

**An Empirical Study of the Object-Oriented
Paradigm and Software Reuse**

*John A. Lewis, Sallie M. Henry,
Dennis G. Kafura, and Robert S. Schulman*

TR 92-14

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

April 13, 1992

An Empirical Study of the Object-Oriented Paradigm and Software Reuse

by

John A. Lewis

Sallie M. Henry

Dennis G. Kafura

(Department of Computer Science)

and

Robert S. Schulman

(Department of Statistics)

ABSTRACT

Little or no empirical validation exists for many of software engineering's basic assumptions. While some of these assumptions are intuitive, the need for scientific experimentation remains clear. Several assumptions are made about the factors affecting software reuse, and in particular, the role of the object-oriented paradigm. This paper describes the preliminary results of a controlled experiment designed to evaluate the impact of the object-oriented paradigm on software reuse. The experiment concludes that (1) the object-oriented paradigm substantially improves productivity, although a significant part of this improvement is due to the effect of reuse, (2) reuse without regard to language paradigm improves productivity, (3) language differences are far more important when programmers reuse than when they do not, and (4) the object-oriented paradigm has a particular affinity to the reuse process.

Virginia Tech
Blacksburg, Virginia 24061
Internet: lewis@vtopus.cs.vt.edu

1. Introduction

The use of precise, repeatable experiments to validate any claim is the hallmark of a mature scientific or engineering discipline. Far too often, claims made by software engineers remain unsubstantiated because they are inherently difficult to validate or because their intuitive appeal seems to dismiss the need for scientific confirmation.

The effects of the object-oriented paradigm on software reuse is one area which begs further empirical investigation. While the characteristics of the object-oriented approach and the qualities which support successful reuse seem to complement each other, little empirical evidence has been given to support this relationship.

Studies related to software reuse are important because of the key role reuse assumes in improving software productivity. Brooks asserts that while no single development will result in an order-of-magnitude increase in productivity, software reuse is an area where the greatest results can be achieved because reuse addresses the "essence", as opposed to the "accidents," of the development problem [BROF87]. However, reusable software is not being exploited to its full potential. According to Freeman, the state-of-the-practice of reuse in the United States is embarrassing [FREP87]. On an evolutionary scale, he puts reuse technology in an awakening stage, slowly approaching an early utilization period. Developers and users of potentially reusable products are unnecessarily hampered because they lack specific knowledge concerning the factors which influence software reusability.

Studies of the object-oriented paradigm are important because, according to Biggerstaff's framework of software reuse [BIGT87], the object-oriented paradigm has a good balance between power and generality. In his framework, procedural based solutions are also depicted having a good balance, but are considered less effective than object-oriented solutions. The fundamental characteristics of the object-oriented paradigm seem to complement the needs of the reusing developer. Encapsulation capabilities create self-contained objects which are easily incorporated into a new design [KERB84]. The data-based decomposition of objects, resulting in class-hierarchies and inheritance, promotes reuse far more than the top-down approach which promotes "one-of-a-kind" development [MEYB87]. Greater abstraction is the key to to greater reusability, and object-based languages provide abstraction far better than procedural languages [WEGP83].

Tracz makes several points which tend to support the use of the object-oriented paradigm to promote successful reuse [TRAW88]. For instance, special tools are not employed in current successful software reuse situations and are not the answer to the problem. Reuse success comes from formalizations of process and product, which an object-oriented environment creates. Furthermore, unplanned software reuse is costly. Software must be designed for reuse, with an emphasis on interface and modularization. The object-oriented paradigm stresses these characteristics. However, Tracz admits that no language, including an object-oriented one, will solve the the reuse problem inherently.

While the reuse potential of the object-oriented paradigm has been promoted, little scientific experimentation has been conducted to confirm this expectation. The research described in this paper provides a comparison of a standard procedural approach to that of the object-oriented techniques. A controlled experiment was designed and executed in order to measure the relative affects of a procedural language and an object-oriented language in terms of software reuse.

Similar experiments have been successful in making comparisons of the object-oriented and procedural approaches relative to other aspects of software development. One study determined that the object-oriented paradigm is quantitatively more beneficial than a procedural approach in terms of software maintenance [HENS90]. An interesting point made in that research is that subjects viewed the object-oriented techniques as more difficult to accomplish, even though all objective data supported the hypothesis that using it resulted in fewer maintenance tasks and reduced maintenance effort. This result clearly illustrates the danger of relying only on anecdotal evidence to assess software engineering technology.

The goal of the experiment described in this paper is to answer the following questions with respect to the impact of the object-oriented paradigm vs. the procedural paradigm on the successful reuse of software components:

- 1) Does the object-oriented paradigm promote higher productivity than the procedural paradigm?
- 2) Does reuse promote higher productivity than no reuse?

- 3) Does the object-oriented paradigm promote higher productivity than the procedural paradigm when programmers do not reuse?
- 4) Does the object-oriented paradigm promote higher productivity than the procedural paradigm when programmers reuse?
- 5) Does the object-oriented paradigm provide incentives to reuse above those of the procedural paradigm?

The experimental design was constructed with these questions in mind. We define productivity as the inverse of the effort expended to produce a specific software product. Effort is measured in several quantifiable ways. We hypothesize that both reuse and the object-oriented paradigm are important factors in the software development effort.

The next section describes the design of the experiment and discusses the specifications of the tasks performed. Section 3 defines the data collected and the statistical analysis performed. Section 4 draws conclusions from the analysis, specifically addressing the questions presented above. Finally, Section 5 summarizes the experimental results and discusses future work in this area.

2. Experimental Design

Some reuse experiments employ hypothetical, question-and-answer situations where the subjects do not actually perform all the various tasks inherent in the reuse process. The authors believe, however, that to accurately determine influential factors, the experimental subjects must perform all of the following tasks: evaluating potentially reusable products, adapting them to the new situation, and integrating them into a functionally complete product. It is important to create, as accurately as possible, a representative situation while maintaining a valid experimental design [CURB80].

The experiment described in this paper is based on a target system developed by each of a set of senior-level software engineering students. The use of students as subjects, while sometimes considered unrealistic, is justified in this case due to two

overriding considerations. First, empirical evidence by Boehm-Davis indicates that students are equal to professionals in many quantifiable measures, including their approach to developing software [BOED84]. Although new techniques are learned and further refinement does occur, a programmer's basic approach and development habits are formed quite early in their professional development. Second, given the amount of control necessary to execute this experiment, students are the only viable alternative. The efficacy of students as subjects is supported for within-subject experiments by Brooks [BROR80].

The subjects in this experiment developed a specified target system. The system specification is couched in the guise of computerizing a fictional company and is separated into two tasks. The specific functions making up the system were abstracted from the commercial software development experience of the first author. They involve a variety of programming techniques including data management, numerical processing, and graphics.

Previous research investigating the factors affecting software reuse have concentrated on two issues: 1) the impact of software engineering characteristics of code components, such as readability, structured code, etc., and 2) the techniques used to find appropriate code components from a set of possible candidates. Neither of these issues are the focus of this study. Code quality was allowed to vary only within the controlled confines of "adequate" testing and software engineering standards. All completed projects were verified to meet a set of requirements concerning documentation, code quality, and functional correctness. Furthermore, subjects were given no special tools for searching or identifying candidate components. It is assumed that any assistance in this area would only improve the reuse results. This study focuses on language issues, specifically the comparison of the object-oriented paradigm to the procedural approach with respect to reuse of previously developed components.

In the experiment, reusable code components were made available to the subjects implementing the target system. To affect further control, the code component sets were specifically generated for this study. Therefore, the research consists of two phases. The first phase was preparatory, in which potentially reusable components were designed and implemented. The experiment was executed in the second phase, in which the target system was developed by a set of subjects, who are unrelated to the programmers who designed and implemented the reusable components. These two phases are described in the following sections.

2.1 Phase One: Component Development

Two sets of potentially reusable components were created during phase one. One set was implemented in a procedural based language, Pascal, and the other in an object-oriented language, C++. The choice of languages was not arbitrary. We deliberately chose not to use C as the procedural programming language to make as clear a distinction as possible between the object-oriented approach and the procedural approach. Since C is basically a subset of C++, we feared the similarities of the two languages might cloud that distinction. C++ was chosen over Smalltalk because we believed the powerful programming environment of Smalltalk, not available to a Pascal programmer, might jeopardize the comparison. Finally, both C++ and Pascal emphasize strong typing, thus controlling another possible source of variation.

Both component sets were implemented on Apple Macintosh II's running A/UX. They were designed to be functionally equivalent, though design and coding techniques naturally vary. Equivalence was guaranteed by ensuring that all code met the same fundamental functional and error-handling requirements. Furthermore, all developed code passed the same level of testing thoroughness.

Knowing the requirements of the target system to be implemented in the second phase, each component was designed to have a specific level of applicability. The levels of reuse can be described as:

- 1) completely reusable,
- 2) reusable with slight revision (< 25%),
- 3) reusable with major revision (> 25%), and
- 4) not applicable for reuse.

With respect to the target system, the component sets were designed to contain elements from each reuse level. The 25% marks of levels 2 and 3 are only intuitive guidelines and refer to the amount of code that must be added, deleted and modified to create a component that meets the target system's requirements. Providing components which span a wide range of applicability ensures a realistic, diverse collection from which subjects evaluate and choose components.

2.2 Phase Two: Project Implementation

Using the two sets of components, independent subjects were assigned the task of implementing the target project. The subjects were divided into four groups, pictured as cells in Figure 1. Half the subjects implemented the project in Pascal, the other half in C++. Furthermore, a portion of the students from each language were not allowed to reuse at all, while the others were encouraged to reuse. The "no reuse" groups serve as control groups.

Twenty-one subjects were distributed unevenly across the groups. The uneven distribution of subjects was factored into the statistical analysis. The subjects were divided into the groups randomly, but were statistically blocked across their computer science grade point averages. This blocking was an effort to reduce variability within each group. An anova test comparing the grade point averages of subjects showed no significant differences between groups ($p < 0.9795$).

No Reuse Procedural	Reuse Procedural
No Reuse Object-Oriented	Reuse Object-Oriented

Figure 1: Subject group breakdown.

The functional requirements of the system are divided into two equal tasks concerned with "employee management" and "business management." Employee management functions deal with processes such as an employee database, payroll, security control, and cost center management. Business management functions are concerned with the details of shop floor control, quality control testing, warehouse management, and customer interactions.

Although the two tasks focus on different aspects of running a business, they were designed to be comparable in computational complexity. Both are divided into seven

subtasks, each of which has a counterpart in the other task designed to require approximately the same amount of effort to develop. Since preliminary analyses showed that the results are not affected by the difference between the two tasks, this factor was ignored in subsequent analysis, thereby increasing statistical power in addressing the questions of primary interest. To further control this aspect of the experiment, half of the subjects designed and implemented the employee management task first, while the other half of the subjects designed and implemented the business management task first. Then each half switched, resulting in both system tasks being developed by each subject. This organization offsets any learning benefit of doing a particular task first. An introductory material section was provided with information both groups need to perform their respective tasks.

3. Data Analysis

Given a feasible and well-controlled experimental design, the rigorous analysis of data collected during the experiment determines the conclusions that can be made about the hypotheses. In this experiment, the goal is to determine which groups from Figure 1, on average, had a significantly different productivity rate than others.

The data collected during the experiment measures the productivity of a subject in implementing the target system. Productivity and effort are considered to have an inverse relationship. Therefore, the less effort expended by a subject to satisfy the requirements of one task, the higher the productivity of that subject. The measurements of effort, and therefore of productivity, are:

Main productivity measures

- **Runs** - The number of runs made during system development and testing,
- **RTE** - The number of run time errors discovered during system development and testing,
- **Time** - The time (in minutes) to fix all run time errors,

Secondary productivity measures

- **Edits** - The number of edits performed during system development and testing, and
- **Syn.** - The number of syntax errors made during system development and testing.

Since each subject implemented the same tasks, a comparison of data across subjects yields a relative measure of the effort used to develop a task. A subject with a high value for a given measure is considered less productive than a subject with a low value.

Multiple productivity variables are used to obtain a complete picture of the development process. The Runs, RTE and Time variables, given their significance to the development process, are considered the main variables of interest. The Edits and Syntax Errors variables are gathered for completeness, but are given less emphasis. To reduce the overhead of the data collection, some measures that might have been of interest, such as total development time, were not collected.

Data was collected by the subjects using tally sheets. To assure the data's validity, subjects were informed that their names would not be associated with these data, and that the values themselves would have no bearing on their course evaluation. They were also told that a negative impact on their course evaluation would occur if they did not record their development information honestly and completely. The tally sheets were coded such that no subject name was ever connected to particular data.

The group means for each productivity variable are depicted graphically in Figure 2. These charts give a rough indication of how the groups compare although statistical analyses are employed to verify perceived differences. In each analysis, a difference in means was considered significant if the p-value for the test was less than 5% ($p < 0.05$), which is an accepted norm. Since our research questions all predict the direction of difference, all tests were performed in a one-sided manner.

Initial analysis of the task factor determined that the difference between the two tasks played no role in influencing any of the productivity variables (all p-values for task effects were ≥ 0.2073). In other words, the two tasks were determined to be equally difficult. The lack of difference is attributed to the careful design of task specifications and the blocking of subjects across grade point average. Therefore, all further analyses ignore

the task factor, effectively creating 42 observations on which to perform the tests, which gives them more statistical power.

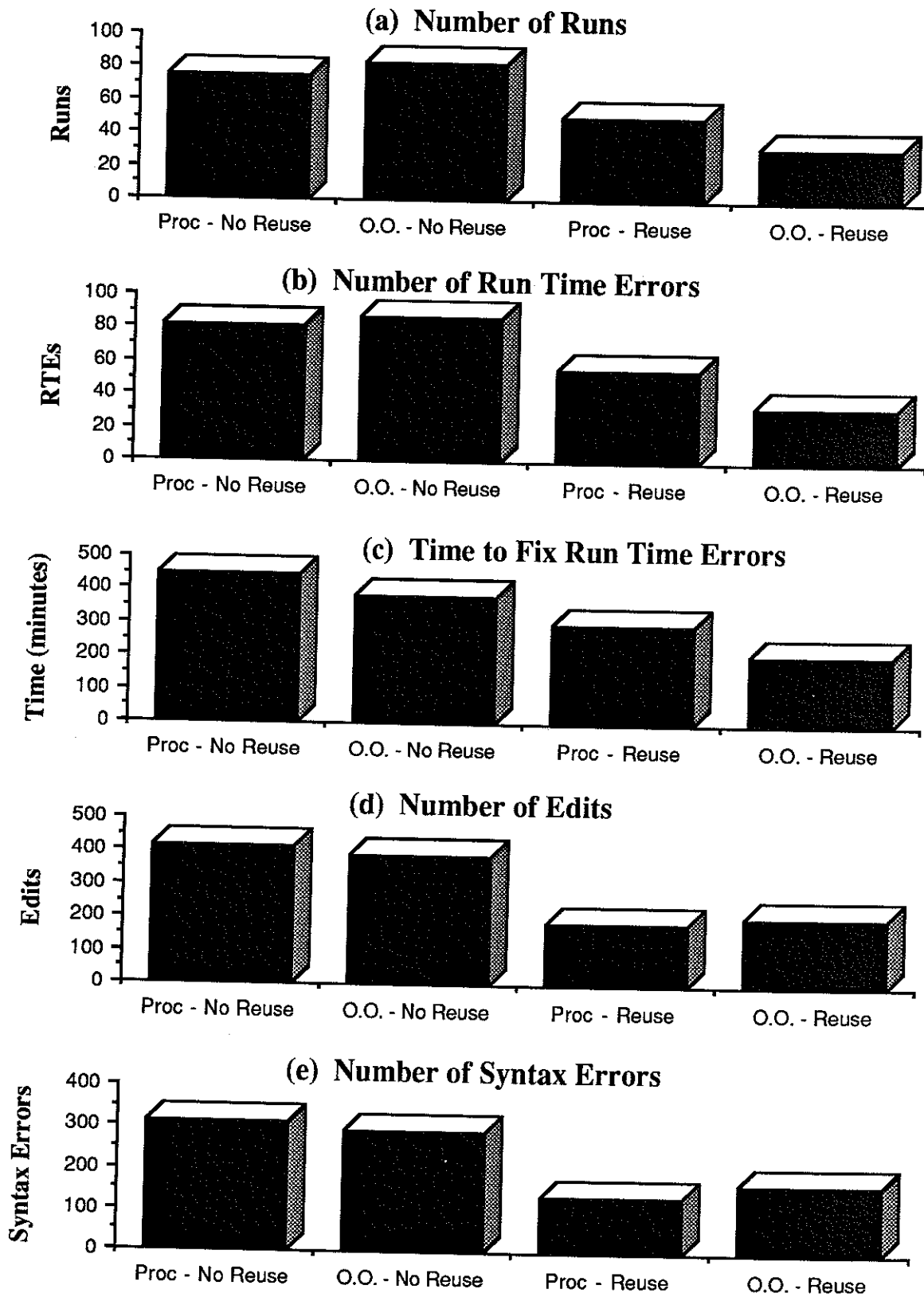


Figure 2. Group means for each production variable.

4. Experimental Results

This section draws conclusions from the analysis performed on the productivity data. In general, the hypotheses suggested at the beginning of this paper are supported, with some notable exceptions.

The experimental questions posed in Section 1 will be used as a framework for discussion of the statistical analysis. Each question will be addressed separately, giving the results of the appropriate analysis.

1) Does the object-oriented paradigm promote higher productivity than the procedural paradigm?

The third column in Table 1 list the means of the productivity variables calculated from all subjects using the procedural language, including subjects who reused as well as those who did not. The fourth column shows similar means for subjects in the object-oriented categories. Our hypothesis is that the values for the object-oriented paradigm will be lower than those of the procedural paradigm, indicating a higher productivity for the object-oriented subjects.

	Significant?	p-value	Means	
			Procedural	O - O
Runs	Yes	0.0066	59.27	47.50
RTE	Yes	0.0078	65.00	50.20
Time	Yes	0.0104	354.41	261.70
Edits	No	0.3469	271.55	263.65
Syn.	No	0.8675	183.67	202.40

Table 1. Language Main Effect

For each productivity variable, a p-value was computed for the difference between the means. The p-value is the probability that the difference could have been obtained by chance, rather than reflecting a true difference in productivity. Following conventional criteria, a difference is deemed statistically significant if its p-value is less than 0.05. In such cases, it is extremely unlikely that the difference in means is due to chance, and we

conclude that productivity was indeed higher for subjects using the object-oriented paradigm.

The three main productivity variables (Runs, RTE and Time) show a significant difference between the means, favoring the object-oriented paradigm. In addition, the object-oriented mean for the Edit variable was also lower than the procedural mean, although not to a significant degree. The means on the Syntax Errors variable did not differ in the predicted direction. Considering the nature of the Edits and Syntax Errors variables, the lack of significance is attributed to the subjects lack of practice using the object-oriented language. The results of the analysis on the main variables indicate that the object-oriented paradigm does promote higher productivity than the procedural paradigm.

2) Does reuse promote higher productivity than no reuse?

From the results in Table 2, the answer to this question is clearly yes. The means in the third column of Table 2 are calculated for all subjects who did not reuse, regardless of the language used. Likewise, the fourth column shows means for all subjects who did reuse. Our hypothesis is that the means will be lower for the reuse groups, indicating a higher productivity for the subjects who were encouraged to reuse. This hypothesis is strongly supported by all variables.

	Significant?	p-value	Means	
			No Reuse	All Reuse
Runs	Yes	0.0001	78.71	41.14
RTE	Yes	0.0001	83.79	45.04
Time	Yes	0.0001	420.07	255.36
Edits	Yes	0.0001	405.71	198.82
Syn.	Yes	0.0001	302.14	150.92

Table 2. Reuse Main Effect

This result is further supported by the charts in Tables 3 and 4, which view the data across the reuse factor, but consider each language separately. Table 3 shows the means for the procedural groups with respect to reuse, and Table 4 shows the means for the

object-oriented groups with respect to reuse. In both analyses, all variables showed a significant difference in the hypothesized direction.

	Significant?	p-value	Means	
			Procedural No Reuse	Procedural All Reuse
Runs	Yes	0.0001	75.38	50.07
RTE	Yes	0.0008	81.25	55.71
Time	Yes	0.0047	446.38	301.86
Edits	Yes	0.0001	416.00	189.00
Syn.	Yes	0.0001	311.00	137.14

Table 3. Procedural (No Reuse vs. Reuse)

	Significant?	p-value	Means	
			O - O No Reuse	O - O All Reuse
Runs	Yes	0.0001	83.17	32.21
RTE	Yes	0.0001	87.17	34.36
Time	Yes	0.0017	385.00	208.86
Edits	Yes	0.0001	392.00	208.64
Syn.	Yes	0.0001	290.33	164.71

Table 4. Object-Oriented (No Reuse vs. Reuse)

3) Does the object-oriented paradigm promote higher productivity than the procedural paradigm when programmers do not reuse?

The means listed in Table 5 are calculated for subjects who did not reuse. The third column represents subjects using the procedural language, and the fourth column represents subjects using the object-oriented language. Our hypothesis is that the object-oriented values will be lower than the procedural values. Surprisingly, none of the variables indicate a significant difference.

Interestingly, the group means do not consistently favor one language or the other. The means for the object-oriented groups are lower for Time, Edits, and Syntax Errors, but

the means for the procedural groups are lower for Runs and RTEs. According to this analysis, we must conclude that when reuse is not a factor, the object-oriented paradigm does not promote higher productivity. In other words, in the absence of reusable components, either language works equally well.

	Significant?	p-value	Means	
			Procedural No Reuse	O - O No Reuse
Runs	No	0.8909	75.38	83.17
RTE	No	0.7506	81.25	87.17
Time	No	0.1607	446.38	385.00
Edits	No	0.2360	416.00	392.00
Syn.	No	0.1733	311.00	290.33

Table 5. No Reuse (Procedural vs. Object-Oriented)

- 4) Does the object-oriented paradigm promote higher productivity than the procedural paradigm when programmers reuse?

Given the answers to the first and third questions, the answer to this question should logically be yes. The results in Table 6 confirm this for the three main productivity variables. The means listed are for subjects who did reuse, with the third column representing subjects using the procedural paradigm and the fourth column representing subjects using the object-oriented paradigm. Once again, our hypothesis favors the object-oriented paradigm.

	Significant?	p-value	Means	
			Procedural All Reuse	O - O All Reuse
Runs	Yes	0.0001	50.07	32.21
RTE	Yes	0.0005	55.71	34.36
Time	Yes	0.0153	301.86	208.86
Edits	No	0.8380	189.00	208.64
Syn.	No	0.9767	137.14	164.71

Table 6. Reuse (Procedural vs. Object-Oriented)

Variables Runs, RTE and Time all proved significant with means favoring the object-oriented group, but the Edits and Syntax Errors variables did not differ in the hypothesized direction. Given the importance of the main productivity variables, we can conclude that the object-oriented paradigm does promote higher productivity than the procedural paradigm when reuse is employed. Note that most of the support given to the first question comes from differences between the groups which were encouraged to reuse.

5) Does the object-oriented paradigm provide incentives to reuse above those of the procedural paradigm?

As shown by the results in Tables 3 and 4, reuse improved productivity over non-reuse for both the procedural and object-oriented paradigms. The fifth question asks whether the *extent* of improvement is comparable for the two language paradigms. Our hypothesis is that the improvement due to reuse will be greater for the subjects using the object-oriented paradigm than those using the procedural paradigm, indicating that the object-oriented paradigm is particularly suited to reuse.

The third column in Table 7 shows for each variable the difference between the mean of the procedural non-reuse group and the mean of the procedural reuse group. This is a measure of the amount of improvement in productivity due to reuse -- the larger the difference, the greater the increase in productivity. The fourth column shows comparable mean differences for the object-oriented groups. Therefore, our hypothesis predicts that values in the fourth column should be greater than those in the third column.

	Significant?	p-value	Mean Differences	
			Procedural NR - R	O - O NR - R
Runs	Yes	0.0009	25.31	50.96
RTE	Yes	0.0062	25.54	52.81
Time	No	0.3176	144.52	176.14
Edits	No	0.8753	227.00	183.36
Syn.	No	0.9716	173.86	125.62

Table 7. Interaction (Extent of Improvement)

On the Runs and RTE variables, the increase in productivity due to reuse was greater for the object-oriented paradigm than for the procedural paradigm. The same pattern occurred on the Time variable, although the difference in means was not large enough to be statistically significant. Once again, contrary to the main productivity variables, the Edits and Syntax Errors variables seem to oppose the hypothesis. Given that two of the three main measures of productivity (Runs and RTE) show significant differences in the hypothesized direction, and that the third main variable (Time) favored the same direction, we conclude that there is a significant difference between the extent of improvement due to reuse across the two language paradigms. In other words, the results show that the object-oriented paradigm demonstrates a particular affinity to the reuse process.

5. Summary and Future Work

The experiment in this paper has shown that:

- (1) The object-oriented paradigm substantially improves productivity, although a significant part of this improvement is due to the effect of reuse (questions 1, 3 and 4),
- (2) Software reuse improves productivity no matter which language paradigm is used (question 2),
- (3) Language differences are far more important when programmers reuse than when they do not (questions 3 and 4), and
- (4) The object-oriented paradigm has a particular affinity to the reuse process (question 5).

Although we did not demonstrate that the object-oriented paradigm promotes productivity when reuse is not a factor, the development efforts using either language paradigm were not significantly different. Furthermore, given the reuse potential demonstrated by the object-oriented paradigm, greater benefits can be achieved by using the object-oriented paradigm than by using a procedural approach.

An important facet of the experimental method is that the results are repeatable. Experiments similar to the one described in this paper should be conducted to verify the

results of this experiment. In particular, the secondary variables of Edits and Syntax Errors did not always support the analysis of the main variables, even when intuition says they should. This tendency deserves further investigation.

Other experiments should be conducted which independently investigate the two main elements of this research: software reuse and the object-oriented paradigm. The factors which affect software reuse are many and varied. Similar experiments can be designed to determine the impact of human factors, code characteristics, and other language differences.

The object-oriented paradigm contains a wealth of possible benefits that have yet to be proven empirically. Claims that associate the object-oriented approach with improved design, less and easier maintenance, and higher reliability when compared to its procedural counterpart demand further investigation. Experience reports alone, while useful, are not enough to validate the assumptions that are associated with the object-oriented paradigm. Experimental research into these areas is necessary to provide a solid base to support the theories that shape state-of-the-art software production.

References

- [BIGT87] Biggerstaff, T., Richter, C., "Reusability Framework, Assessment, and Directions," *IEEE Software*, March 1987, pp. 41-49.
- [BROF87] Brooks, F.P., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, April 1987, pp. 10-19.
- [BOED84] Boehm-Davis, D., Ross, L., "Approaches to Structuring the Software Development Process," *International Journal of Man-Machine Systems*, (to appear 1991).
- [BROR80] Brooks, R., "Studying Programmer Behavior Experimentally: The Problems of Proper Methodology," *Communications of the ACM*, 1980, Volume 23, Number 4, pp. 207-213.
- [CURB80] Curtis, B., "Measurement and Experimentation in Software Engineering," *Proceedings of the IEEE*, 1980, Volume 68, Number 9, pp. 1144-1157.
- [FREP87] Freeman, P., "A Perspective on Reusability," *Software Reusability*, Computer Society Press of the IEEE, 1987, pp. 2-8.
- [HENS90] Henry, S.M., Humphrey, M., Lewis, J.A., "Evaluation of the Maintainability of Object-Oriented Software," *Proceedings of the Conference on Computer and Communication Systems*, Volume 1, Hong Kong, September 1990, pp. 404-409.
- [KERB84] Kernighan, B.W., "The Unix System and Software Reusability," *IEEE Transactions on Software Engineering*, September 1984, pp. 513-518.
- [MEYB87] Meyer, B., "Reusability: The Case for Object-Oriented Design," *IEEE Software*, March 1987, pp. 50-64.
- [TRAW88] Tracz, W., "Software Reuse Myths," *ACM SIGSOFT Software Engineering Notes*, January 1988, pp. 17-21.

[WEGP83] Wegner, P., "Varieties of Reusability," ITT Proceedings of the Workshop on Reusability in Programming, 1983, pp. 30-44.