A Controlled Experiment to Evaluate Maintainability of Object-Oriented Software

By Sallie M. Henry and Matthew Humphrey

TR 90-39

*

A Controlled Experiment to Evaluate Maintainability of Object-Oriented Software

Sallie M. Henry Matthew Humphrey

Department of Computer Science
Virginia Polytechnic Institute
Blacksburg, Virginia 24061-0106
Internet: henry@vtodie.cs.vt.edu

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 with a standard procedural language.

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 [18] [19] [17] [15] [9] [20] [10] [21]. 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" [19]. 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 a "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 of 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" [4]. Object-oriented design has several definable characteristics. Object-oriented programming directly supports these characteristics. The four attributes are encapsulation, messaging, inheritance, and polymorphism.

Object-oriented methodologies and object-oriented languages have put forth many unsupported claims. Here are a few examples: an object-oriented approach cuts development time [5], makes software "resist both accidental and malicious corruption attempts" [4], is more maintainable [4] [14], more understandable [4], has greater clarity of expression [5], supports the buy versus build software trend [5][8], is easier to enhance [7] [8] [16], enables better prototyping and iterative development [1] [7], and reduces value-type errors because of uniformity of objects [13] [7] [8] [16].

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 their experiment [3]. 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 [11]. 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 [12].

There are other problems with experimentation in software engineering. Many experiments that have been done were conducted on trivial programs which were only a dozen statements long [11]. 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 the other 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.

Experimental Methods

One goal of this research was to support the claims that object-oriented design and implementation yield more maintainable systems. This 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 (C), and the other designed with object-oriented design techniques using an object-oriented language (Objective C). 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 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 two tasks. Each task was performed once on each of the two programs. All subjects performed the tasks in the same order. The tasks were 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 has 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; they performed the tasks and data were collected. All students were enrolled in 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 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. Also, during this time, students were taught the Objective-C language which was available on the same machine as the C language. The programming assignments again included designing, 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 their opinions of the experiment, and short descriptions of their involvement.

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 the use of students as subjects is supported for within subjects experiments by Brooks [6] and supported with empirical evidence by Boehm-Davis [2].

Tasks

There were two 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 user's request for change, and to isolate 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, 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; it 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 it 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 various control constructs; and 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 each of the systems. Performing each part of the task had to be completed before proceeding to the next task. Subjects were allowed to work only 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. Two 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

Data Collection

This experiment collected two sets of data. The first set described the subjects and was used to show homogeneity 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), and
- TASK, the task identifier (1 or 2).

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.

Table 1. Summary of Student Data Dependent Variables

Variable	Synopsis	Measured	Mean A	Mean B
GPA	Subject's overall GPA	before	3.101	2.860
CSGPA	Computer Science GPA	before	3.482	3.090
CURRIC	Subject's 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
MODULES	Number of files changed	Yes	2.21	1.95
SECTIONS	Number of sections changed	Yes	6.95	6.93
LINES	Number of lines different	Yes	69.23	67.40
TOTLINES	Difference in file sizes	Yes	46.67	48.51
CERR	Number of failed compilations	No	2.50	2.41
TC	Number of compilation errors	No	10.02	7.02
LE	Number of linking errors	No	0.18	0.25
RE	Number of program crashes	No	0.90	1.41
LGE	Number of program logic errors	No	1.80	1.37
TOTERR	CERR+TC+LE+RE+LGE		15.40	12.47
STHIN	Thinking difficulty	No	2.53	3.77
SMOD	Modifying difficulty	No	2.97	4.51
STEST	Testing difficulty	No	2.64	3.95
SALL	Task difficulty	No	2.78	3.95
TTHIN	Minutes thinking	No	31.90	35.70
PTHIN	Percent attention thinking	No		
TMOD	Minutes modifying	No	77.00	67.90
PMOD	Percent attention modifying	No		
TTEST	Minutes testing	No	46.50	40.90
PTEST	Percent attention testing	No		
TASKTIME	TTHIN+TMOD+TTEST		155.40	144.50

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.

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 3.

Table 3 also shows the variables over which statistically significant differences were found.

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. Table 3 presents a synopsis of the conclusions for each variable.

Supporting the Hypothesis

Even though this is a single experiment which used students inexperienced in object oriented programming, 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 with a procedure-oriented language. 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

Table 3. Summary of Task Data Variables

Variable	Synopsis	Means
MODULES	number of changed modules	Objective-C requires changing fewer modules
SECTIONS	number of changed sections	
LINES	number of different lines	Objective-C requires changing fewer lines
TOTLINES	number of new lines	Objective-C requires adding fewer lines
CERR	number of failed compilations	
TC	total of compiler errors	
LE	number of linking errors	
RE	number or run-time errors	
LGE	number of logic errors	Task 3 produces more logic errors
TOTERR	total error count	
STHIN	subjective thinking difficulty	Group B finds thinking about task more difficult
SMOD	subjective modifying difficulty	Group B finds modifying task more difficult Subjects found task 3 more difficult to modify
STEST	subjective testing difficulty	
SALL	total subjective difficulty	Group B finds tasks more difficult overall
TTHINK	minutes spent thinking	
TMOD	minutes modifying	Subjects and task interacted to increase testing time
TTEST	minutes spent testing	
TASKTIME	total time on task	

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.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

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

[7] Cox, B., "Message/Object Programming: An Evolutionary Change in Programming Technology," IEEE Software, Vol. 1, No. 1, Jan. 1984.

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

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

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

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

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

[13] MacLennan, B., "Values and Objects in Programming Languages," SIGPLAN Notices, Vol. 17, No.12, p. 70, Dec. 1982.

[14] Meyer, B., "Towards a Two-dimensional Programming Environment," Readings in AI, Palo Alto, CA, Tioga, 1981, p.178.

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

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

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

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

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

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

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