TOTERR is discriminated by TASK and Task 2 has the largest mean. This means that something about task 2 causes it generate more errors than the other tasks. Probably, task 2 contains some flaw that makes it unlike the other tasks. The tasks were designed to be of equal difficulty, but this disproves that.

STHIN: the subjective difficulty of thinking about the task. Group B subjects, the ones who used Objective-C first for each task, found thinking about the task more difficult than Group A subjects since STHIN is discriminated by GROUP and Group B has a higher mean. This

means that the subjects who used Objective-C first perceived the thinking portion of the task as being more difficult than those subjects who used 'C' first. This is probably because Objective-C is a super-set of 'C' and therefore has more programming options for implementing a task.

SMOD: the subjective difficulty in modifying the task. Group B subjects, the ones who used Objective-C first for each task, found modifying each task more difficult than Group A subjects since SMOD is discriminated by GROUP and Group B has a higher mean. Additionally, subjects found task 2 more difficult to modify than other tasks since SMOD is discriminated by TASK and task 2 has a higher mean than other tasks. These two effects probably have no relationship, since the GROUP * TASK discriminant is not significant. The difference between the groups exists for the same reason as for the previous variable. The problem with task 2 is the same as was noted before; some flaw in the task that causes it to be different from most other tasks.

STEST: the subject difficulty in testing the task. Neither group had any more difficulty than the other in testing the task.

SALL: the overall difficulty with the task. Group B subjects, the ones who used Objective-C first for each task, found each task overall more difficult than Group A subjects since SALL is discriminated by GROUP and Group B has the higher mean. Additionally, subjects perceived Task 2 as being more difficult overall since SALL is discriminated by TASK and Task 2 has the highest mean. Again, these two effects are probably not related and belong to the same effects as listed above. The differences in the groups is due to subject perceptions of Objective-C being more difficult and Task 2 is simply too much unlike the other tasks.

TTHINK: the number of minutes spent thinking about the task. Task 2 required more time think about than other tasks since TTHINK is discriminated by TASK and Task 2 has the highest mean. This means that something in the nature of Task 2 required subjects to spend more time thinking about it than other tasks. This is probably due to a flaw in the task.

TMOD: the number of minutes spent modifying code for the task. No language, group, subject or task required more time to perform the task than any other.

TTEST: the number of minutes spent testing the code for the task. There was some interaction of subjects with the tasks in testing the code for the tasks since TTEST is discriminated by the interaction of SUBJECT and TASK. Additionally Task 2 took longer to test since TTEST is discriminated by TASK and Task 2 has the highest mean. Both of these effects are probably due to the fact that task 2 is not the same as other tasks. There is some flaw in task two that makes it take longer to test. Additionally, the subjects approached the testing of task 2 differently thereby generating different times for testing. Both of these difficulties are probably due to task 2 problems

TASKTIME: the total number of minutes required for the task: Task 2 requires the most time for the task since TASKTIME is discriminated by TASK and Task 2 has the highest mean. This means again that some flaw in task two causes it take longer to program for all subjects, regardless of which group they are in.

## Task Fault in the Experiment

As can be seen from the above conclusions, especially as presented in table 11, task 2 accounts for important differences in many of the variables. The assumptions of the experiment require that the tasks be of approximately equal difficulty. This is quite obviously violated by task 2. Since the faults of task 2 may obscure other legitimate results, the ANOVA statistics have been repeated in table 10, which omits the data of task 2. The means of the data values are visible in tables 5, 6 and 7.

Table 12 describes the conclusions based on the assumption that task 2 is invalid. It is presented in the same format as table 11. Notice that many variables are no longer dependent on the task and many are not dependent on anything at all. Variables that depend on nothing show that Objective-C is no worse than 'C' for the intended measure.

**Table 12. Summary of Task Data Variables without Task 2**

| Variable | Synopsis | Discrim | Means |
|---|---|---|---|
| MODULES | number of changed modules | LANG | Objective-C requires changing fewer modules |
| SECTIONS | number of changed sections | -none- | |
| LINES | number of different lines | LANG | Objective-C requires changing fewer lines |
| TOTLINES | number of new lines | LANG | Objective-C requires adding fewer lines |
| CERR | number of failed compilations | -none- | |
| TC | total of compiler errors | -none- | |
| LE | number of linking errors | -none- | |
| RE | number or runtime errors | -none- | |
| LGE | number of logic errors | TASK | Task 4 produces more logic errors |
| TOTERR | total error count | -none- | |
| STHIN | subjective thinking difficulty | GROUP | Group B finds thinking about task more difficult |
| SMOD | subjective modifying diff. | GROUP | Group B finds modifying task more difficult |
| | | TASK | Subjects found task 4 more difficult to modify |
| STEST | subjective testing difficulty | -none- | |
| SALL | total subjective difficulty | GROUP | Group B finds tasks more difficult overall |
| TTHINK | minutes spent thinking | -none- | |
| TMOD | minutes modifying | SUBJECT by TASK | Subjects and task interacted to increase testing time |
| TTEST | minutes spent testing | -none- | |
| TASKTIME | total time on task | -none- | |

Each dependent variable from the task data is re-explained in detail. The direct implications of a significant difference is explained now assuming that task 2 was flawed and produced biased data.

MODULES: the number of modules that were changed. Objective-C requires changing fewer modules since it is discriminated by LANGUAGE and Objective-C has a smaller mean compared to 'C'. This means that a typical request for change will cause fewer files of the program to be changed.

SECTIONS: the number of sections that were changed. Objective-C may require changing fewer sections since it is barely discriminated by LANGUAGE and Objective-C has a smaller mean compared to 'C'. This means that a typical enhancement request will cause fewer places in a body of code to be change. This implies that the changes are more localized, less spread throughout the program. Previously, this result had been much stronger.

LINES: the number of lines that were changed. Objective-C requires changing fewer lines of code since it is discriminated by LANGUAGE and Objective-C has a smaller mean compared to 'C'. This means that a typical enhancement request requires less manipulation of code to implement for Objective-C than 'C'.

TOTLINES: the number of lines added to the file. Objective-C requires adding fewer lines of new code since it is discriminated by LANGUAGE and Objective-C has a smaller mean compared to 'C'. This means that a typical enhancement request requires less lines of code to add to implement.

CERR: the number of failed compilations. No task produces more compiler errors than any other. Task 2 has been eliminated as the cause of the distinction.

TC: total number of compiler errors. No task produces more total compiler errors than ay other. Task 2 has been eliminated as the cause of the distinction.

LE: the number of linking errors. Neither language produces any more linking errors than the other.

RE: the number of runtime errors. No task produces more runtime errors than any other. Task 2 has been eliminated as the cause of the difference.

LGE: the number of logic errors. Task 3 now produces more logic errors since it is discriminated by TASK and Task 3 has the largest mean. Previous task 2 was the source of logic errors. This probably indicates that logic errors is not a useful measure for comparing languages since the number of logic errors is too dependent on the task that is being implemented to yield any useful information about the differences in the languages or methodologies.

TOTERR: the total error count. No task produces more total errors than any other. Task 2 has been eliminated as the cause of the distinction.

STHIN: the subjective difficulty of thinking about the task. Group B subjects, the ones who used Objective-C first for each task, found thinking about the task more difficult than Group A subjects since STHIN is discriminated by GROUP and Group B has a higher mean. This means that the subjects who used Objective-C first perceived the thinking portion of the task as being more difficult than those subjects who used 'C' first. This is probably because Objective-C is a super-set of 'C' and therefore has more programming options for implementing a task. This is not different from the previous analysis.

SMOD: the subjective difficulty in modifying the task. Group B subjects, the ones who used Objective-C first for each task, found modifying each task more difficult than Group A subjects since SMOD is discriminated by GROUP and Group B has a higher mean. Additionally, subjects found task 3 more difficult to modify than other tasks since SMOD is discriminated by TASK and task 3 has a higher mean than other tasks. These two effects probably have no relationship, since the GROUP * TASK discriminant is not significant. While the difference between groups is expected and explained above, there is no accounting for the differences in the tasks. For unknown reasons, task 3 was perceived as being more difficult to modify.

STEST: the subject difficulty in testing the task. Neither group had any more difficulty than the other in testing the task.

SALL: the overall difficulty with the task. Group B subjects, the ones who used Objective-C first for each task, found each task overall more difficult than Group A subjects since SALL is discriminated by GROUP and Group B has the higher mean. The differences between the groups has been explained above. TASK is no longer significant difference. Task 2 has been eliminated as the source of the distinction.

TTHINK: the number of minutes spent thinking about the task. No task is more difficult to think about than any other. TASK is no longer a significant difference. Task 2 has been eliminated as the source of the distinction.

TMOD: the number of minutes spent modifying code for the task. Previous, TMOD had not had any significant differences. However, omitting task 2 leads to the appearance of an interaction between the subjects and the task without there being a significant difference for either the subjects or the task alone. This observation, then, is probably spurious and has relation to any independent variable.

TTEST: the number of minutes spent testing the code for the task. TASK is no longer a significant difference. Task 2 has been eliminated as the source of the distinction.

TASKTIME: the total number of minutes required for the task: TASK is no longer a significant difference. Task 2 has been eliminated as the source of the distinction.

In the original data, nine of the variables were dependent on TASK as a discriminating factor. However, after omitting task 2 from the analysis, only two of the variables were dependent on TASK. These dependencies are probably legitimate and the remaining inferences are free from invalid data brought on by task 2. The remaining conclusions are based on the inferences by omitting task 2.

## Supporting the Hypothesis

This experiment supports the hypothesis that object-oriented languages produce more maintainable code than procedure-oriented languages. The conclusion is reached by observing the data collected above. For four main code variables, Objective-C produces code

that requires fewer modules to be edited, fewer sections to be edited, fewer lines of code to be changed and fewer new lines to be added. This leads to the conclusion that Objective-C produces code in which changes are more localized, less global to the entire system than other languages. For the other variables, Objective-C is no worse that 'C', indicating that while it yields better results for some variables, under no conditions does it generate results that are any worse. The number of sections changed shows marginally statistically significant results.

No other variables show statistically significant results over the discriminant of language. Objective-C produces results that are as least as good as those of 'C'. In addition, subjects had no previous training in either object-oriented languages or in Objective-C language. However, they did have significant training in Pascal and structured programming. This gives even more support to the power of Objective-C over 'C' since the data yielded good results even though there was a bias from the subjects toward the 'C' paradigm.

Finally, it is important that three of the four subjective variables showed that Group B subjects perceived the tasks to be more difficult than Group A subjects did. When implementing the tasks, Group B always used the Objective-C language first and then used the 'C' language. Group A did the reverse. Group A always used the 'C' language first and then used the Objective-C language for the task. This means that, in general, subjects using Objective-C for a task before using 'C' found the task to be more difficult.

A possible explanation for this is that Objective-C is a super-set of 'C' and therefore more options are available. Objective-C contains additional mechanisms that allow the object-oriented treatment of code, such as messaging, encapsulation and inheritance that 'C' does not have. Additionally, Objective-C also contains all of the constructs that are already present in 'C'. These additional options may require more thought and more decisions from the subject. The subject may have been unsure, due to lack of experience with Objective-C, as to when to use a message and when to use a procedure call.

The drawback to this theory is that only the subjective variables show Objective-C as being perceived as being more difficult when performed before using 'C'. No objective variable confirms the idea that Objective-C takes longer to program, produces more changes in the source code or produces more errors. In fact the reverse is true. Objective-C requires fewer

changes to the code than 'C'. Objective-C also has no more errors than 'C' and takes no more time to effect a change than 'C'.

In short, no objective variable supports the idea that Objective-C is more difficult than 'C'. The difficulty that the subjects encountered with using Objective-C first is only a problem of perception. This may account for the resistance with which object-oriented languages and methodologies have been met. They are perceived as being more difficult, but actually require no more time and produce no more errors than existing languages. And in addition, the changes are more localized than if a procedural language were used.

While it is not possible to fully explain why these perceptions exist without further study, the bias that subjects have towards structured design and functional decomposition probably accounts for most of it. Students today are taught software engineering techniques that emphasize hierarchical nesting of procedures and control-flow based computing paradigms. While these are useful within their own realm, they make new languages and new methodologies difficult for all types of developers -- from programmers through system architects -- to accept.

The knowledge that this bias exists may be particularly useful to managers and teachers when considering switching to an object-oriented language. While it may be resisted or perceived as being more difficult, there are no objective data that supports that fact that it is any more difficult or causes any greater number of errors. While this perception of difficulty could therefore be overcome by teaching object-oriented methodologies from the start, frequently that route is not an option. But knowing that the student's fears are unfounded can help to conquer them.

## Future Work

Teaching object-oriented methodology from the start may remove the perception that object-oriented languages are more difficult. The only way to be sure is to try an experiment with subjects that have no methodology training. Teaching some subjects object-oriented methodology and others structured design and then performing experiments is the only way to rigorously verify the results. Most of the future work from this thesis will consist of

performing new experiments.

Other experiments should change the base program being used for maintenance. It has often been suggested that different methodologies are better suited for different applications. Perhaps 'C' is the best choice for operating system construction while Objective-C is best for user-interface applications. The only way to find out is to perform an experiment.

Of even greater interest is determining if object-oriented languages are better for the entire lifecycle. This experiment has shown that the maintenance portion is improved by using an object-oriented language, but testing the entire lifecycle would have a much greater impact since a program has to maintained in the same language it was written in.

Of course, there are many more experiments to be tried to show the superiority of object-oriented programming of existing programming methodologies. And only through careful experimentation can results be verified. Rash statements and claims of increased productivity or superior code can only be supported through rigorous experimentation and analysis of data.

## Conclusion

The final conclusion of this study is that Objective-C produced fewer changes in the source code and that these changes were more localized. For all other variables, there were no significant differences, indicating that Objective-C is no worse than 'C' as far as maintainability. While having changes that were more localized did not reduce the error rate, that may be a result of the scope of the experiment. It seems likely that on much larger systems, of say 10,000 lines being maintained for many months or years, localizing changes will have a much stronger impact in reducing both the number of errors encountered and the amount of time to effect a change. Of course, this can only be supported by sound experimentation. Hopefully, this experiment will open the way for more tests to verify the claims of other methodologies.

# Bibliography

[ALFM77] Alford, M.W., "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, Jan 1977, pp 60-69.

[ATWM78] Atwood, M.E. and Ramsey, H.R., "Cognitive structures in the comprehension and memory of computer programmers: An investigation of computer debugging", Technical Report T.R.-78-A21, U.S. Army Research Institute for the Behavioral and Social Sciences, Alexandria, Virginia, August, 1978.

[BAKF75] Baker, F.T., "Structured Programming in a Production Programming Environment," IEEE Transactions on Software Engineering, June 1975, Vol. SE-1, No. 2, pp.241-252.

[BARM69] Barnett, M.P., Ruhsam, W.M., "SNAP: An experiment in natural language programming", AFIPS Conference Proceedings, 34, Montvale, New Jersey, 1969, pp 75- 87.

[BARM77] Bariff, M.L., and Lush, E.J., "Cognitive and personality test for the design of management information systems", Management Science, 23, 8, April 1977, pp 820-829.

[BATD77] Bates, D. (ed.) (1977), Software Engineering Techniques, Infotech state of the art report, Infotech, England.

[BASV75] Basili, V., Turner, A., "Iterative Enhancement: A Practical Technique for Software Development," IEEE Transactions of Software Engineering, Vol SE-1, no. 4, 1975, pp 390-396.

[BERG81] Bergland, G.D., "A Guided Tour of Program Design Methodologies," IEEE Computer, October 1981, pp 13-35.

[BOEB73] Boehm, B.W., Brown, J.R., Lipow, M., "Quantitative evaluation of software quality", Software Phenomenology Working Papers of the Software Lifeycle Management Worksop, August 1977, pp 81-94.

[BOEC66] Boehm, C. and Jacopini, G., (1966), "Flow Diagrams, Turing Machines and Languages with only Two Formation Rules," Communications of the ACM, Vol. 9, No. 5, pp.366-371.

[BOED84] Boehm-Davis, D., Ross, L., "Approaches to Structuring the Software Development Process," Technical Report GEC/DIS/TR-84-B1V-1, Software Management Research Data & Information Systems, General Electric Co., Arlington, Va 22202, Oct. 1984.

[BOED86] Boehm-Davis, D., Holt, R., Schultz, A., Stanley, P., "The role of program structure in software maintenance," Technical Report TR-86-GMU-P01, Psychology Department, George Mason University, Fairfax, Virginia 22030, May 1986.

[BOIS74a] Boies S. and Gould J., "Syntactic Errors in Computer Programming," Human Factors, 1974, 16(3), 253-257.

[BOIS74b] Boies, S.J., and Gould, J.D., "User behavior on an interactive computer system", IBM systems Journal, 13, 1, 1974, pp 253-257.

[BOOG86] Booch, G., "Object-Oriented Development," IEEE 1986.

[BROF82] Brooks, F., The Mythical Man-Month, Addison-Wesley Publishing Co., Reading, Mass., 1982.

[BROF86] Brooks, F., "No Silver Bullet: Essence and Accidents of Software Engineering," Information Processing '86, H.J. Kugler, ed., Elsevier Science Publishers B.V. (North-Holland) (C) IFIP 1986.

[BROR80] Brooks, R., "Studying Programmer Behavior Experimentally: The Problems of Proper Methodology," Communications of the ACM, 1980, Vol. 23, No. 4, 207-213.

[BROP80] Brown, P., "Why does software die?," Pergamon Infotech Start of the Art Report "Life Cycle Management, "Pergamon Infotech Ltd., 1980.

[CARJ70] Carlisle, J.H., "Comparing behavior at various computer display consoles in time-shared legal information", Rand Corportaion, Santa Monica, CA Report No. AD712695, September 1970.

[CARE77] Carlson, E.D., Grace, B.F. and Sutton, J.A., "Case studies of end user requirements of interactive problem-solving systems, MIS Quarterly, March 1977, pp 51-63.

[COXB84] Cox, B., "Message/Object Programming: An Evolutionary Change in Programming Technology," IEEE Software, vol. 1, no. 1, Jan. 1984.

[COXB86] Cox, B., Hunt, B., "Objects, Icons, and Software-ICs," Byte Magazine, Aug. 1986, 161-176.

[COXB86] Cox, B., Object-Oriented Programming: An Evolutionary Approach, Addison-Wesley Publishing Co., Reading, Mass., 1986.

[CURB79] Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A. and Love, T., "Measuring the psychological complexity of software maintenance tasks with Halstead and McCabe metrics", IEEE Transactions of Software Engineering, SE-5, 2, March 1979, pp 96-104.

[CURB80] Curtis, B., "Measurement and Experimentation in Software Engineering," Proceedings of the IEEE, 1980, Vol. 68, No. 9, pp 1144-1157.

[DANA87] Dandekar, A.V., "A Procedural Approach to the Evaluation of Software Development Methodologies," September, 1987, Virginia Polytechnic Institute, Master's Thesis.

[DAVC76] Davis, C.G., and Vick, C.R., "The Software Development System," Proceedings of the Second International Software Engineering Conference, October 1976.

[DEMT79] DeMarco, T. (1979), Concise Notes on Software Engineering, Yourdon Press, New York, NY.

[DIJE65] Dijkstra, E.W. (1965), "Programming Considered as a Human Activity," Proc. IFP Congress, pp. 213-217.

[DIJE72] Dijkstra, E.W. , Dahl, O.J. and Hoare, C.A.R. (1972), Structured Programming, Academic Press, New York, NY.

[DUNH77] Dunsmore, H.E., and Gannon J.D., "Experimental investigation of programming complexity, Proceedings of Sixteenth Annual ACM Technical Symposium: Systems and Software", Washington, D.C., June 1977, pp 1-14.

[ELSJ76] Elshoff, J.L., "An analysis of some commercial PL/1 programs", IEEE Transactions on Software Engineering, SE-2, 1976, pp 113-121.

[FAIR85] Fairley, R., Software Engineering Concepts, McGraw-Hill Book Co., New York, New York, 1985.

[GANC77] Gane, C., Sarson, T., Structured Systems Analysis: Tools and Techniques, Improved System Technologies, Inc., 1977.

[GANJ76] Gannon, J., "An experiment for the evaluation of language features," Int. J. Man-Machine Studies (1976) 8, pp 61-73.

[GANJ77] Gannon, J., "An Experimental Evaluation of Data Type Conventions," Communications of the ACM, Vol. 20, No. 8, pp 584-595, Aug 1977.

[GOLA83] Goldberg, A., Robson, D., Smalltalk-80, The Language and its Implementation, Addison-Wesley Publishing Co., Reading, Mass., 1983.

[GOUJ75] Gould, J.D., "Some psychological evidence on how people debug computer programs", International Journal of Man-Machine Studies, 7, 1975, pp 151-182.

[HENS81] Henry, S., Kafura, D., "Software Structure Metrics Based on Information

Flow," IEEE Transactions on Software Engineering, Vol. SE-07, No. 5, Sept.1981.

[HENS??] Henry, S., Kafura, D., Harris, K., "On the Relationships among Three Software Metrics," ACM ??

[HENS85] Henry, S., Arthur, J. D., Nance, R. E., "A Procedural Approach to Evaluating Software Development Methodologies," Virginia Polytechnic Institute, Department of Computer Science, Technical Report TR-85-20, March 1985.

[HICC64] Hicks, Charles, Fundamental Concepts in the Design of Experiments, Holt, Rinehart and Winston, Inc., 1964.

[HOLR87] Holt, R., Boehm-Davis, D., Schultz, A., "Mental Representations of Programs for Student and Professional Programmers," Psychology Department, George Mason University, Fairfax, VA 22030.

[JACM75] Jackson, M., "Principles of Program Design", Academic Press, 1975.

[JACJ87] Jacky, J.P., Kalet, I.J., "An Object-Oriented Programming Discipline for Standard Pascal, "Communcations of the ACM, Vol. 30, No. 9, 772-776, Sept. 1987.

[KAFD87] Kafura, D., Reddy, G., "The Use of Software Complexity Metrics in Software Maintenance," IEEE Transactions of Software Engineering, Vol. SE-13, No. 3, March 1987.

[KNUD71] Knuth, D.E., "An Empirical Study of FORTRAN Programs," Software--Practice and Experience, Vol. 1, pp 105-133 (1971).

[KRAK78] Krause, K.W. and Diamant, L.W., "A Management methodology for testing software requirements," Proceedings of the IEEE Conference on Computer Software and Applications, 1978.

[LEEJ78] Lee, J.M. and Shneiderman, B., "Personality and programming: Time-sharing vs. batch processing", Proceedings of the ACM National Conference, 1978, pp 561-569.

[LEHM80] Lehman, M., "Programs, Life Cycles, and Laws of Software Evolution," Proceedings of the IEEE, vol. 68, no. 9, Sep. 1980.

[LIES78] Lientz, B., Swanson, E., Tompkins, G., "Characteristics of Application Software Maintenance," Communications of the ACM, vol. 21, no. 6, Jun. 1978.

[LISB79] Liskov, B., et al, CLU Reference Manual, MIT-TR 225, October, 1979.

[LOVT77] Love, T., "Relating individual differences in computer programming performance to human information processing abilities", Ph.D. Dissertation, University of Washinton, 1977.

[LOVL79] Love, T. and Bowman A., "An Independent Test of the Theory of Software Physics," SIGPLAN Notices 1979.

[LUCH74] Lucas, H.L., Kaplan, R.B., "A Structured Programming Experiment," Computer Journal, Vol. 19, pp 136-138, 1974.

[MACB82] MacLennan, B., "Values and Objects in Programming Languages," SIGPLAN Notices, vol. 17, no.12, p.70, Dec. 1982.

[MATY78] Matsumoto, Y., Nakajima, S., Yamamoto, S. and Sakai, T., "SPS: A Software Production System for Mini-Computers and Micro-Computers," Proceedings of the IEEE Conference on Computer Software and Applications, 1978.

[MCCC75] McClure, C.L. (1975), "Top-Down, Bottom-Up and Structured Programming," IEEE Transactions on Software Engineering, Vol. SE-1, No. 4, pp. 397-403.

[MCCT76] McCabe, T.J., "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976.

[MEYB81] Meyer, B., "Towards a two-dimensional programming environment", Readings in AI, Palo Alto, CA, Tioga, 1981, p.178.

[MEYJ72] Meyers, J., Fundamentals of Experimental Design, Allyn & Bacon, Inc., Boston, 1972.

[MYEG78] Myers, G.J., "A controlled experiment in program testing and code walk throughs / inspections", Communications of the ACM, 21, 9, Sept. 1978, pp 760-768.

[MYEJ72] Myers, J., Fundamentals of Experimental Design, Allyn and Bacon, Inc., Boston, 19872.

[MUNJ78] Munson, J.B., "Software Maintainability: A practical concern for life-cycle costs," IEEE Proceedings of the IEEE Computer Software and Applications Conference, Compsac 1978.

[NAKY78] Nakamure, Y., Miyahama, R., Takeuchi, H., "Complementary approach to the effective software development environment", Proceedings of the IEEE Conference on Computer Software and Applications, 1978.

[PARD71] Parnas, D.L. (1971), "Information Distribution Aspects of Design Methodology," Proc. IFIP Congress 71, Vol. 1, pp. 339-344, North-Holland Publishing Co.

[PARD72] Parnas, D., "On the Criteria To Be Used in Decomposing Systems into Modules," Communications of the ACM, Dec. 1972, Vol. 15, no. 12, 1053-1058.

[PARD74] Parnas, David (1974), "On A Buzzword: Hierarchical Structure," Proc. IFIP Congress, pp. 336-339.

[PARD79] Parnas, D.L., "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, Vol. SE-5, No. 2, March 1979, pp 128-138.

[PPI86] Productivity Products International, Inc., Objective-C Reference Manual, Sandy Hook, Connecticut, 1986.

[PRIJ82] Privitera, Dr. J.P., "Ada design language for the Structured Design Methodology", Proceedings of the AdaTEC Conference, October 1982, pp 76-90.

[RENT82] Rentsch, T., "Object Oriented Programming," SIGPLAN Notices, vol. 17, no. 9, p. 51, Sep. 1982.

[ROMH85] Rombach, H.D., "Impact of Software Structure on Maintenance," IEEE Transactionson Software Engineering 152-160.

[ROSD75] Ross, D.T., Goodenough, J.B. and Irvine, C.A. (1975), "Software Engineering: Process, Principles and Goals," IEEE Computer, Vol. 8, No. 5, pp. 17-27.

[ROSD77] Ross, D.T., Schoman, K.E. Jr., "Structured Analysis for Requirements Definition," IEEE Transactionson Software Engineering, Vol. SE-3, No. 1, January 1971, pp 69-84.

[SAAH77] Saal, H.J., and Weiss, Z., "An empirical study of APL programs", Computer Languages, 2, 3, 1977, pp 47-60.

[SHAM84] Shaw, M., "Abstraction Techniques in Modern Programming Languages," IEEE Software, Vol. 1, no. 4, 10-25 1984.

[SCHH85] Schaefer, H., "Metrics for Optimal Maintenance Management", IEEE 1985 p.114

[SELC87] Selig, C., "ADLIF: A Structured Design Language for Metrics Analysis", Master's Thesis, Virginia Tech, Blacksburg, Virginia, 1987.

[SCHN77] Schneidewind, N.F. and Hoffman, H.M., "An experiment in software error data collection and analysis", Proceedings of the Sixth Texas Conference on Computing Systems, November 15-14, 1977.

[SHNB76] Shneiderman, B., "Exploratory Experiments in Programmer Behavior," Int'l J. of Computer and Information Sci., Vol. 5, No. 2, 123-143 1976.

[SHNB77] Shneiderman, B., Mayer, R., McKay, D. and Heller, P., "Experimental investigations of the utility of detailed flowcharts in programming", Communications of the ACM, 20, 1977, pp 373-381.

[SHNB79] Shneiderman, B. and Mayer, R., "Syntactic / semantic interactions in programmer behavior: A model and experimental results", International Journal of Computer and Information Sciences, 7, 1979, pp 219-239.

[SHNB80] Shneiderman, B. Software Psychology, Little, Brown and Co., Boston, 1980.

[STAJ76] Stay, J.F.," HIPO and Integrated Program Design", IBM Systems Journal, IBM Corp, Vol. 15, No. 2, 1976, pp 143-154.

[STEW81] Stevens, W., Using Structured Design: How to Make Programs Simple, Changeable, Flexible, and Reusable, John Wiley & Sons, New York, New York, 1981.

[STEW74] Stevens, W.P., Myers, G.J., Constantine, L.L., "Structured Design", IBM Systems Journal, IBM Corp., 1974.

[SWAG78] Swann, G.H., Top-down Structured Design Techniques, PBI Inc., New York, 1978.

[SYNJ78] Synnott, J.B. III, "Managing Software Development--Requirements to Delivery," Proceedings of the IEEE Conference on Computer Software and Applications", 1978.

[TAUR77] Tausworth, R.C. (1977), Standardized Development of Computer Software, Prentice-Hall, Englewood Cliffs, N.J.

[TRAW79] Tracz, W., "Computer Programming and the Human Thought Process," Software-Practice and Experience, Vol. 9, 127-137, (1979).

[WOLR74] Wolverton, R.W., "The Cost of Developing Large-Scale Software," IEEE Transactions on Computers, Vol. C-23, No. 6, June 1974.

[YOUE74] Youngs, E. "Human Errors in Programming," Int. J. Man-Machine Studies (1974) 6, pp 361-376.

[YOUE82] Yourdon, E., Managing the System Life Cycle: A Software Development Methodology Overview, Yourdon Press, New York, New York, 1982.

# Appendix A. Rules of the Experiment for Subjects

## Task Instructions

For every task that you perform, you are required to keep track of the intervals during which you actually worked on the task. These intervals constitute an important part of the data collection and their accuracy is important. You will not be graded on the number of sizes of your intervals. Everyone has a different style, so don't worry about the numbers.

The time chart is used to indicate what you are working on (subtask), how much time you are spending on each sub task, the actual period of time for the subtask and the system performance during the sub-task. While it may appear to be a lot of bookkeeping at first, I'll show how it isn't really so bad.

There are only three possible subtasks, and they occur in order. The subtasks are:

1) Thinking about the task, before editing
2) Modifying code, before running with official data
3) Testing code with official data, before completed

For each of the subtasks, the clock time when the subtask began must be recorded. You can round the time value to the nearest quarter of an hour. A subtask may have any number of intervals during which you worked on that portion of the subtask. However, once the next subtask in sequence begins, the previous subtask is over and accumulates no more intervals. This is why these subtasks always occur in order. Each subtask includes its predecessors by default. Modifying code implies thinking about code. Testing code with data implies having made modifications to test and having thought about those modifcations.

Each of the three main subtasks can have a set of intervals during which you worked on the task. Only document intervals that are 15 minutes long or greater. It is also acceptable to round the interval boundaries to within a quarter of an hour. Of course, some intervals will be short and others will be long, but that depends upon the style of the person and is not graded.

For each interval specify what percent of the time you were actually occupied with the task. For example, if you start the Modification subtask at (about) 2:15 and work until 3:00, but were interrupted several times and got a phone call, you may decide that you really only worked for 60% of the listed time.

For those intervals which require access to the machine, at the end of the interval (when ever you logout) you need to record your perception of the

system performance. Use the following "frustrating to speedy" chart to deside how to rate the system.

1-----3-----5-----7-----9

frustating: speedy:

After recording your perception of the performance, you need to record the actual system performance. Do the "finger" command to get the system loading factor. It is on the third line of output and is the last real number on the row. Record that number on the time keeping sheet.

The following is now an example of a time keeping sheet properly filled out:

| Subtask | interval | %active | perc perf | sys perf |
|---|---|---|---|---|
| Thinking | 12/4 10:00 - 10:45 am | 70% | | |
| | 2:15 - 3:30 pm | 70% | | |
| | 12/5 1:00 - 3:00 pm | 30% | | |
| Modifying | 12/6 7:30 - 9:45 pm | 90% | 4.45 | .1 |
| | 12/7 1:00 - 1:00 pm | 80% | 1.95 | .2 |
| | 12/7 6:00 - 9:00 pm | 70% | 2.54 | .2 |
| Testing | 12/7 9:00 - 10:00 pm | 100% | 3.55 | .2 |
| | 11:00 pm - 1:00 am | 100% | 6.11 | .5 |

There are several things to note about this time sheet. First, thinking does not require recording of the system performance, since most thinking is done off-line. The thinking interval includes getting printouts and looking at the code, but not touching. As soon as editing begins, then the subtask becomes "modifying".

The date must be included in these timesheets, since some subtasks may require several days. Likewise, AM/PM information is must also be included.

The starting time of each subtask is equivalent to the time of the first interval. So it is really only necessary to denote which interval starts a subtask.

Ideally, an interval that is interrupted by more than 15 minutes should be come two separate intervals. However, this can also be accounted for in the percent activity rating.

1. Do each task one at a time. Do not attempt to work on more than one task at a time.

2. Once you've submitted a task as completed, stop working on it. If you need another day, take another day and submit it late. Within 1 or 2 extra weeks, there is no late penalty.

3. Do the tasks in the order received.

4. Follow all task questionnaire instructions. It is the completeness with which you fill out the questionnaire that determines your grade.

5. You are given one week to complete each task. However, the deadline is not fixed. You may turn in assignments late with no penalty. However, you must complete all tasks with sufficient time so that I can grade them. Failure to complete all tasks will be detrimental to your grade.

6. Do not ask other students for specific help with the assignment. If you have questions about the assignment, see me. I have lots of office hours or you can send mail or make an appointment.

7. Do not talk to the testing group. They can not give you any information about the assignment. You will be penalized if you pester them.

8. Keep accurate log information for working on the assignment. I don't care how much or how little time you spend, but you must be honest. These numbers are all anonymous and no one will know how you did personally.

9. Don't let the assignments pile up at the end. You must do all of the tasks. These tasks are designed to require thought, but not require many hours of work. If you think you are getting behind or think that you will be getting behind soon, see me. Maybe we can work something out.

10. Most importantly, these tasks have been designed for two purposes: To give you programming experience in a real-world environment and to give me data to do my thesis. No data that is collected will affect your grade in any way. However, if you fail to do all of the tasks, your grade will suffer.

**Task Details**

For each task you will get a new copy of the "account" application system from the appropriate directory. Students making the modifications to the 'C' version will retrieve all of the code from the directory

[cs498000.account.cversion].

There are *.c and *.h files in that directory. Everything in that directory is used to produce a main program "main".

Students making modifications to the 'Obj-C' version will retrieve all of the code from the directory:

[cs498000.account.objc]

There are *.m *.c and *.h files in that directory. Everything in that directory is used to produce a main program "main".

Both systems produce a main program out of the module named "main". Your executable file should, therefore, be named "main.exe". Your code and your executable image should have world read access. The professional testers will enter you directory, copy out your program for analysis and will also run your program as if it were in actual use. They have a huge set of test data that you cannot get to that they will use to test your program. If you make an effort at the task, you will receive credit. Something should work. However, if you make no effort and no changes are evident, the task will not be accepted as completed.

You will need to save the tasks where the testers can find them. It is not known when the testers will actually execute your code. Therefore you must save the code in a sub-directory. The name of the subdirectory will be [.taskNc] or [.taskNobjc]. Thus, when you begin working on the 'C' version of task 3 and you have account #"01" then you :

```
create/dir [cs498001.task3c]
```

When you do the objective 'c version of task 3, you:

```
create/dir [cs498001.task3objc]
```

Make sure that there is world read access of these files

```
set prot=(w:r) task3c.dir
set prot=(w:r) [.task3c]*.*
```

# Appendix B. Raw Data

## Student Data

| GROUPNM | SEQ | INTEGR | TESTX | LEVELNM | COURSES | SUBJTASK | SUBJQUES |
|---|---|---|---|---|---|---|---|
| A | 1 | 21 | 41 | SEN | 7 | 2.0 | 1.5 |
| A | 2 | 3 | 36 | SEN | 8 | 2.5 | 1.0 |
| A | 3 | 0 | 46 | JUN | 7 | 3.0 | 1.0 |
| A | 4 | 6 | 105 | SEN | 6 | 2.0 | 1.0 |
| A | 5 | 6 | 68 | JUN | 7 | 2.0 | 2.0 |
| A | 6 | 6 | 43 | SEN | 6 | 1.0 | 1.0 |
| A | 7 | 3 | 21 | SEN | 6 | 2.5 | 1.0 |
| A | 8 | 1 | 39 | SEN | 7 | 1.5 | 1.0 |
| A | 9 | 2 | 47 | SEN | 8 | 2.0 | 1.0 |
| A | 10 | 3 | 46 | SEN | 5 | 1.0 | 1.0 |
| B | 1 | 2 | 53 | SEN | 14 | 4.0 | 1.0 |
| B | 2 | 12 | 57 | JUN | 6 | 2.5 | 1.0 |
| B | 3 | 15 | 66 | SEN | 9 | 2.0 | 1.0 |
| B | 4 | 3 | 4 | SEN | 7 | 2.0 | 1.0 |
| B | 5 | 3 | 25 | SEN | 8 | 2.0 | 1.0 |
| B | 6 | 6 | 61 | SEN | 9 | 2.5 | 2.0 |
| B | 7 | 4 | 69 | SEN | 7 | 2.0 | 2.0 |
| B | 8 | 4 | 40 | SEN | 7 | 2.0 | 1.0 |
| B | 9 | 20 | 76 | SEN | 7 | 2.0 | 1.0 |
| B | 10 | 12 | 58 | SEN | 5 | 3.0 | 1.0 |

| GROUPNM | SEQ | QCA | CSQCA | CURRICNM | C | PASCAL | OBJC | SMALLT |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 3.30 | 3.30 | CS | 3 | 26 | 3 | 0 |
| A | 2 | 2.90 | 3.46 | CS | 3 | 21 | 3 | 0 |
| A | 3 | 2.40 | 3.50 | CS | 3 | 30 | 3 | 0 |
| A | 4 | 3.14 | 3.30 | CS | 4 | 48 | 3 | 0 |
| A | 5 | 2.40 | 3.00 | CS | 6 | 42 | 3 | 0 |
| A | 6 | 3.12 | 3.92 | CS | 3 | 15 | 3 | 0 |
| A | 7 | 3.50 | 3.60 | CS | 3 | 24 | 3 | 0 |
| A | 8 | 3.50 | 3.72 | CS | 4 | 27 | 3 | 0 |
| A | 9 | 3.30 | 3.60 | CS | 3 | 24 | 3 | 0 |
| A | 10 | 3.45 | 3.42 | CS | 15 | 21 | 3 | 0 |
| B | 1 | 2.37 | 2.40 | CS | 7 | 45 | 3 | 0 |
| B | 2 | 2.40 | 3.02 | CS | 3 | 43 | 3 | 0 |
| B | 3 | 2.50 | 2.80 | CS | 7 | 32 | 3 | 0 |
| B | 4 | 2.67 | 2.71 | CS | 3 | 25 | 3 | 1 |
| B | 5 | 3.43 | 3.54 | CS | 13 | 23 | 3 | 0 |
| B | 6 | 3.15 | 3.50 | CS | 1 | 30 | 3 | 0 |
| B | 7 | 2.40 | 2.60 | CS | 4 | 40 | 3 | 0 |
| B | 8 | 2.78 | 3.10 | CS | 3 | 30 | 3 | 0 |
| B | 9 | 3.98 | 3.93 | CS | 3 | 33 | 3 | 0 |
| B | 10 | 2.92 | 3.30 | CS | 6 | 31 | 3 | 0 |

---------------------------- GROUP=1 TASK=2 LANG=1 ----------------------------

| GROUPNM | TASK | LANGNAME | MODULES | SECTIONS | LINES | TOTLINES | CERR | TC |
|---|---|---|---|---|---|---|---|---|
| A | 2 | C | 2 | 6 | 65 | 64 | 6 | 22 |
| A | 2 | C | 2 | 13 | 249 | 67 | 3 | 8 |
| A | 2 | C | 2 | 18 | 219 | 74 | 30 | 67 |
| A | 2 | C | 3 | 5 | 43 | 41 | 2 | 12 |
| A | 2 | C | 5 | 8 | 34 | 33 | 2 | 6 |
| A | 2 | C | 3 | 6 | 54 | 54 | 5 | 146 |
| A | 2 | C | 2 | 5 | 79 | 78 | 3 | 10 |
| A | 2 | C | 2 | 5 | 84 | 83 | 2 | 6 |
| A | 2 | C | 3 | 6 | 65 | 63 | 2 | 9 |
| A | 2 | C | 3 | 8 | 69 | 59 | 2 | 9 |

---------------------------- GROUP=1 TASK=2 LANG=2 ----------------------------

| GROUPNM | TASK | LANGNAME | MODULES | SECTIONS | LINES | TOTLINES | CERR | TC |
|---|---|---|---|---|---|---|---|---|
| A | 2 | OBJC | 0 | 0 | 0 | 0 | 2 | 1 |
| A | 2 | OBJC | 2 | 4 | 30 | 30 | 4 | 13 |
| A | 2 | OBJC | 2 | 5 | 38 | 38 | 2 | 16 |
| A | 2 | OBJC | 1 | 4 | 17 | 17 | 2 | 5 |
| A | 2 | OBJC | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 2 | OBJC | 2 | 4 | 35 | 35 | 0 | 0 |
| A | 2 | OBJC | 2 | 5 | 40 | 38 | 3 | 4 |
| A | 2 | OBJC | 2 | 4 | 39 | 39 | 3 | 9 |
| A | 2 | OBJC | 3 | 6 | 27 | 26 | 1 | 1 |
| A | 2 | OBJC | 2 | 4 | 27 | 27 | 2 | 5 |

---------------------------- GROUP=1 TASK=3 LANG=1 ----------------------------

| GROUPNM | TASK | LANGNAME | MODULES | SECTIONS | LINES | TOTLINES | CERR | TC |
|---|---|---|---|---|---|---|---|---|
| A | 3 | C | 2 | 7 | 50 | 47 | 8 | 38 |
| A | 3 | C | 2 | 5 | 68 | 68 | 2 | 7 |
| A | 3 | C | 3 | 7 | 84 | 84 | 4 | 7 |
| A | 3 | C | 2 | 4 | 46 | 45 | 1 | 4 |
| A | 3 | C | 5 | 11 | 54 | 49 | 0 | 0 |
| A | 3 | C | 3 | 24 | 255 | 70 | 0 | 0 |
| A | 3 | C | 2 | 25 | 249 | 68 | 6 | 14 |
| A | 3 | C | 0 | 0 | 0 | 0 | 1 | 2 |
| A | 3 | C | 3 | 6 | 48 | 46 | 1 | 1 |
| A | 3 | C | 2 | 5 | 65 | 57 | 1 | 3 |

---------------------------- GROUP=1 TASK=2 LANG=1 ----------------------------

| GROUPNM | TASK | LANGNAME | LE | RE | LGE | TOTERR | STHIN | SMOD |
|---|---|---|---|---|---|---|---|---|
| A | 2 | C | 1 | 1 | 5 | 35 | 7 | 7.5 |
| A | 2 | C | 0 | 0 | 2 | 13 | 2 | 3.0 |
| A | 2 | C | 0 | 5 | 2 | 104 | 3 | 7.0 |
| A | 2 | C | 0 | 0 | 2 | 16 | 3 | 8.0 |
| A | 2 | C | 1 | 2 | 0 | 11 | 3 | 3.0 |
| A | 2 | C | 0 | 0 | 0 | 151 | 3 | 5.0 |
| A | 2 | C | 0 | 2 | 4 | 19 | 2 | 2.0 |
| A | 2 | C | 1 | 0 | 1 | 10 | 3 | 3.0 |
| A | 2 | C | 0 | 0 | 7 | 18 | 3 | 4.0 |
| A | 2 | C | 0 | 0 | 0 | 11 | 2 | 1.0 |

---------------------------- GROUP=1 TASK=2 LANG=2 ----------------------------

| GROUPNM | TASK | LANGNAME | LE | RE | LGE | TOTERR | STHIN | SMOD |
|---|---|---|---|---|---|---|---|---|
| A | 2 | OBJC | 0 | 1 | 2 | 6 | 2 | 3 |
| A | 2 | OBJC | 0 | 2 | 0 | 19 | 4 | 4 |
| A | 2 | OBJC | 0 | 4 | 4 | 26 | 1 | 3 |
| A | 2 | OBJC | 1 | 0 | 0 | 8 | 3 | 1 |
| A | 2 | OBJC | 1 | 5 | 0 | 6 | 3 | 4 |
| A | 2 | OBJC | 0 | 1 | 1 | 2 | 1 | 2 |
| A | 2 | OBJC | 1 | 0 | 6 | 14 | 3 | 3 |
| A | 2 | OBJC | 0 | 1 | 1 | 14 | 3 | 2 |
| A | 2 | OBJC | 1 | 3 | 0 | 6 | 2 | 3 |
| A | 2 | OBJC | 0 | 2 | 0 | 9 | 2 | 1 |

---------------------------- GROUP=1 TASK=3 LANG=1 ----------------------------

| GROUPNM | TASK | LANGNAME | LE | RE | LGE | TOTERR | STHIN | SMOD |
|---|---|---|---|---|---|---|---|---|
| A | 3 | C | 0 | 0 | 1 | 47 | 4 | 5 |
| A | 3 | C | 0 | 0 | 1 | 10 | 1 | 2 |
| A | 3 | C | 0 | 0 | 0 | 11 | 3 | 3 |
| A | 3 | C | 0 | 0 | 0 | 5 | 2 | 2 |
| A | 3 | C | 0 | 1 | 1 | 2 | 3 | 3 |
| A | 3 | C | 0 | 0 | 3 | 3 | 4 | 5 |
| A | 3 | C | 2 | 5 | 1 | 28 | 1 | 1 |
| A | 3 | C | 0 | 0 | 0 | 3 | 3 | 2 |
| A | 3 | C | 0 | 0 | 2 | 4 | 2 | 2 |
| A | 3 | C | 0 | 1 | 1 | 6 | 2 | 1 |

## Task Data

GROUP=1 TASK=2 LANG=1

| STEST | SALL | TTHINK | PTHINK | TMOD | PMOD | TTEST | PTEST | TASKTIME |
|---|---|---|---|---|---|---|---|---|
| 5 | 7 | 45 | 88 | 70 | 95 | 95 | 90 | 210 |
| 4 | 2 | 25 | 75 | 95 | 60 | 35 | 10 | 155 |
| 3 | 7 | 15 | 100 | 405 | 100 | 30 | 100 | 450 |
| 3 | 3 | 15 | 85 | 60 | 90 | 60 | 75 | 135 |
| 3 | 3 | 30 | 70 | 30 | 95 | 75 | 60 | 135 |
| 3 | 4 | 45 | 85 | 195 | 80 | 30 | 95 | 270 |
| 5 | 4 | 30 | 70 | 15 | 90 | 270 | 75 | 315 |
| 3 | 3 | 135 | 40 | 105 | 100 | 5 | 100 | 245 |
| 2 | 3 | 25 | 95 | 95 | 95 | 10 | 100 | 130 |
| 1 | 1 | 60 | 100 | 105 | 90 | 30 | 80 | 195 |

GROUP=1 TASK=2 LANG=2

| STEST | SALL | TTHINK | PTHINK | TMOD | PMOD | TTEST | PTEST | TASKTIME |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 25 | 90 | 70 | 92 | 5 | 95 | 100 |
| 2 | 3 | 25 | 90 | 45 | 95 | 40 | 95 | 110 |
| 3 | 3 | 15 | 80 | 30 | 100 | 30 | 60 | 75 |
| 1 | 2 | 15 | 100 | 45 | 100 | 30 | 100 | 90 |
| 4 | 3 | 15 | 80 | 45 | 70 | 45 | 30 | 105 |
| 2 | 2 | 15 | 100 | 45 | 95 | 60 | 60 | 120 |
| 6 | 5 | 60 | 70 | 80 | 70 | 240 | 77 | 380 |
| 1 | 2 | 150 | 60 | 70 | 100 | 5 | 100 | 225 |
| 2 | 3 | 20 | 100 | 100 | 90 | 15 | 100 | 135 |
| 1 | 1 | 45 | 80 | 60 | 90 | 15 | 100 | 120 |

GROUP=1 TASK=3 LANG=1

| STEST | SALL | TTHINK | PTHINK | TMOD | PMOD | TTEST | PTEST | TASKTIME |
|---|---|---|---|---|---|---|---|---|
| 3 | 4 | 30 | 80 | 155 | 77 | 10 | 60 | 195 |
| 0 | 0 | 5 | 80 | 45 | 50 | 15 | 90 | 65 |
| 3 | 3 | 30 | 100 | 60 | 70 | 15 | 90 | 105 |
| 2 | 2 | 15 | 100 | 55 | 90 | 20 | 100 | 90 |
| 5 | 3 | 15 | 90 | 75 | 92 | 15 | 50 | 105 |
| 5 | 4 | 15 | 100 | 75 | 90 | 90 | 80 | 180 |
| 3 | 2 | 120 | 70 | 30 | 80 | 200 | 75 | 350 |
| 1 | 2 | 40 | 100 | 55 | 100 | 5 | 100 | 100 |
| 2 | 2 | 25 | 95 | 50 | 100 | 10 | 100 | 85 |
| 3 | 2 | 75 | 75 | 150 | 66 | 15 | 100 | 240 |

----------------------------- GROUP=1 TASK=3 LANG=2 -----------------------------

| GROUPNM | TASK | LANGNAME | MODULES | SECTIONS | LINES | TOTLINES | CERR | TC |
|---|---|---|---|---|---|---|---|---|
| A | 3 | OBJC | 3 | 5 | 33 | 33 | 1 | 1 |
| A | 3 | OBJC | 2 | 4 | 52 | 51 | 1 | 3 |
| A | 3 | OBJC | 4 | 7 | 47 | 47 | 2 | 2 |
| A | 3 | OBJC | 1 | 4 | 22 | 22 | 2 | 9 |
| A | 3 | OBJC | 3 | 7 | 36 | 33 | 1 | 4 |
| A | 3 | OBJC | 2 | 4 | 35 | 35 | 0 | 0 |
| A | 3 | OBJC | 2 | 4 | 46 | 45 | 0 | 0 |
| A | 3 | OBJC | 0 | 0 | 0 | 0 | 1 | 1 |
| A | 3 | OBJC | 3 | 5 | 35 | 35 | 2 | 2 |
| A | 3 | OBJC | 2 | 4 | 30 | 26 | 2 | 4 |

----------------------------- GROUP=1 TASK=4 LANG=1 -----------------------------

| GROUPNM | TASK | LANGNAME | MODULES | SECTIONS | LINES | TOTLINES | CERR | TC |
|---|---|---|---|---|---|---|---|---|
| A | 4 | C | 2 | 7 | 66 | 65 | 3 | 49 |
| A | 4 | C | 2 | 5 | 75 | 75 | 0 | 0 |
| A | 4 | C | 3 | 7 | 60 | 60 | 9 | 19 |
| A | 4 | C | 2 | 4 | 65 | 65 | 1 | 1 |
| A | 4 | C | 4 | 10 | 60 | 55 | 2 | 16 |
| A | 4 | C | 2 | 23 | 248 | 64 | 2 | 3 |
| A | 4 | C | 2 | 23 | 246 | 67 | 3 | 15 |
| A | 4 | C | 2 | 6 | 72 | 72 | 1 | 1 |
| A | 4 | C | 2 | 23 | 253 | 64 | 1 | 1 |
| A | 4 | C | 2 | 8 | 97 | 70 | 1 | 7 |

----------------------------- GROUP=1 TASK=4 LANG=2 -----------------------------

| GROUPNM | TASK | LANGNAME | MODULES | SECTIONS | LINES | TOTLINES | CERR | TC |
|---|---|---|---|---|---|---|---|---|
| A | 4 | OBJC | 2 | 7 | 51 | 31 | 3 | 4 |
| A | 4 | OBJC | 2 | 4 | 62 | 62 | 1 | 5 |
| A | 4 | OBJC | 4 | 7 | 41 | 41 | 2 | 8 |
| A | 4 | OBJC | 1 | 5 | 51 | 50 | 1 | 8 |
| A | 4 | OBJC | 2 | 4 | 30 | 28 | 1 | 1 |
| A | 4 | OBJC | 2 | 4 | 35 | 35 | 1 | 1 |
| A | 4 | OBJC | 2 | 4 | 45 | 45 | 1 | 1 |
| A | 4 | OBJC | 2 | 4 | 45 | 45 | 1 | 3 |
| A | 4 | OBJC | 2 | 4 | 32 | 32 | 3 | 4 |
| A | 4 | OBJC | 2 | 4 | 51 | 47 | 1 | 3 |

------------------------- GROUP=1 TASK=3 LANG=2 -------------------------

| GROUPNM | TASK | LANGNAME | LE | RE | LGE | TOTERR | STHIN | SMOD |
|---|---|---|---|---|---|---|---|---|
| A | 3 | OBJC | 0 | 0 | 1 | 3 | 1 | 1.5 |
| A | 3 | OBJC | 0 | 0 | 6 | 10 | 2 | 3.0 |
| A | 3 | OBJC | 0 | 0 | 0 | 4 | 3 | 3.0 |
| A | 3 | OBJC | 0 | 1 | 1 | 13 | 2 | 1.0 |
| A | 3 | OBJC | 0 | 1 | 0 | 6 | 3 | 1.0 |
| A | 3 | OBJC | 0 | 0 | 0 | 0 | 2 | 2.0 |
| A | 3 | OBJC | 0 | 0 | 0 | 0 | 1 | 1.0 |
| A | 3 | OBJC | 0 | 1 | 0 | 3 | 2 | 2.0 |
| A | 3 | OBJC | 0 | 1 | 2 | 7 | 2 | 3.0 |
| A | 3 | OBJC | 0 | 1 | 0 | 7 | 3 | 2.0 |

------------------------- GROUP=1 TASK=4 LANG=1 -------------------------

| GROUPNM | TASK | LANGNAME | LE | RE | LGE | TOTERR | STHIN | SMOD |
|---|---|---|---|---|---|---|---|---|
| A | 4 | C | 0 | 0 | 4 | 56 | 2.5 | 2.5 |
| A | 4 | C | 0 | 0 | 2 | 2 | 2.0 | 2.0 |
| A | 4 | C | 0 | 1 | 1 | 30 | 5.0 | 7.0 |
| A | 4 | C | 0 | 3 | 1 | 6 | 3.0 | 5.0 |
| A | 4 | C | 0 | 0 | 3 | 21 | 4.0 | 3.0 |
| A | 4 | C | 1 | 2 | 0 | 8 | 3.0 | 4.0 |
| A | 4 | C | 0 | 0 | 1 | 19 | 2.0 | 2.0 |
| A | 4 | C | 0 | 1 | 8 | 11 | 3.0 | 4.0 |
| A | 4 | C | 0 | 0 | 0 | 2 | 1.0 | 2.0 |
| A | 4 | C | 0 | 0 | 1 | 9 | 1.0 | 2.0 |

------------------------- GROUP=1 TASK=4 LANG=2 -------------------------

| GROUPNM | TASK | LANGNAME | LE | RE | LGE | TOTERR | STHIN | SMOD |
|---|---|---|---|---|---|---|---|---|
| A | 4 | OBJC | 1 | 1 | 9 | 18 | 2.5 | 6 |
| A | 4 | OBJC | 0 | 1 | 2 | 9 | 1.0 | 2 |
| A | 4 | OBJC | 0 | 0 | 0 | 10 | 5.0 | 3 |
| A | 4 | OBJC | 0 | 0 | 2 | 11 | 3.0 | 2 |
| A | 4 | OBJC | 0 | 0 | 2 | 4 | 3.0 | 2 |
| A | 4 | OBJC | 0 | 0 | 1 | 3 | 3.0 | 3 |
| A | 4 | OBJC | 0 | 1 | 3 | 6 | 3.0 | 3 |
| A | 4 | OBJC | 0 | 0 | 7 | 11 | 2.0 | 3 |
| A | 4 | OBJC | 0 | 2 | 1 | 10 | 1.0 | 3 |
| A | 4 | OBJC | 0 | 1 | 3 | 8 | 1.0 | 3 |

GROUP=1 TASK=3 LANG=2

| STEST | SALL | TTHINK | PTHINK | TMOD | PMOD | TTEST | PTEST | TASKTIME |
|---|---|---|---|---|---|---|---|---|
| 1.5 | 1.5 | 15 | 95 | 45 | 95 | 25 | 97 | 85 |
| 2.0 | 2.0 | 10 | 90 | 40 | 95 | 80 | 95 | 130 |
| 3.0 | 3.0 | 15 | 100 | 30 | 95 | 15 | 100 | 60 |
| 4.0 | 3.0 | 5 | 100 | 50 | 100 | 75 | 80 | 130 |
| 4.0 | 3.0 | 15 | 50 | 60 | 90 | 30 | 75 | 105 |
| 2.0 | 2.0 | 30 | 85 | 30 | 95 | 30 | 95 | 90 |
| 1.0 | 1.0 | 40 | 80 | 30 | 100 | 90 | 90 | 160 |
| 1.0 | 2.0 | 30 | 100 | 55 | 100 | 5 | 100 | 90 |
| 1.0 | 2.0 | 15 | 100 | 100 | 90 | 10 | 100 | 125 |
| 1.0 | 2.0 | 60 | 70 | 120 | 95 | 15 | 100 | 195 |

GROUP=1 TASK=4 LANG=1

| STEST | SALL | TTHINK | PTHINK | TMOD | PMOD | TTEST | PTEST | TASKTIME |
|---|---|---|---|---|---|---|---|---|
| 2 | 2.5 | 30 | 95 | 85 | 90 | 100 | 97 | 215 |
| 2 | 2.0 | 5 | 100 | 40 | 90 | 25 | 100 | 70 |
| 5 | 5.0 | 30 | 100 | 45 | 90 | 30 | 100 | 105 |
| 3 | 4.0 | 15 | 100 | 50 | 100 | 75 | 100 | 140 |
| 5 | 4.0 | 15 | 80 | 60 | 85 | 15 | 50 | 90 |
| 6 | 5.0 | 30 | 95 | 120 | 90 | 60 | 90 | 210 |
| 2 | 3.0 | 30 | 80 | 30 | 100 | 105 | 50 | 165 |
| 1 | 2.0 | 60 | 80 | 145 | 97 | 5 | 100 | 210 |
| 1 | 2.0 | 30 | 97 | 50 | 90 | 15 | 100 | 95 |
| 1 | 1.0 | 30 | 100 | 45 | 70 | 15 | 100 | 90 |

GROUP=1 TASK=4 LANG=2

| STEST | SALL | TTHINK | PTHINK | TMOD | PMOD | TTEST | PTEST | TASKTIME |
|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 15 | 80 | 255 | 88 | 20 | 10 | 290 |
| 3 | 2 | 5 | 80 | 55 | 87 | 80 | 25 | 140 |
| 5 | 3 | 15 | 100 | 30 | 100 | 15 | 100 | 60 |
| 2 | 2 | 15 | 100 | 25 | 100 | 80 | 60 | 120 |
| 5 | 3 | 15 | 90 | 60 | 65 | 15 | 80 | 90 |
| 3 | 3 | 15 | 90 | 45 | 85 | 30 | 95 | 90 |
| 4 | 3 | 30 | 70 | 30 | 70 | 180 | 50 | 240 |
| 1 | 2 | 30 | 75 | 145 | 100 | 5 | 100 | 180 |
| 2 | 2 | 25 | 100 | 95 | 92 | 10 | 100 | 130 |
| 1 | 2 | 60 | 30 | 135 | 50 | 15 | 100 | 210 |

---------------------------- GROUP=2 TASK=2 LANG=1 ----------------------------

| GROUPNM | TASK | LANGNAME | MODULES | SECTIONS | LINES | TOTLINES | CERR | TC |
|---|---|---|---|---|---|---|---|---|
| B | 2 | C | 2 | 4 | 45 | 45 | 0 | 0 |
| B | 2 | C | 2 | 5 | 80 | 76 | 3 | 9 |
| B | 2 | C | 2 | 5 | 50 | 50 | 4 | 7 |
| B | 2 | C | 2 | 7 | 106 | 79 | 4 | 16 |
| B | 2 | C | 2 | 4 | 54 | 53 | 1 | 1 |
| B | 2 | C | 2 | 12 | 214 | 53 | 5 | 25 |
| B | 2 | C | 2 | 15 | 253 | 71 | 4 | 13 |
| B | 2 | C | 3 | 6 | 30 | 29 | 2 | 9 |
| B | 2 | C | 2 | 5 | 67 | 66 | 1 | 1 |
| B | 2 | C | 2 | 45 | 519 | 283 | 2 | 8 |

---------------------------- GROUP=2 TASK=2 LANG=2 ----------------------------

| GROUPNM | TASK | LANGNAME | MODULES | SECTIONS | LINES | TOTLINES | CERR | TC |
|---|---|---|---|---|---|---|---|---|
| B | 2 | OBJC | 3 | 5 | 15 | 14 | 2 | 2 |
| B | 2 | OBJC | 2 | 5 | 43 | 40 | 4 | 9 |
| B | 2 | OBJC | 3 | 8 | 35 | 31 | 9 | 15 |
| B | 2 | OBJC | 3 | 5 | 31 | 30 | 5 | 8 |
| B | 2 | OBJC | 1 | 4 | 16 | 16 | 1 | 3 |
| B | 2 | OBJC | 3 | 5 | 30 | 28 | 10 | 30 |
| B | 2 | OBJC | 2 | 6 | 33 | 25 | 4 | 7 |
| B | 2 | OBJC | 0 | 0 | 0 | 0 | 3 | 7 |
| B | 2 | OBJC | 2 | 4 | 25 | 25 | 2 | 12 |
| B | 2 | OBJC | 1 | 18 | 93 | 18 | 2 | 9 |

---------------------------- GROUP=2 TASK=3 LANG=1 ----------------------------

| GROUPNM | TASK | LANGNAME | MODULES | SECTIONS | LINES | TOTLINES | CERR | TC |
|---|---|---|---|---|---|---|---|---|
| B | 3 | C | 2 | 5 | 36 | 36 | 1 | 3 |
| B | 3 | C | 2 | 6 | 66 | 65 | 4 | 14 |
| B | 3 | C | 2 | 6 | 73 | 72 | 2 | 20 |
| B | 3 | C | 2 | 5 | 67 | 67 | 0 | 0 |
| B | 3 | C | 2 | 4 | 59 | 59 | 0 | 0 |
| B | 3 | C | 2 | 5 | 53 | 53 | 4 | 21 |
| B | 3 | C | 3 | 17 | 183 | 76 | 4 | 32 |
| B | 3 | C | 2 | 13 | 146 | 57 | 1 | 3 |
| B | 3 | C | 2 | 5 | 57 | 57 | 2 | 12 |
| B | 3 | C | 2 | 10 | 73 | 56 | 2 | 5 |

GROUP=2 TASK=2 LANG=1

| GROUPNM | TASK | LANGNAME | LE | RE | LGE | TOTERR | STHIN | SMOD |
|---|---|---|---|---|---|---|---|---|
| B | 2 | C | 0 | 0 | 0 | 0 | 1 | 9 |
| B | 2 | C | 0 | 1 | 4 | 17 | 1 | 6 |
| B | 2 | C | 0 | 0 | 3 | 14 | 4 | 4 |
| B | 2 | C | 1 | 0 | 1 | 22 | 5 | 3 |
| B | 2 | C | 0 | 0 | 4 | 6 | 5 | 6 |
| B | 2 | C | 0 | 12 | 0 | 42 | 3 | 4 |
| B | 2 | C | 0 | 3 | 3 | 23 | 5 | 5 |
| B | 2 | C | 1 | 0 | 0 | 12 | 6 | 3 |
| B | 2 | C | 0 | 0 | 0 | 2 | 1 | 5 |
| B | 2 | C | 0 | 0 | 0 | 10 | 3 | 3 |

GROUP=2 TASK=2 LANG=2

| GROUPNM | TASK | LANGNAME | LE | RE | LGE | TOTERR | STHIN | SMOD |
|---|---|---|---|---|---|---|---|---|
| B | 2 | OBJC | 1 | 0 | 4 | 9 | 5 | 7 |
| B | 2 | OBJC | 0 | 2 | 1 | 16 | 6 | 3 |
| B | 2 | OBJC | 0 | 2 | 3 | 29 | 3 | 3 |
| B | 2 | OBJC | 0 | 2 | 2 | 17 | 5 | 5 |
| B | 2 | OBJC | 1 | 2 | 0 | 7 | 4 | 6 |
| B | 2 | OBJC | 0 | 2 | 0 | 42 | 5 | 5 |
| B | 2 | OBJC | 0 | 4 | 3 | 18 | 5 | 5 |
| B | 2 | OBJC | 1 | 2 | 2 | 15 | 6 | 5 |
| B | 2 | OBJC | 0 | 4 | 1 | 19 | 5 | 5 |
| B | 2 | OBJC | 0 | 7 | 0 | 18 | 5 | 7 |

GROUP=2 TASK=3 LANG=1

| GROUPNM | TASK | LANGNAME | LE | RE | LGE | TOTERR | STHIN | SMOD |
|---|---|---|---|---|---|---|---|---|
| B | 3 | C | 1 | 0 | 1 | 6 | 1 | 1 |
| B | 3 | C | 2 | 0 | 7 | 27 | 1 | 6 |
| B | 3 | C | 0 | 2 | 2 | 26 | 3 | 4 |
| B | 3 | C | 1 | 0 | 2 | 3 | 3 | 1 |
| B | 3 | C | 0 | 0 | 0 | 0 | 4 | 6 |
| B | 3 | C | 0 | 3 | 2 | 30 | 3 | 3 |
| B | 3 | C | 0 | 0 | 0 | 36 | 3 | 3 |
| B | 3 | C | 0 | 0 | 0 | 4 | 8 | 6 |
| B | 3 | C | 0 | 0 | 1 | 15 | 4 | 6 |
| B | 3 | C | 0 | 0 | 0 | 7 | 3 | 3 |

GROUP=2 TASK=2 LANG=1

| STEST | SALL | TTHINK | PTHINK | TMOD | PMOD | TTEST | PTEST | TASKTIME |
|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 45 | 75 | 120 | 90 | 5 | 90 | 170 |
| 1 | 4 | 15 | 100 | 75 | 100 | 15 | 100 | 105 |
| 4 | 4 | 60 | 100 | 110 | 100 | 45 | 100 | 215 |
| 5 | 5 | 60 | 100 | 45 | 100 | 30 | 100 | 135 |
| 2 | 5 | 45 | 100 | 75 | 80 | 15 | 40 | 135 |
| 9 | 6 | 60 | 100 | 30 | 100 | 170 | 92 | 260 |
| 7 | 5 | 30 | 70 | 50 | 40 | 70 | 90 | 150 |
| 8 | 6 | 15 | 95 | 45 | 90 | 30 | 80 | 90 |
| 1 | 3 | 10 | 100 | 35 | 100 | 5 | 100 | 50 |
| 3 | 3 | 30 | 100 | 45 | 90 | 15 | 100 | 90 |

GROUP=2 TASK=2 LANG=2

| STEST | SALL | TTHINK | PTHINK | TMOD | PMOD | TTEST | PTEST | TASKTIME |
|---|---|---|---|---|---|---|---|---|
| 3.0 | 5.0 | 135 | 70 | 180 | 85 | 15 | 90 | 330 |
| 1.0 | 5.0 | 75 | 90 | 120 | 95 | 15 | 90 | 210 |
| 3.0 | 3.0 | 90 | 100 | 120 | 100 | 60 | 100 | 270 |
| 7.0 | 5.0 | 45 | 75 | 210 | 92 | 30 | 100 | 285 |
| 2.0 | 4.0 | 45 | 55 | 120 | 78 | 15 | 50 | 180 |
| 3.5 | 4.3 | 20 | 100 | 30 | 100 | 30 | 100 | 80 |
| 7.0 | 5.0 | 15 | 70 | 115 | 55 | 120 | 80 | 250 |
| 4.5 | 5.0 | 35 | 55 | 90 | 70 | 20 | 90 | 145 |
| 1.0 | 3.0 | 105 | 50 | 35 | 30 | 85 | 80 | 225 |
| 7.0 | 7.0 | 150 | 100 | 30 | 95 | 165 | 100 | 345 |

GROUP=2 TASK=3 LANG=1

| STEST | SALL | TTHINK | PTHINK | TMOD | PMOD | TTEST | PTEST | TASKTIME |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 15 | 75 | 75 | 72 | 15 | 90 | 105 |
| 3 | 5 | 30 | 70 | 45 | 400 | 135 | 100 | 210 |
| 4 | 4 | 60 | 100 | 135 | 100 | 45 | 100 | 240 |
| 3 | 3 | 60 | 80 | 45 | 100 | 30 | 70 | 135 |
| 3 | 5 | 15 | 100 | 30 | 100 | 15 | 20 | 60 |
| 3 | 3 | 10 | 100 | 60 | 100 | 70 | 95 | 140 |
| 7 | 3 | 5 | 80 | 40 | 70 | 15 | 100 | 60 |
| 4 | 6 | 15 | 80 | 30 | 90 | 15 | 95 | 60 |
| 7 | 5 | 15 | 100 | 50 | 50 | 40 | 75 | 105 |
| 5 | 3 | 15 | 95 | 45 | 100 | 60 | 87 | 120 |

GROUP=2 TASK=3 LANG=2

| GROUPNM | TASK | LANGNAME | MODULES | SECTIONS | LINES | TOTLINES | CERR | TC |
|---|---|---|---|---|---|---|---|---|
| B | 3 | OBJC | 2 | 4 | 14 | 13 | 6 | 12 |
| B | 3 | OBJC | 2 | 4 | 57 | 54 | 1 | 2 |
| B | 3 | OBJC | 2 | 4 | 37 | 35 | 2 | 1 |
| B | 3 | OBJC | 2 | 5 | 50 | 46 | 2 | 3 |
| B | 3 | OBJC | 1 | 4 | 20 | 20 | 1 | 1 |
| B | 3 | OBJC | 2 | 6 | 37 | 32 | 5 | 25 |
| B | 3 | OBJC | 2 | 6 | 28 | 28 | 2 | 2 |
| B | 3 | OBJC | 2 | 4 | 33 | 32 | 1 | 5 |
| B | 3 | OBJC | 1 | 4 | 30 | 30 | 1 | 7 |
| B | 3 | OBJC | 2 | 20 | 114 | 32 | 0 | 0 |

GROUP=2 TASK=4 LANG=1

| GROUPNM | TASK | LANGNAME | MODULES | SECTIONS | LINES | TOTLINES | CERR | TC |
|---|---|---|---|---|---|---|---|---|
| B | 4 | C | 2 | 5 | 42 | 42 | 3 | 1 |
| B | 4 | C | 2 | 5 | 64 | 64 | 1 | 3 |
| B | 4 | C | 2 | 5 | 60 | 58 | 1 | 5 |
| B | 4 | C | 2 | 5 | 67 | 66 | 0 | 0 |
| B | 4 | C | 2 | 4 | 48 | 48 | 0 | 0 |
| B | 4 | C | 2 | 5 | 60 | 59 | 2 | 3 |
| B | 4 | C | 2 | 5 | 86 | 86 | 2 | 8 |
| B | 4 | C | 2 | 5 | 59 | 59 | 0 | 0 |
| B | 4 | C | 2 | 4 | 35 | 36 | 1 | 1 |
| B | 4 | C | 2 | 7 | 62 | 46 | 1 | 8 |

GROUP=2 TASK=4 LANG=2

| GROUPNM | TASK | LANGNAME | MODULES | SECTIONS | LINES | TOTLINES | CERR | TC |
|---|---|---|---|---|---|---|---|---|
| B | 4 | OBJC | 2 | 4 | 22 | 22 | 10 | 6 |
| B | 4 | OBJC | 3 | 6 | 59 | 54 | 6 | 5 |
| B | 4 | OBJC | 2 | 4 | 37 | 36 | 2 | 6 |
| B | 4 | OBJC | 1 | 4 | 54 | 52 | 1 | 0 |
| B | 4 | OBJC | 1 | 4 | 27 | 27 | 1 | 1 |
| B | 4 | OBJC | 2 | 4 | 32 | 31 | 3 | 6 |
| B | 4 | OBJC | 2 | 6 | 64 | 62 | 1 | 1 |
| B | 4 | OBJC | 1 | 4 | 44 | 44 | 1 | 8 |
| B | 4 | OBJC | 1 | 5 | 25 | 24 | 0 | 0 |
| B | 4 | OBJC | 2 | 20 | 125 | 43 | 0 | 0 |

GROUP=2 TASK=3 LANG=2

| GROUPNM | TASK | LANGNAME | LE | RE | LGE | TOTERR | STHIN | SMOD |
|---|---|---|---|---|---|---|---|---|
| B | 3 | OBJC | 0 | 0 | 1 | 19 | 1 | 5 |
| B | 3 | OBJC | 0 | 2 | 1 | 6 | 4 | 3 |
| B | 3 | OBJC | 0 | 0 | 0 | 3 | 4 | 3 |
| B | 3 | OBJC | 2 | 4 | 0 | 11 | 7 | 7 |
| B | 3 | OBJC | 0 | 0 | 0 | 2 | 3 | 4 |
| B | 3 | OBJC | 0 | 5 | 2 | 37 | 4 | 5 |
| B | 3 | OBJC | 0 | 3 | 1 | 8 | 5 | 3 |
| B | 3 | OBJC | 0 | 0 | 1 | 7 | 5 | 6 |
| B | 3 | OBJC | 0 | 0 | 1 | 9 | 4 | 5 |
| B | 3 | OBJC | 0 | 1 | 0 | 1 | 3 | 3 |

GROUP=2 TASK=4 LANG=1

| GROUPNM | TASK | LANGNAME | LE | RE | LGE | TOTERR | STHIN | SMOD |
|---|---|---|---|---|---|---|---|---|
| B | 4 | C | 1 | 0 | 1 | 6 | 1 | 7 |
| B | 4 | C | 0 | 0 | 0 | 4 | 1 | 1 |
| B | 4 | C | 0 | 0 | 2 | 8 | 3 | 4 |
| B | 4 | C | 0 | 0 | 2 | 2 | 5 | 5 |
| B | 4 | C | 0 | 0 | 0 | 0 | 5 | 5 |
| B | 4 | C | 0 | 0 | 0 | 5 | 5 | 3 |
| B | 4 | C | 0 | 0 | 2 | 12 | 5 | 5 |
| B | 4 | C | 0 | 0 | 1 | 1 | 3 | 4 |
| B | 4 | C | 0 | 1 | 0 | 3 | 3 | 5 |
| B | 4 | C | 0 | 0 | 1 | 10 | 0 | 3 |

GROUP=2 TASK=4 LANG=2

| GROUPNM | TASK | LANGNAME | LE | RE | LGE | TOTERR | STHIN | SMOD |
|---|---|---|---|---|---|---|---|---|
| B | 4 | OBJC | 1 | 2 | 2 | 21 | 5.0 | 5.0 |
| B | 4 | OBJC | 0 | 4 | 6 | 21 | 2.0 | 4.0 |
| B | 4 | OBJC | 0 | 1 | 1 | 10 | 3.0 | 3.0 |
| B | 4 | OBJC | 1 | 2 | 1 | 5 | 5.0 | 5.0 |
| B | 4 | OBJC | 0 | 0 | 0 | 2 | 4.0 | 5.0 |
| B | 4 | OBJC | 0 | 6 | 0 | 15 | 4.5 | 4.5 |
| B | 4 | OBJC | 1 | 3 | 1 | 7 | 5.0 | 5.0 |
| B | 4 | OBJC | 0 | 2 | 5 | 16 | 5.0 | 9.0 |
| B | 4 | OBJC | 0 | 0 | 2 | 2 | 3.0 | 3.0 |
| B | 4 | OBJC | 0 | 1 | 2 | 3 | 3.0 | 3.0 |

GROUP=2 TASK=3 LANG=2

| STEST | SALL | TTHINK | PTHINK | TMOD | PMOD | TTEST | PTEST | TASKTIME |
|---|---|---|---|---|---|---|---|---|
| 1 | 1.0 | 15 | 95 | 15 | 60 | 15 | 90 | 45 |
| 3 | 3.0 | 105 | 85 | 225 | 85 | 30 | 100 | 360 |
| 2 | 3.0 | 30 | 100 | 25 | 100 | 10 | 100 | 65 |
| 5 | 7.0 | 60 | 60 | 235 | 30 | 45 | 80 | 340 |
| 2 | 3.0 | 30 | 10 | 45 | 80 | 15 | 20 | 90 |
| 5 | 4.5 | 30 | 90 | 50 | 100 | 130 | 80 | 210 |
| 7 | 3.0 | 20 | 80 | 45 | 40 | 75 | 90 | 140 |
| 6 | 6.0 | 15 | 90 | 45 | 95 | 15 | 95 | 75 |
| 7 | 4.0 | 45 | 10 | 75 | 78 | 30 | 100 | 150 |
| 5 | 3.0 | 15 | 100 | 30 | 100 | 30 | 100 | 75 |

GROUP=2 TASK=4 LANG=1

| STEST | SALL | TTHINK | PTHINK | TMOD | PMOD | TTEST | PTEST | TASKTIME |
|---|---|---|---|---|---|---|---|---|
| 3 | 3 | 15 | 60 | 90 | 100 | 15 | 100 | 120 |
| 1 | 1 | 15 | 100 | 15 | 100 | 15 | 100 | 45 |
| 3 | 3 | 45 | 100 | 55 | 100 | 30 | 100 | 130 |
| 3 | 5 | 20 | 30 | 40 | 98 | 15 | 100 | 75 |
| 2 | 4 | 45 | 70 | 30 | 90 | 15 | 20 | 90 |
| 3 | 4 | 30 | 100 | 30 | 100 | 25 | 100 | 85 |
| 5 | 3 | 30 | 85 | 30 | 60 | 45 | 80 | 105 |
| 3 | 4 | 15 | 70 | 45 | 60 | 15 | 50 | 75 |
| 5 | 4 | 10 | 100 | 30 | 100 | 15 | 100 | 55 |
| 5 | 3 | 15 | 100 | 60 | 100 | 45 | 100 | 120 |

GROUP=2 TASK=4 LANG=2

| STEST | SALL | TTHINK | PTHINK | TMOD | PMOD | TTEST | PTEST | TASKTIME |
|---|---|---|---|---|---|---|---|---|
| 5.0 | 5.0 | 15 | 90 | 120 | 75 | 75 | 82 | 210 |
| 2.0 | 4.0 | 15 | 100 | 75 | 93 | 105 | 80 | 195 |
| 3.0 | 3.0 | 15 | 100 | 35 | 100 | 25 | 100 | 75 |
| 5.0 | 5.0 | 30 | 100 | 60 | 95 | 20 | 100 | 110 |
| 3.0 | 4.0 | 30 | 80 | 60 | 70 | 15 | 40 | 105 |
| 4.5 | 4.5 | 30 | 100 | 25 | 100 | 55 | 100 | 110 |
| 7.0 | 5.0 | 30 | 54 | 30 | 80 | 15 | 100 | 75 |
| 4.0 | 7.0 | 15 | 95 | 165 | 60 | 15 | 90 | 195 |
| 3.0 | 3.0 | 20 | 50 | 25 | 50 | 60 | 100 | 105 |
| 5.0 | 3.0 | 15 | 100 | 30 | 100 | 45 | 100 | 90 |

# Appendix C. Questionnaires

Background Questionnaire 12/7/87

Account number :

## I. Academic experience

For how many months have you formally studied computer science in a high school, technical school or college setting. Only count time actively spent; don't include summers where you worked as a lifeguard:

List obligations you are holding this quarter and list the approximate amount of time you expect to spend on each obligation each week. Include time spent in class. List courses first and include this course. Obligations include activies like: each course, club/social organizations, non-Tech courses, work. Omit free time and leisure-time activities.

| | Activity | hours per week |
|---|---|---|
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |
| 9 | | |
| 10 | | |
| 11 | | |
| 12 | | |
| 13 | | |
| 14 | | |
| 15 | | |

What is your overall QCA:
What is your QCA computed using only CS courses:
What is your overall QCA for just the past two years (6 quarters):
What is your CS QCA for just the past two years (6 quarters):

What is your general field of study (major):
What is your particular field of study (interest):

## II. Professional experience

For each of the following activites, list as accurately as possible the number of full time **months** you have engaged in the listed activity, both for personal use and professionally in the appropriate columns. Part-time jobs count, but ratio them appropriately. "For personal use" means designing and implementing for any reason including class work and professional work. Professional programming means designing and implementing exclusively for money, either by wage or by contract. Home-brewing software for sale is not considered part of professional programming.

Do not count intervening time. For example: if you took a course in FORTRAN 3 years ago, but havn't programmed in FORTRAN since, then only put down 3 months (1 quarter) under "academic and personal use". Don't worry if many categories are zero. No one is expected to have done all of these for a lifetime.

| | academic and personal tasks | professional tasks |
|---|---|---|
| Programming | | |
| in FORTRAN | | |
| in COBOL | | |
| in C | | |
| in Pascal | | |
| in Lisp | | |
| in Assembly language | | |
| in BASIC | | |
| in Ada | | |
| in Objective-C | | |
| in Smalltalk | | |
| Discussing requirements with end-users: | | |
| Discussing requirements with managers: | | |
| Designing new functionality from specs: | | |
| Coding new functionality from designs: | | |
| Integrating new features to someone else's code: | | |
| Writing external documentation: | | |
| Writing in-line documentation: | | |
| Testing your own code: | | |
| Testing someone else's code: | | |
| Fixing bugs in your own code: | | |
| Fixing bugs in someone else's code: | | |
| Using the VMS operating system: | | |

## III. Programming methodology

The following pertain to academic and professional programming and system design tasks. For each category, mark the frequency of behavior that fits your style best. Please be honest with yourself. Don't mark how you'd like to be. Mark how you really are. You're not being graded on this:

### A. Design behavior

(always)(often)(50%)(rarely)(never) draw structure charts before coding.
(always)(often)(50%)(rarely)(never) draw dataflow diagrams before coding.
(always)(often)(50%)(rarely)(never) draw data structures before coding.
(always)(often)(50%)(rarely)(never) draw flowcharts before coding.
(always)(often)(50%)(rarely)(never) define all subroutine interfaces before coding.

(always)(often)(50%)(rarely)(never) design top-down.
(always)(often)(50%)(rarely)(never) design bottom-up.
(always)(often)(50%)(rarely)(never) design middle-out:

Describe any other design methods that you use:

### B. Implementation behavior

(always)(often)(50%)(rarely)(never) build and discard prototypes before coding.
(always)(often)(50%)(rarely)(never) build product from specs exactly once.
(always)(often)(50%)(rarely)(never) build the product via evolution of code.

(always)(often)(50%)(rarely)(never) implement top-down.
(always)(often)(50%)(rarely)(never) implement bottom-up.
(always)(often)(50%)(rarely)(never) implement middle-out.

(always)(often)(50%)(rarely)(never) document before coding.
(always)(often)(50%)(rarely)(never) document while coding.
(always)(often)(50%)(rarely)(never) document after coding.

Describe any other implementation methods that you use:

**Task Questionaire**

Account number date:
Team letter Task number C or ObjC

**Work progress**

Number of faulty compiles (One for each compile that produces an error message):

Total number of compile errors (sum of those listed at end of compile)

Number of link errors (One for each link that fails):

Number of runtime errors (One for each each run that BOMBS):

Total number of logic errors (One for each run that produces an incorrect answer):

Rate the difficulty of the thinking portion of this task on this scale
1-----3-----5-----7-----9
easy hard

Rate the difficulty of the modification portion of this task on this scale
1-----3-----5-----7-----9
easy hard

Rate the difficulty of the testing portion of this task on this scale
1-----3-----5-----7-----9
easy hard

Rate the difficulty of the whole task on the following scale:
1-----3-----5-----7-----9
easy hard

| Subtask | interval | %active | perc perf | sys perf |
|---|---|---|---|---|
| Thinking | | | | |
| Finished | | | | |

**Post Class Questionnaire** Account no. ___________

Mark the appropriate response and add any comments you feel strongly about.

1. Did you think the tasks in general were: 1---2---3---4---5
   hard easy

2. Did you think the task questionnaires were: 1---2---3---4---5
   hard easy

3. What have you gained from these exercises?
   (wasted time, nothing, experience, knowledge)

4. What are you comments on the class. Did you like it or hate it, what did you get out of it:

5. What would you do to make the class better, both the first part and the second part.

Do you want a copy of the results?
If so, specify the address they can be mailed to.

# Appendix D. Task Specifications

## Warm-up Task

Task 1

The users of the "account" system have decided that while they like being able to move the current item highlight bar up and down over the list, they also want the ability to set the bar on a particular line. Since the line numbers are indicated on the screen, it seems that all they should have to do is type the command "goto" or "g" followed by the line number they want to go to.

For this task, you are guaranteed that the user entering the "goto" command will follow the command with an integer number. It is possible that the number is not one of the line numbers on the screen. If this is the case, then an error message should be issued.

Don't forget to keep track of how much time you spend on the task. We'll meet again on Tuesday to discuss how the assignment went.

**Task 2**

The "account" users have decided that they need to sort **lists** based on the text field of the list (the "name" of the list). This new facility should appear in the list menu as "organize" activated by letter 'o'. It should cause the items of the list on the screen to be sorted in ascending order by their text content:

text line entries are sorted by the text itself
ledgers are sorted according to the name of the ledger
sub-lists are sorted according to the name of the sub-list

For example, assume the screen contains the following list:

1 This is some text line
2 Christmas-ledger [food] $3421
3 dance-partners-list

After giving the 'o' command for organize the list will be:

1 Christmas-ledger [food] $3421
2 This is some text lnie
3 dance-partners-list

Note that the dance-partners still comes last. This is because the items are sorted by the ASCII collating sequence (strcmp) exactly as they appear . This means that lower case letters come after upper case letters. There is no problem with this, it is automatically done. Also note that the change is permanent to the list. It's not just a change to the screen. The current item in the list stays the same, which means that if the 3rd item were highlighted before the sorting, then the 3rd item will still be highlighted, even if it is a different item.

## Task 3

The "account" users have decided that they need to sort ledgers based on the value amount of moneyed items. This new facility should appear in the list menu as "organize" activated by letter 'o'. It should cause the items of the ledger on the screen to be sorted in descending order by their dollar amount.

comment and other "no money" entries are considered to have "no value" and should appear last in the list in any order
ledgers are sorted according to the summation amount of the ledger
purchase entries are sorted according to the amount of the purchase.

For example, assume the screen contains the following ledger:

1 This is some text line
2 Christmas-ledger [food] $21
3 Fruit-cake [food] $123

After giving the 'o' command for organize the ledger will be:

1 Fruit-cake [food] $123
2 Christmas-ledger [food] $21
3 This is some text line

Note that the comment comes last. This is because the comment is not a moneyed item and is there for considered to have "zero money". Also note that the change is permanent to the ledger. It's not just a change to the screen. The current item in the ledger stays the same, which means that if the 3rd item were highlighted before the sorting, then the 3rd item will still be highlighted, even if it is a different item.

## Task 4

The users have decided on another change. Within **ledgers**, they find it necessary to be able to jump from the current item to the next item of greater or equal value, skipping over comments. This new command called 'j' for "jump" should test the value of the current entry. If the current entry does not exist (empty ledger) or the current entry is not a moneyed item, then an error message should be issued. Otherwise, a new current item past the current item whose value is greater than or equal to the value of the current item should be found. If no such entry exists, then an error message should be issued, indicating that there are no greater items. Otherwise, that new item should become the current item.

For example, if the ledger contains the following where line 2 is the current entry:

```
1    A useless comment
2    christmas-ledger [food] $341
3    another comment
4    candy-for-mom [candy] $21
5    martini-lunch [drinks] $412
```

And the user enters the "jump" command, then the ledger should become as follows:

```
1    A useless comment
2    christmas-ledger [food] $341
3    another comment
4    candy-for-mom [candy] $21
5    martini-lunch [drinks] $412
```

The last line is made the new current entry because it contains an money amount that is greater than or equal to the amount in line 2, which had been the current line.

# Appendix E. Program Specification

**Automated Toy Company**
**Software specification**

We at Action Toys, Inc have decided to upgrade our present secretarial and accounting system by adding a computer system. After a thorough analyis, we've decided exactly what we think this system should do. It should be easy to use, well documented and maintainable. Unfortunately, we won't know if we like it or not until we see it. Therefore, if it does not meet our needs, we reserve the right to change the specification until the product becomes satisfactory.

Editing a list

The user primarily manages a list of items on the screen.. This list uses one screen row for each item, with as many items as will conveniently fit on the screen. It is not necessary to have multiple screens or to scroll the screen. Twenty rows is a reasonable number of items for the list. If there is a current item in the list, it must be highlighted. There can only be no current item if the list is empty.

A list is composed of items. Each item may be one of the following:

a) A text line, about 70 characters worth
b) An account ledger (the name of the ledger)
c) A sub-list (the name of the sub-list)

For account ledgers and sub-lists, the names of such items appear on the appropriate row. Text is merely displayed on the row.

Commands are entered via a typed-text menu at the bottom of the screen. All commands should be visible in this menu. There should never be more than a dozen menu items on the screen at any time. Commands should be unique from their first letter and their first letter should be sufficient for activating the function.

Users must be able to move the "current-item" highlight bar up and down through the list. Moving past the end of the list is not permitted. Wrapping the bar from the bottom to the top is not permitted. This is done with the "up" command and the "down" command.

At any time, the user may be able to create a new entry in the list. This entry always appears at the end of the list. For text lines, the entry will consist of a blank line. For account ledgers, the line will create an empty

ledger with the name "<unnamed ledger>" on the line and for sub-lists the line will create a new empty sub-list with the name "<unnamed sub-list>" on the line. The commands to activate these are "text-create", "sub-list create" and "account-create".

The user must be able to edit the current entry in the list. Editing a line of text is simply retyping the line. Editing a sub-list is identical to editing a list, which this section describes. Editing an account ledger is described later. After the editing has completed, the screen must reflect the current state of the system. Any names that have changed must be made up to date. The "edit" command is used to edit the current item.

The user must be able to delete the current item. This removes the item from the list and destroys all data associated with it. In addition, all items below this item are shifted up on the screen to maintain the neat appearance of items in the list. The "delete" command is used to delete an item from the list.

Hierarchy of lists

By allowing a list to contain a sub-list as an element, this accounting system is allowed to have any number of levels of lists, arranged in a strict hierarchy. A sub-list is entered by requesting to edit that list. One returns to the previous list via an "exit" command. There is never any need to explicitly save the contents of a list. Exiting always saves. In fact, there is no way to exit a list without saving the list's contents. Exiting from the top most list in the hierarchy terminates the program.

Naming the list

While editing the list, it is possible to give the lit a name. There should be no ambiguity between lists or accounts, even those with identical names.

Command entry

Specialty graphics are not required. Entering the full command name or the first letter followed by a carriage return is sufficient. Invalid commands should display an error message in the error message area. This area should be cleared after the following command. Commands should be case insensitive

Saving the system

When the system is saved, there should be only one file into which all the

data used by the system is stored. This will prevent our directories from becoming cluttered with many irrelevant files. This file will be named "acctng.dat".

Starting up the system

When the system is invoked, it should restart any previously running system at the point where that system had stopped. This means that any data that was available then must be available and in the same format now. If no previous invocation exists, then the system must create a new empty list and issue some kind of message to the user that no previous version of the system could be found.

There must be a command to exit they system. However, the user must exit the system from the same point that he had entered it, which is always at the top most list. And, of course, exiting always saves the current version of the system.

Editing Account Ledger

The account ledger is a kind of spread sheet. It has account rows on which are listed the dollar amounts of purchases or credits. It has columns in which are listed the catagory of the purchase, the name of the purchase (or credit) and the dollar amount.

Each row can be either a single item line, which has a catagory, name and dollar value, or the row can be the result of a sub-ledger summed over the category listed or the row can be a header row, which has no dollar value and only adds information to the ledger, such as a kind of comment.

Creating a new row adds the values "<no-category>" "<no-name>" and $0.00 to the account. Editing an account-row consists of retyping the category, the name and the cost amount using some kind of form that pops up into the screen.

Creating a new account ledger puts "*" in the category position, "<unnamed account>" into the name position and $0.00 into the cost position of the last row of the account. Editing an account is identical to this scenario.

Creating a comment field puts a blank line with no money into the last line of the file. Editing the comment field is identical to editing a textline, except that fewer characters are allowed.

At the bottom of the account ledger is listed the total with respect to

either some category or "*" which represents all categories. The user must be allowed to change either the name or the summation category at any time. This category is also what appears on account that references a sub-account.

The user may rename the account ledger while editing and may also change the summation category. The special category "*" indicates to sum over all fields.

Just like the list, the sub-account must have a "current item" which is highlighted. The user must be able to move the highlight bar up and down. The user must also be able to delete an item from the list.

Commands

Commands should be consistent between different modes. Commands that perform similar functions in both should have similar names.

Screen for a typical LIST

```
Editing List = "Main-List"

1 Book-list
2 Publication-list
3 Secretarial information
4 Alternate secretaries
5 Secretarial accounting ledger $120.45
6 Good-Restaurants
7 Lunch-account
8 Personal stuff
9 Phone numbers
--------------------------------------------------------------
Error messages:

--------------------------------------------------------------
Commands: u)p d)own e)dit k)ill q)uit n)ame
          t) Create text entry     a) Create account entry
          l) Create list entry
Command>
```

Screen for a typical Accounting ledger

```
   Editing Account = "Lunch-Account"

1 dinner             McDonalds                          5.72
2 martinis           Dorchester lunch                  90.34
3 martinis              Office-party sub-account      200.15
4 champaign           Christmas party                  93.12
5 Remember to buy    more chips & dips this year!
6 dinner              Maxim's                         321.11
7 martinis              K-mart blue-light special       2.50

                      Category: martinis    total $291.34
Error messages:

Commands:   u)p   d)own   e)dit   k)ill   q)uit  n)ame
            t) Create text entry     a) Create account entry
            l) Create list entry     s) Set category
Command>
```

# Vita

Matthew Cameron Humphrey was born in Alexandria, Virgina on May 1, 1963. He showed an amazing aptitude for mathematics and science and earned a high school letter for mathematics achievement. His parents encouraged his scientific studies to the point of enabling him to attend college and graduate from Virgina Polytechnic Institute in December, 1985 with a B.S. degree in computer science and a minor in mathematics.

Unable to simply graduate from VPI as his father had done years before, Mr. Humphrey returned immediately as a graduate student, hot on the trail of his quarry, an M.S. degree. While pursuing this degree, he supported himself through various and sundry research projects which were often far more exciting than the required classes. While this enabled Mr. Humphrey to live in grand and expensive style, it lengthed his M.S. degree from an expected year and a half to nearly three. Despite his best efforts, he accidently completed the degree requirements in July, 1988.

Mr. Humphrey, thusly ejected a second time from VPI, has accepted a teaching position at the University of Waikato in Hamilton, New Zealand, where he does not promise that he will not hunt a Ph.D.

Matthew C. Humphrey