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

Sallie Henry and Matthew Humphrey

TR 88-49

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

by

Sallie Henry
and
Matthew Humphrey

Department of Computer Science
Virginia Polytechnic Institute
Blacksburg, Virginia 24061
Internet: henry@vtodie.cs.vt.edu

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

(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 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 paper describes an experiment which compares the maintainability of two functionally equivalent systems in order to explore the claim that systems developed with object-oriented languages are more easily maintained than those programmed with procedural languages. We found supporting evidence that programmers produce more maintainable code with an object oriented language than a standard procedural language.

I. 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." More scientific evidence is needed.

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] [SWAG78] [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 a 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.

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 consists of 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 programmers to implement the necessary methods, but with the freedom to implement any way they choose, as long as the interface remains the same [BROF86] [RENT82] [BOOG86].

Messages accomplish the processing in an object-oriented system. In a purely object-oriented system (such as SmallTalk [GOLA83]), 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 senders 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 still have doors and can go places, but they are distinct. Inheritance yields the most capability for code reusability, understandability and simplicity [RENT82].

Object-oriented methodologies and object-oriented languages have put forth many unsupported claims. An 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], enables better prototyping and iterative development [BASV75] [COXB84],

and reduces value-type errors because of uniformity of objects [MACB82] [COXB84] [COXB86] [RENT82].

While these claims have a qualitative "rightness" there are little supporting quantitative data. Boehm-Davis claims that object-oriented designs are the hardest to modify, but a procedural language was used in the experiment [BOED86]. Gannon claims that dynamically typed operands (polymorphism) result 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 that have been done were 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 supports the claim that systems developed with object-oriented languages are more maintainable than those developed with procedural languages. In this empirical study, student subjects determined the maintainability of systems developed with two languages by performing maintenance tasks on two functionally identical large programs, one written in an object-oriented language and one written in a procedural language. Maintenance times, error counts, change counts, and programmers impressions were collected. The analysis of the data from this single experiment showed 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 to produce better software systems. One method of testing this goal is by controlled experiment and analysis. This experiment is another piece in the software engineering "mosaic." 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.

II. Criteria to Test

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 a methodology can take months or 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.

In the Boehm-Davis experiment, eighteen 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 provide 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]. 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 were complete when they were 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.

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 subjects [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.

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.

Maintenance and Enhancement

Software typically passes through the life cycle stages of requirements, design, implementation, testing and maintenance. The experiment described in this paper 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)" [LIEB78]. 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 $50 billion to $100 billion on software in 1977, or approximately $25 billion to $75 billion on maintenance [LEHM80].

Software enhancement is the largest portion of maintenance. Fairley estimates that 60% of all maintenance money is spent on perfective maintenance [FAIR85]. That is 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.

III. Experimental Methods

A goal of this research was to support 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 was 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 a "within subjects" test with three independent variables. The variables were the programming language, the subject group and the task. The subjects were randomly divided into two groups: 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 using the other for the task.

All subjects performed three 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. As a warm - up exercise, all subjects performed an initial task that was not included in the data analysis.

This task was equivalent to the others in difficulty, and the subjects were not told that data would not be collected.

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 experimental phase, such that a phase was one academic quarter. The first 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. All 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. During the next section the students were also taught the C language and familiarized themselves with the VAX/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 students code.

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. The background questionnaire measured the students overall Grade Point Average, their Computer Science G.P.A., the number of

months experience programming in C, Pascal, Objective-C and SmallTalk, the number of months experience in integrating code with other programmers code, and the number of months experience in testing software. They were then given a packet 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 students 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. 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 in the specified order, and that only exceeding the eleven week limit would endanger their grade.

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.

Subjects

There were 24 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. Both Groups A and B had ten subjects each.

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].

Tasks

There were three modification tasks that generated the actual data used in this study. 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 users request for change and isolated the task specification from the implementation language.

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 criterion for both systems to be considered identical. 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 looked 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 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. The original systems were developed by a graduate student experienced with both C and Objective C.

Each task consisted of two parts. For Group A subjects, the first part was to perform the task using the C system and the second part was to perform the task using the Objective-C system. For group B subjects, the first part was to perform the task using the Objective-C system and the second part was to perform the task using the C system. Therefore, subjects actually performed each task twice, once using 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.

Each task required that each modification be made to an original copy of the system, as if the request was received with no knowledge of the other requests. Since the subjects did not change modified code the tasks do not cumulatively interfere with each other. It also provides a basis of comparison for all tasks: the original copy. There was 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. Three tasks were selected 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 chosen because they seemed to be 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 they corrected.

Materials

Subjects were given the following information:

- Complete documented source code for the C system,
- Complete documented source code for the Objective-C system,
- The software specifications from which both systems were built,
- Running copy of the original C system,
- Running copy of the original Objective-C per task,
- One file of test data per task

IV. Data Collection

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

There are four independent variables:

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

Student Data

Background data were collected on the subjects to show that the two groups of students were similar and that the random assignment of students to groups produced a fair mixture. All background data were collected using a three page questionnaire that subjects were given one week to complete.

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.

Task Data

Two methods were used to collect the data associated with each task: questionnaires and an automatic data collection facility. Prior to the beginning of the project, the students filled out a questionnaire which supplied the dependent variables of the student data. Table 1 summarizes those variables. 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. 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 four sets of test data. 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.

Table 2 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, an indication as to how each variable was collected and the means and standard error values for the task data. Formal definitions of the variables can be found in Appendix B.

Table 3 contains the means and standard error values for the task data. Table 3 contains the means and standard error values for the task data averaged over the language variable. Table 4 contains the means and standard error values for the task data averaged among the three tasks.

V. Results

The statistical analysis of the student data is described first, followed by the analysis of the task data.

Significant Values for Student Data

An analysis of variance (ANOVA) test was performed on each of the variables using subject identifier and group 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 5.

Except for the computer science GPA of the subject, every variable 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 and no student had previous experience with Objective-C; they had all learned the language during the first part of the experiment. Additionally, no variable shows a difference between any of the subjects.

Significant Values for Task Data

An analysis of variance test was performed on each of the variables using subject identifier, group, language and task. 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 6.

Table 6 also 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.

Some variables are dependent on a combination of 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 was used later to find interactions between conditions on independent variables.

Faults in the Data

According to the statistics there were nine variables in which the task is statistically significant. Additionally, examining the means and standard error values for the variables showed that task values were very different from the other two tasks. It was possible that task 1 was not on the same level of difficulty as the other tasks, skewing the true results. Since this violates an original assumption, the data were re-analyzed, without task 1 as shown in Table 7.

Many of the task dependencies disappeared by omitting task 1. It is most likely that task 1 was too different from the other tasks and was not a good indicator for the experiment. Of the nine variables that were dependent on TASK, only two remain after eliminating task 1.

VI. Conclusions

In order to reach meaningful conclusions, the data must first be shown to be free of any bias. First, the student data showed that there are no significant differences between the two groups of students. Next, the task data showed that there is an expected bias in the tasks that were given to the subjects. After removing this fault, the final conclusions are given. Table 8 presents a synopsis of the conclusions for each variable with Task 1 removed.

Supporting the Hypothesis

Granted this is a single experiment which used students inexperienced in object oriented programming, however, we feel that some interesting observations resulted from this work. This experiment supports the hypothesis that subjects produce more maintainable code with an object-oriented language than procedure-oriented languages. For source 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 fewer changes that are more localized than procedural languages. Additionally, C is never better than Objective-C for any variable used in this study.

While subjects had no previous training in either object-oriented languages or in Objective-C, 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 procedural 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. 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 since Objective-C is a super-set of C there are more options 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. These additions to the language may require more thought and more decisions from the subject.

However, the drawback to this idea 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 by the conclusions reached above. 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.

While it is not possible to fully explain why these perceptions exist without further study, the biasing 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 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 never 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. Hopefully, this experiment will open the way for more tests to verify those claims.

Bibliography

[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, VA, August, 1978.

[BARM77] Bariff, M.L., and Lush, E.J., "Cognitive and personality test for the design of management information systems," Management Science, Vol. 23, No. 8, April 1977, pp. 820-829.

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

[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.

[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, 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, VA 22030, May 1986.

[BOIS74a] Boies S. and Gould J., "Syntactic Errors in Computer Programming," Human Factors, 1974, Vol. 16, No. 3, pp. 253-257.

[BOIS74b] Boies, S.J., and Gould, J.D., "User behavior on an interactive computer system," IBM systems Journal, Vol. 13, No. 1, 1974, pp. 253-257.

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

[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, pp. 207-213.

[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 Evutionary Change in Programming Technology," IEEE Software, Vol. 1, No. 1, Jan. 1984.

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

[CURB79] Curtis, B., Shepp.ard, 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, Vol. SE-5, No. 2, March 1979, pp. 96-104.

[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, Vol. SE-2, 1976, pp. 113-121.

[FAIR85] Fairley, R., Software Engineering Concepts, McGraw-Hill Book Co., New York, NY, 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," International Journal of Man-Machine Studies (1976) Vol. 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, MA, 1983.

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

[HOLR87] Holt, R., Boehm-Davis, D., Schultz, A., "Mental Representations of Programs for Student and Professional Programmers," Psychology Department, George Mason University, Fairfax, VA , 1987.

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

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

[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.

[LIEB78] Lientz, B., Swanson, E., Tompkins, G., "Characteristics of Application Software Maintenance," Communications of the ACM, Vol. 21, No. 6, June. 1978.

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

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

[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.

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

[MYEG78] Myers, G.J., "A controlled experiment in program testing and code walk throughs / inspections," Communications of the ACM, Vol. 21, No. 9, Sept. 1978, pp. 760-768.

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

[RENT82] Rentsch, T., "Object Oriented Programming," SIGPLAN Notices, Vol. 17, No. 9, p. 51, Sept. 1982.

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

[SAAH77] Saal, H.J., and Weiss, Z., "An empirical study of APL programs," Computer Languages, Vol. 2, No. 3, 1977, pp. 47-60.

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

[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," International Journal of Computer and Information Science., Vol. 5, No. 2, pp. 123-143, 1976.

[SHNB79] Shneiderman, B. and Mayer, R., "Syntactic / Semantic Interactions in Programmer Behavior: A Model and Experimental Results," International Journal of Computer and Information Sciences, July, 1979, pp. 219-239.

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

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

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

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

[YOUE74] Youngs, E. "Human Errors in Programming," International Journal of Man-Machine Studies (1974), Vol. 6, pp. 361-376.

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

Table 1. Summary of Student Data Dependent Variables

Variable	Synopsis	measured	Mean A	Mean B
GPA	Subjects overall GPA	before	3.101	2.860
CSGPA	Computer Science GPA	before	3.482	3.090
CURRIC	Subjects curriculum	before		
C	Months of C experience	before	4.700	5.000
PASCAL	Months of Pascal experience	before	27.800	33.200
OBJC	Months of Objective-C experience	before	3.000	3.000
SMALLT	Months of SmallTalk-80 experience	before	0.000	0.100
INTEGR	Months experience integrating code	before	5.100	8.100
TESTX	Months experience testing code	before	49.200	50.900
LEVEL	Academic level	before	3.800	3.900
COURSES	Number of Computer Science courses	before	6.700	7.900
SUBJTASK	Task difficulty, subjective	after	1.950	2.400
SUBJQUES	Questionnaire difficulty, subjective	after	1.150	1.200

Table 2. Task Data Dependent Variables

Variable	Synopsis	Automatically Collected	mean A	mean B	stderr A	stderr B
MODULES	Number of files changed	Yes	2.21	1.95	0.12	0.83
SECTIONS	Number of sections changed	Yes	6.95	6.93	0.74	0.83
LINES	Number of lines different	Yes	69.23	67.40	8.76	9.67
TOTLINES	Difference in file sizes	Yes	46.67	48.51	2.67	4.67
CERR	Number of failed compilations	No	2.50	2.41	0.52	0.29
TC	Number of compilation errors	No	10.02	7.02	2.75	0.98
LE	Number of linking errors	No	0.18	0.25	0.06	0.07
RE	Number of program crashes	No	0.90	1.41	0.17	0.28
LGE	Number of program logic errors	No	1.80	1.37	0.28	0.20
TOTERR	CERR+TC+LE+RE+LGE		15.40	12.47	3.08	1.30
STHIN	Thinking difficulty	No	2.53	3.77	0.14	0.21
SMOD	Modifying difficulty	No	2.97	4.51	0.21	0.21
STEST	Testing difficulty	No	2.64	3.95	0.19	0.26
SALL	Task difficulty	No	2.78	3.95	0.17	0.17
TTHIN	Minutes thinking	No	31.90	35.70	3.69	3.90
PTHIN	Percent attention thinking	No				
TMOD	Minutes modifying	No	77.00	67.90	8.05	6.70
PMOD	Percent attention modifying	No				
TTEST	Minutes testing	No	46.50	40.90	7.23	4.97
PTEST	Percent attention testing	No				
TASKTIME	TTHIN+TMOD+TTEST		155.40	144.50	10.60	10.50

Table 3. 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 4. Raw Task Data, Averaged Between Tasks

Variable	Task 1	Task 2	Task 3	Err-1	Err-2	Err-3
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 5. Results of ANOVA on Student Data Variables over Group

Variable	Confidence	F-table (df)	Significant
GPA		0.2706	No
CSGPA	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 6. 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