01 May 1984

An Experimental Study of the Effects of Modularity on Resource Consumption in Software Development

Alan D. Christiansen

Arlan R. Dekock
Missouri University of Science and Technology, adekock@mst.edu

John Bruce Prater
Missouri University of Science and Technology

Recommended Citation
Christiansen, Alan D.; Dekock, Arlan R.; and Prater, John Bruce, "An Experimental Study of the Effects of Modularity on Resource Consumption in Software Development" (1984). *Computer Science Technical Reports*. 58.
https://scholarsmine.mst.edu/comsci_techreports/58

AN EXPERIMENTAL STUDY OF THE EFFECTS OF MODULARITY ON RESOURCE CONSUMPTION IN SOFTWARE DEVELOPMENT

Alan D. Christiansen*, Arlan R. DeKock, and John B. Prater

CSc-84-3

Department of Computer Science

University of Missouri-Rolla

Rolla, MO 65401 (314)-341-4491

*This report is substantially the M.S. thesis of the first author, completed May, 1984.

ABSTRACT

Many authors have encouraged the use of modular programming techniques in software development. In fact, there is almost total agreement within industrial and academic circles that modularity is a desirable feature of any software package. Unfortunately, the desirability of modular design is almost always voiced without support from experimental evidence.

This paper consists of an experiment comparing the resource consumption of programmers based on the modularity practices employed during the design and programming phases of software development. The experiment tests the effectiveness of modularity in reducing psychological complexity of software.

The results of the research show that in some cases there is indeed a difference in resource consumption between the modularity practices tested. However, the stated benefits of modularity did not carry over to the design and programming phases of software development. The use of modularity seemed, in fact, to increase development costs in some cases.

ACKNOWLEDGEMENT

The authors wish to express their gratitude to all those who contributed to this paper. Dr. V. A. Samaranayake of the UMR Mathematics Department provided many helpful criticisms of the statistical analyses used in the experiment. Also, thanks are extended to all the experimental subjects. The interest, good humor, and flexibility shown by these students is greatly appreciated.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

Modular design of software is widely accepted as being beneficial to both the programmer and the software product. That is, by employing modular design techniques, it is assumed that the programmer should be able to more easily complete a programming task and should create a more maintainable program than if modular design were not used.

This paper describes an experiment which was performed during the Spring 1983 semester. The experiment was designed to test the relationship of modular program design techniques to the development costs incurred during the design and programming phases of software development. The experiment will directly test the effects of modularity on the effort needed to complete a programming task. It should also be noted that software maintainability is enhanced by modularity, and therefore this research relates to both the initial programming and maintenance phases of the software life cycle.

In the remainder of this paper, the experiment is presented. Section II describes the literature related to the performed experiment. In Section III, the experiment is detailed, and the statistical methods used for analysis of the data are discussed. The analyses are summarized in Section IV, and then interpreted in Section V. Finally, in Section VI, the research is summarized and suggestions are made for further related study.

II. REVIEW OF THE LITERATURE

A. PSYCHOLOGICAL JUSTIFICATION FOR MODULARITY

It is quite rare to find complete agreement on techniques employed in software engineering. Modular design is one such technique, however. Almost no one questions the necessity of modular design of large software systems. The justification for the use of modularity is generally based upon the premise that modularity minimizes the "psychological complexity" of a software package. This complexity is assumed important in both the design of new software and the modification of existing software. A minimum psychological complexity of program would in some way correspond to the maximum modifiability of the program.

There is a theory within psychology that humans can only manage a small number of intellectual tasks at one time [1]. This number is usually claimed to be seven, plus or minus two tasks (or sometimes, mental discriminations). This "magic number seven" is well known in the literature. Since programming is, more than anything else, a thought process, Frost [1] states that programming should be viewed as an intellectual management task and should be subject to the same research methods as are applied in the field of psychology. There are in fact several references [2],[3],[4] which are surveys of the considerations of applying experimentation on cognitive behavior to the field of computer science. References [5],[6], and [7] are examples of such experimentation.

Following the example of these authors, the experiment described in this paper was designed and executed. The underlying ideas within the tested hypothesis are related to the ability of programmers to program quickly and efficiently, as controlled by the methods of program design.

B. MODULARITY, MODIFIABILITY, AND SOFTWARE MAINTENANCE

Though the motive of this research is to study the behavior of individuals in designing and programming new software, one of the primary reasons that modularity is encouraged does not directly concern itself with the ability of programmers to create a new software package efficiently. Rather, the concern is in creating a modifiable and maintainable program. This maintainability is achieved through the use of a restricted set of programming techniques affecting control flow and data flow. Modular design in control flow is achieved through the use of subroutines and functions. Modularity in data flow is achieved through the exclusion of global variables and common (external) data areas. Estimates vary, but it has been said that much, if not most, of the cost of software development is due to maintenance considerations [2]. So, if these assumptions are correct, there is a powerful inducement towards modular design from a purely economic standpoint.

This modularity is thoroughly ingrained within all the popular design methodologies [8],[9],[10],[11]. In fact, the work of Myers [11] treats the ideas of control and data flow modularity in great detail. He uses the terms module strength and module coupling for the respective degrees of modularity. Levels of quality for each type of modularity are identified and discussed.

This work is unique among the above methodologies as Myers has attempted to quantify the degree of modularity that is achieved in a software package. Through these measures it is then possible to make broad judgements of the quality of a software package by inspecting the attributes of strength and coupling. It is very important to recognize that there are two unique types of modularity. Control-flow and data-flow modularity may not impact the programmer (or the software produced) to the same degree. Later in this paper, these two types of modularity will be separately tested to ascertain their respective impacts.

Finally, it should be noted that the ideas of minimizing psychological complexity of software and maximizing maintainability of software are completely compatible. In fact, when the complexity of software is decreased, the understandability of that software is increased, and therefore modification tasks should be more easily completed. It can then be argued that the research described in following sections is also related to aspects of software development past the design and programming phases.

III. THE EXPERIMENT

Following standard methods of experimentation on cognitive behavior, a null hypothesis was formed. This null hypothesis (H_0) is the negation of the hypothesis of interest, and various statistical tests can be used to determine whether the null hypothesis may be rejected. If H_0 is rejected, then the alternative hypothesis (H_1) must be accepted, as H_1 is the logical negation of H_0. For this experiment, the null and alternative hypotheses may be stated as:

H_0: "There is no difference in resource consumption levels of programmers due to modularity practice employed by the programmers."

H_1: "There is a difference in resource consumption levels of programmers due to modularity practice employed by the programmers."

In testing whether H_0 should be accepted or rejected, a way of numerically measuring "resource consumption" is needed if the statistical methods mentioned above are to be used. In fact, several metrics (measures) were chosen, and each was individually tested. These metrics are described later in this section along with the details of the experiment. By separately considering each metric, the null hypothesis is interpreted in slightly differing ways. In a sense, there is a separate null hypothesis for each metric employed in the experiment.

It is also necessary to select a confidence level for accepting or rejecting H_0 based on the statistics. One arbitrary but usual choice is to select a 95% confidence level, which means that on average, 5% of the inferences made from the statistics will be incorrect. Another way of saying this is that the statistical tests will be made at the .05 level of significance. Again, this numerical value is typical of that chosen for similar experiments, but is arbitrary, and is based on how often, on average, an incorrect inference will be allowed to be made.

A. <u>DESIGN</u>

The experimental design chosen was a 3 x 3 Latin square arrangement with each cell assigned more than one observation. The Latin square was blocked by program number (there were three programming assignments) and group assignment (the participating students were initially assigned randomly into one of three arbitrary groups). During the remainder of this paper, these blocks will be designated as programs 1, 2, and 3 and groups A, B, and C. The assignment of programmers to groups helped to facilitate the assignment of individuals to treatments during the experiment. Because of the random nature of the group assignment, no effect was expected due to this grouping. However, it should be noted that all programmers did not follow the same order of applied treatments. Therefore, an ordering effect, if present, would be confounded with any effect of the above grouping.

The independent variable in the experiment was modularity practice. At some time during the experiment, each of the programmers followed each of the three practices for a programming assignment. The three methodologies were designated ML, NG, and MG corresponding to Modular design with Local variables, Non-modular design with Global variables, and Modular design with Global variables. (The remaining possible practice, Non-modular design with Local variables, or NL, is not distinct from NG under the instructions given, and was not employed.) See figures 1, 2, and 3 for the instructions given to the programmers for each modularity practice.

The model referred to above as a Latin square is described by:

$$R_{ijkl} = U + P_i + G_j + M_k + L_{ijk} + e_{ijkl}$$

where

R_{ijkl} = observed value of resource consumption

U = grand mean value of resource consumption

P_i = effect of program i

INSTRUCTIONS FOR TREATMENT ML

For this program, you are to observe the following general guidelines:

1) Construct your program as a set of modules. You should create one procedure for each sub-task which you can identify as part of the program solution. As a general rule, try to structure your program so that no procedure has more than about twenty executable statements.

2) Limit the scope of variables within your program so that all communication of values between procedures is by parameter list. No variables are to be accessed except as local within a procedure or as explicitly passed arguments to the procedure.

Except for the above restrictions, you may construct your program in any way you deem appropriate.

Figure 1: Modular/Local Treatment Instructions

INSTRUCTIONS FOR TREATMENT <u>NG</u>

For this program, you are to observe the following general guidelines:

1) If possible, you are to write your program as one procedure only. If you do use more than one procedure, you should use as few procedures as possible.

2) You must declare all of your variables within the main procedure. No declarations of variables will be allowed inside procedures other than the "OPTIONS(MAIN)."

Except for the above restrictions, you may construct your program in any way you deem appropriate.

Figure 2: Non-modular/Global Treatment Instructions

INSTRUCTIONS FOR TREATMENT <u>MG</u>

For this program, you are to observe the following general guidelines:

1) Construct your program as a set of modules. You should create one procedure for each sub-task which you can identify as part of the program solution. As a general rule, try to structure your program so that no procedure has more than about twenty executable statements.

2) You must declare all of your variables within the main procedure. No declarations of variables will be allowed inside procedures other than the "OPTIONS(MAIN)."

Except for the above restrictions, you may construct your program in any way you deem appropriate.

Figure 3: Modular/Global Treatment Instructions

G_j = effect of grouping j

M_k = effect of modularity practice k

L_{ijk} = the "lack of fit" or "residual" of the model -- the pooled effects of the interactions of P_i, G_j, and M_k

e_{ijkl} = experimental error for individual defined by program i, grouping j, modularity practice k, and observation l .

The program and group effects are assumed to be random effects, while the modularity effect is assumed to be a fixed effect. The Latin square design assumes that all variation in observed values of resource consumption from the overall mean is due to the effects of three factors (program, group, and modularity practice). However, there are assumed to be no interaction effects due to combinations of these main effects (program x group, program x modularity, group x modularity, program x group x modularity). The lack of fit term above represents the pooled effect of these interactions (if present). If shown to be statistically significant, this effect would cast doubt on the adequacy of the Latin square design.

The experimental error e_{ijkl} is assumed to be normally distributed with a mean value of zero. Also, since U gives the overall mean value of resource consumption, it must be the case that

$$P_1 + P_2 + P_3 = 0$$

$$G_A + G_B + G_C = 0$$

and $$M_{ML} + M_{NG} + M_{MG} = 0 \quad .$$

That is, the effects are perturbations to the overall mean.

Finally, it should be noted that the experiment was constructed according to a Latin square design for several pragmatic reasons:

- The Latin square design is well known. The methods of experimentation and analysis for the Latin square have been used often successfully, particularly in agricultural experiments.

- This design allowed a minimal amount of disruption of the course in which the experiment was performed. The cyclic progression of the programmers through the three modularity practices seemed to give them a feeling of "fairness" -- that everyone would eventually have the same advantages and disadvantages.

- The Latin square allowed some measure of control over learning effects within the experiment, as the experimental subjects did not all receive the three modularity treatments in the same order. However, the design did not allow control over the so called "order effects" of the applied treatments -- those effects caused by the differing treatment orders between groups.

B. SUBJECTS

The experimental subjects were enrolled in the course Computer Science 253, Data Structures and Logic. They were generally junior level computer science majors plus a few semi-experienced programmers from other academic departments. Following Schneiderman [2], the level of proficiency of the experimental subjects can be classified by the following scheme:

EXPERIENCE CLASSIFICATION	EXPERIENCE LEVEL
NAIVE	no programming experience
NOVICE	less than one year of experience
INTERMEDIATE	1 to 3 years of experience
EXPERT	more than 3 years of experience

According to this classification scheme, the study described herein employed intermediate programmers.

Though each member of the experiment was enrolled in Computer Science 253, not every member of the class participated. At the beginning of the semester in which the experiment was performed, class members were asked to participate and were given a description of the experiment to be undertaken. Participation was not required and a subject could choose to leave the experiment at any time. Each class member completed the same assignments, but data was collected concerning the activities of the participating programmers. In general, it seemed as though the subjects did not feel inconvenienced by the experimental procedures. The subjects were quite cooperative, and it is believed that the data collected was a good indication of the actual behavior of the subjects.

C. PROCEDURES

Three programming assignments were selected to be used in the experiment, and were typical of the assignments encountered in an undergraduate data structures course. (The assignments involved (1) infix to postfix conversion of expressions, (2) insertion and deletion of records into a binary tree structure, and (3) best-first search of a weighted graph structure.) All assignments were completed using the PL/I programming language. The assignments were of varying difficulty, with an overall average of 126 lines of code (not including comments) produced by the experimental subjects. Further descriptive statistics will be given in Section IV.

Again, Schneiderman provides a classification scheme which can be used to describe the assignments employed in the experiment:

PROGRAM SIZE CLASSIFICATION	LINES OF CODE
SMALL	less than 100 lines of code
MEDIUM	100 to 1000 lines of code
LARGE	1000 to 10,000 lines of code
VERY LARGE	more than 10,000 lines of code

So, the experiment can be said to have used intermediate programmers working on small to medium programs.

It is assumed that the method of design, or modularity practice, is an indicator of the cognitive behavior of the programmers. That is, by placing constraints on the way programs are designed, constraints are placed on the way programmers think. Figure 4 shows the order of treatment assignments (constraints) employed in the experiment.

The dependent variable in the experiment was resource consumption, and several metrics were used as indicators of human and machine costs in software development. In general, there were three main sources of experimental data:

- After each programming assignment, the programmers completed a survey which questioned them about their activities during the completion of the program assignment. (See figure 5.)

- The final listing of the program (the one turned in to

ORDER OF APPLIED TREATMENTS

GROUP:	A	B	C
PROGRAM NUMBER			
1	ML	NG	MG
2	NG	MG	ML
3	MG	ML	NG

Figure 4: Treatment Order

SURVEY

1) Approximately how many hours did you spend on the assignment just completed? ____________

2) Of this time, how many hours did you spend on the design of the program? ____________

3) How many hours did you spend on coding and keying the program? ____________

4) How many hours did you spend on debugging the program?

____

5) Estimate the number of runs you think you made in the course of completing this program. ____________

6) Estimate the number of runs you think you made before all syntax errors were removed. ____________

7) For this particular programming assignment and the group to which you were assigned, what was the most difficult problem that you faced?

____
____
____
____
____

8) Additional comments and/or complaints:

____
____
____
____

Acct. No. ____________________
Name ____________________
Date ____________________
Assignment ____________________

Figure 5: Survey Completed after Each Programming Assignment

be graded) was retained. Several of the employed metrics were based on data taken directly from program printouts.

- When each program was submitted to the computer for execution, the submission was unobtrusively monitored and the time of submission and user number was recorded. In this way, the number of total runs for a particular user for each programming assignment was obtained.

The metrics employed in the experiment were:

ESTRUNS	Estimated number of program submissions, as given by the programmers on the survey following each program.
RUNSWOSYN	Estimated number of submissions made by the programmer after syntax errors were removed. The data for this metric was calculated by subtracting the number of runs estimated to result in syntax errors from ESTRUNS above.
ACTRUNS	The number of program submissions counted by the system for a particular user during the time frame allocated to a particular programming assignment.
HOURS	Estimated total number of hours to complete the programming assignment, as taken from the survey.
DESIGNHRS	Estimated hours spent on design of the program, as taken from the survey.
DEBUGHRS	Estimated hours spent on debugging the program, as taken from the survey.
HRSWOCK	Estimated hours devoted to the assignment less the hours devoted to coding and keying the source code, as taken from the survey.

EXSTMTS	The number of statements executed by the computer in running the programmer's final version of the program. This information was taken from the program listing turned in at the end of the assignment.
LINES	The number of PL/I statements (not including comments) in the final version of the program, as taken from the final listing of the program text.
MINUTES	The number of hundredths of minutes of CPU time used by the final version of the program in execution, as given by the final program listing.
COST	The estimated job cost (compile and go) of the final version of the program in cents, as given by the operating system and printed on the final program listing.

Each metric was analyzed independently according to the methods described in the next subsection.

D. <u>ANALYSIS METHODS</u>

It has been shown previously by McNicholl [12] that resource consumption of individuals in software development is not normally distributed, but rather follows a log-normal distribution. This distribution also fit the data obtained by the authors of this paper. For this reason, before analyzing the collected data, the natural logarithm of each observation was calculated. Performing this transformation causes the transformed data to follow a normal distribution and allows an analysis of variance to be performed.

An analysis of variance was performed for each of the eleven metrics according to standard methods [13]. After showing statistically significant differences in the main effects (program, grouping, modularity), the method of least squares differences was employed to establish differences between individual modularity practices (ML vs. NG, ML vs. MG, NG vs. MG).

Also, two linear contrasts (comparisons) were performed, grouping modularity practice in terms of control-flow modularity (ML,MG vs. NG) and data-flow modularity (ML vs. MG,NG). In the first comparison, the modular control-flow practices are grouped together and contrasted with the non-modular control-flow practice. The second comparison tests differences between the uses of global and local variables. In each case, one type of modularity is ignored while the other is tested. Implicit in this procedure is the assumption that there are no interaction effects between the two types of modularity. But this is an assumption only. The experiment does not allow the two types of modularity to be analyzed independently. The usual descriptive statistics were also calculated, and are given in section IV, along with the results of the above mentioned tests.

It should be noted that in calculating the above statistics, not every raw observation was used. There were two conditions under which data was censored or modified:

- If the final version of a program was not correct (as judged by program output only) then <u>all</u> data for that individual on that programming assignment was censored. Only data arising from a successfully completed assignment was used in the calculation of the above statistics.

- In a few instances, individuals who correctly completed an assignment gave invalid or nonsensical responses to survey questions. It is believed that these invalid answers were due to misunderstandings of the survey questions. These invalid responses caused resource consumption values to be negative or zero. As it is impossible to complete an assignment with negative or zero effort, the observations were replaced and the modified data was used in the statistical analyses. If an observation was less than or equal to zero, it was replaced by the smallest observed feasible value reported by any individual.

Problems of this second type only occurred in metrics using program submissions or hours spent by the programmer as effort indicators. For program submissions, the replacement value used was 1. For any question asking for a number of hours as a response, the value used was 0.25. Both of the above criteria were checked (and values modified or deleted) before the logarithmic transformation was applied.

IV. EXPERIMENTAL RESULTS

This section presents the results of the statistical tests mentioned in Section III. Table I gives overall descriptive statistics for the eleven resource consumption metrics. In Table II, descriptive statistics for the metrics are given for individual cells in the experimental layout. The values reported in Tables I and II have not been logarithmically transformed. However, some data values were modified according to the rules of the previous section before the means were calculated.

Following these tables, the summaries of the analyses of variance for the transformed data are given in Tables III through XIII. Note that the F values are given for both transformed (F_t) and untransformed (F_u) data. The F_u values are provided for reference only, and represent the F values that would be obtained if the raw observations were assumed to be normally distributed. For those metrics which showed significance at the .05 level, the results of the linear contrasts described in Section III are summarized in Table XIV. And lastly, for those same metrics, the results of the least squares difference tests are given in Table XV.

TABLE I

OVERALL DESCRIPTIVE STATISTICS FOR METRICS

Metric	N	Raw Score Mean	Standard Deviation	Min.	Max.
ESTRUNS	142	18.73	12.96	4.00	65.00
RUNSWOSYN	142	13.35	11.73	1.00	58.00
ACTRUNS	142	21.56	15.21	1.00	95.00
HOURS	142	18.49	12.76	3.00	75.00
DESIGNHRS	142	4.78	4.10	0.25	30.00
DEBUGHRS	142	7.64	8.21	0.25	60.00
HRSWOCK	142	14.61	11.54	1.25	66.00
EXSTMTS	142	1689.10	1896.11	221.00	16418.00
LINES	142	125.65	51.36	58.00	348.00
MINUTES	142	17.95	7.52	8.00	54.00
COST	142	13.27	5.01	6.00	34.00

TABLE II

RESOURCE CONSUMPTION MEAN VALUES FOR INDIVIDUAL CELLS

Metric	Program Number	Modularity Practice	Raw Score Mean
ESTRUNS	1	ML	23.64
	1	NG	20.38
	1	MG	28.64
	2	ML	21.22
	2	NG	13.00
	2	MG	13.06
	3	ML	17.86
	3	NG	22.80
	3	MG	12.00
RUNSWOSYN	1	ML	17.14
	1	NG	16.23
	1	MG	20.86
	2	ML	13.17
	2	NG	8.82
	2	MG	9.41
	3	ML	14.21
	3	NG	15.60
	3	MG	8.53
ACTRUNS	1	ML	28.43
	1	NG	27.54
	1	MG	27.86
	2	ML	22.28
	2	NG	16.77
	2	MG	15.71
	3	ML	27.64
	3	NG	16.20
	3	MG	16.53

TABLE II (continued)

RESOURCE CONSUMPTION MEAN VALUES FOR INDIVIDUAL CELLS

Metric	Program Number	Modularity Practice	Raw Score Mean
HOURS	1	ML	33.79
	1	NG	18.35
	1	MG	24.96
	2	ML	19.50
	2	NG	11.73
	2	MG	11.18
	3	ML	22.00
	3	NG	17.60
	3	MG	12.93
DESIGNHRS	1	ML	7.79
	1	NG	3.79
	1	MG	7.64
	2	ML	5.39
	2	NG	3.02
	2	MG	2.76
	3	ML	4.52
	3	NG	5.10
	3	MG	4.23
DEBUGHRS	1	ML	12.50
	1	NG	6.62
	1	MG	10.14
	2	ML	8.69
	2	NG	4.73
	2	MG	4.32
	3	ML	12.18
	3	NG	6.42
	3	MG	5.38

TABLE II (continued)

RESOURCE CONSUMPTION MEAN VALUES FOR INDIVIDUAL CELLS

Metric	Program Number	Modularity Practice	Raw Score Mean
HRSWOCK	1	ML	26.34
	1	NG	14.79
	1	MG	20.14
	2	ML	14.97
	2	NG	8.93
	2	MG	8.44
	3	ML	18.39
	3	NG	13.83
	3	MG	10.48
EXSTMTS	1	ML	3675.
	1	NG	2469.
	1	MG	4159.
	2	ML	491.
	2	NG	375.
	2	MG	430.
	3	ML	1711.
	3	NG	1801.
	3	MG	1514.
LINES	1	ML	187.6
	1	NG	137.4
	1	MG	200.2
	2	ML	101.6
	2	NG	93.6
	2	MG	101.8
	3	ML	133.6
	3	NG	92.2
	3	MG	117.1

TABLE II (continued)

RESOURCE CONSUMPTION MEAN VALUES FOR INDIVIDUAL CELLS

Metric	Program Number	Modularity Practice	Raw Score Mean
MINUTES	1	ML	18.50
	1	NG	19.38
	1	MG	16.21
	2	ML	19.28
	2	NG	16.95
	2	MG	17.35
	3	ML	19.86
	3	NG	18.40
	3	MG	16.13
COST	1	ML	18.79
	1	NG	10.77
	1	MG	16.36
	2	ML	14.33
	2	NG	10.95
	2	MG	11.71
	3	ML	17.29
	3	NG	9.20
	3	MG	11.60

TABLE III

SUMMARY OF ANOVA FOR METRIC ESTRUNS

Source of Variation	SS	df	MS	F_t	F_u
Program	4.951	2	2.476	7.11*	6.08*
Group	2.953	2	1.477	4.24*	6.10*
Modularity	0.738	2	0.369	1.06	0.48
Residual	0.291	2	0.145	0.40	0.67
Error	46.309	133	0.348		
Total	55.497	141			

*Significant at .05 level

TABLE IV

SUMMARY OF ANOVA FOR METRIC RUNSWOSYN

Source of Variation	SS	df	MS	F_t	F_u
Program	6.357	2	3.178	3.83*	5.42*
Group	1.887	2	0.943	1.14	2.41
Modularity	0.825	2	0.412	0.50	0.33
Residual	0.399	2	0.200	0.24	0.37
Error	110.498	133	0.831		
Total	120.120	141			

*Significant at .05 level

TABLE V

SUMMARY OF ANOVA FOR METRIC ACTRUNS

Source of Variation	SS	df	MS	F_t	F_u
Program	4.541	2	2.271	5.35*	5.54*
Group	0.108	2	0.054	0.13	0.50
Modularity	2.564	2	1.282	3.02	2.58
Residual	0.198	2	0.099	0.23	0.86
Error	56.416	133	0.424		
Total	64.089	141			

*Significant at .05 level

TABLE VI

SUMMARY OF ANOVA FOR METRIC HOURS

Source of Variation	SS	df	MS	F_t	F_u
Program	9.836	2	4.918	15.10*	13.03*
Group	1.457	2	0.729	2.24	1.15
Modularity	5.527	2	2.764	8.48*	9.91*
Residual	0.342	2	0.171	0.52	1.47
Error	43.327	133	0.326		
Total	61.245	141			

*Significant at .05 level

TABLE VII

SUMMARY OF ANOVA FOR METRIC DESIGNHRS

Source of Variation	SS	df	MS	F_t	F_u
Program	6.978	2	3.489	5.97*	5.84*
Group	6.761	2	3.381	5.78*	4.23*
Modularity	3.702	2	1.851	3.17*	2.93
Residual	0.547	2	0.273	0.47	0.49
Error	77.776	133	0.585		
Total	96.874	141			

*Significant at .05 level

TABLE VIII

SUMMARY OF ANOVA FOR METRIC DEBUGHRS

Source of Variation	SS	df	MS	F_t	F_u
Program	7.999	2	4.000	4.19*	2.85
Group	1.559	2	0.780	0.82	0.16
Modularity	8.961	2	4.481	4.69*	5.88*
Residual	0.113	2	0.056	0.06	0.85
Error	127.094	133	0.956		
Total	147.093	141			

*Significant at .05 level

TABLE IX

SUMMARY OF ANOVA FOR METRIC HRSWOCK

Source of Variation	SS	df	MS	F_t	F_u
Program	11.287	2	5.643	11.65*	10.09*
Group	1.549	2	0.775	1.60	0.61
Modularity	5.296	2	2.648	5.47*	7.10*
Residual	0.234	2	0.117	0.24	0.95
Error	64.433	133	0.484		
Total	83.774	141			

*Significant at .05 level

TABLE X

SUMMARY OF ANOVA FOR METRIC EXSTMTS

Source of Variation	SS	df	MS	F_t	F_u
Program	101.888	2	50.944	331.42*	53.22*
Group	0.466	2	0.233	1.51	2.11
Modularity	0.907	2	0.454	2.95	1.60
Residual	0.566	2	0.283	1.84	2.25
Error	20.444	133	0.154		
Total	126.165	141			

*Significant at .05 level

TABLE XI

SUMMARY OF ANOVA FOR METRIC LINES

Source of Variation	SS	df	MS	F_t	F_u
Program	7.312	2	3.656	61.29*	54.67*
Group	0.032	2	0.016	0.27	0.70
Modularity	1.732	2	0.866	14.52*	12.44*
Residual	0.480	2	0.240	4.03*	5.10*
Error	7.934	133	0.060		
Total	17.587	141			

*Significant at .05 level

TABLE XII

SUMMARY OF ANOVA FOR METRIC MINUTES

Source of Variation	SS	df	MS	F_t	F_u
Program	0.093	2	0.047	0.28	0.02
Group	0.218	2	0.109	0.66	0.55
Modularity	0.445	2	0.222	1.35	1.40
Residual	0.010	2	0.005	0.03	0.07
Error	21.949	133	0.165		
Total	22.713	141			

TABLE XIII

SUMMARY OF ANOVA FOR METRIC COST

Source of Variation	SS	df	MS	F_t	F_u
Program	1.026	2	0.513	7.06*	7.06*
Group	0.034	2	0.017	0.23	0.25
Modularity	5.037	2	2.519	34.65*	29.71*
Residual	0.758	2	0.379	5.21*	4.99*
Error	9.668	133	0.073		
Total	16.288	141			

*Significant at .05 level

TABLE XIV

SUMMARIES OF LINEAR CONTRASTS

Metric	Contrast	SS	df	F_t	F_u
HOURS	NG,MG vs. ML	5.526	1	16.96*	19.77*
	ML,MG vs. NG	1.283	1	3.94*	5.88*
DESIGNHRS	NG,MG vs. ML	3.036	1	5.19*	4.51*+
	ML,MG vs. NG	2.474	1	4.23*	4.25*+
DEBUGHRS	NG,MG vs. ML	8.909	1	9.32*	11.54*
	ML,MG vs. NG	2.817	1	2.95	4.36*
HRSWOCK	NG,MG vs. ML	5.295	1	10.93*	14.13*
	ML,MG vs. NG	1.259	1	2.60	4.43*

*Significant at .05 level

+Analysis of variance did not show DESIGNHRS significant at .05 level under assumption of normal distribution of raw data.

TABLE XV

SUMMARIES OF LEAST SQUARES DIFFERENCE TESTS

Metric	Treatment	Least Squares Mean	Probability of hypothesis LSMEAN(I) = LSMEAN(J)		
			ML	NG	MG
HOURS	ML	3.028	---		
	NG	2.609	.001	---	
	MG	2.595	.001	.909	---
DESIGNHRS	ML	1.494	---		
	NG	1.096	.013	---	
	MG	1.261	.148	.300	---
DEBUGHRS	ML	1.962	---		
	NG	1.402	.007	---	
	MG	1.442	.012	.843	---
HRSWOCK	ML	2.725	---		
	NG	2.313	.005	---	
	MG	2.303	.005	.944	---

The results described by the preceding tables can be summarized as:

- In several cases, the program factor showed up as significant.

- Also, some metrics showed significant differences due to grouping.

- Only two metrics (LINES and COST) showed significance in the residual terms.

- Four of the remaining metrics showed significant differences in resource consumption due to modularity practice. These were the metrics HOURS, DESIGNHRS, DEBUGHRS, and HRSWOCK.

- For these four metrics, the results of the linear contrasts show that every contrast was significant except for ML,MG vs. NG on metrics DEBUGHRS and HRSWOCK.

- Table XV indicates significant differences between modularity treatments ML and NG on each of the four metrics and shows significant differences between ML and MG on each metric but DESIGNHRS. None of the four metrics showed significant differences between treatments NG and MG.

The following section discusses the importance of these results as they relate to software development.

V. DISCUSSION

A. INTERPRETATION OF RESULTS

In several of the analyses of variance, the program factor shows significant differences between programming assignments. However, this result is not important to the research undertaken, as it only shows that some assignments were more difficult than others. Also, there were significant differences due to the group factor in several cases. Again, it should be noted that the group factor is confounded with any order effects that may be present. As it seems unlikely that ordering effects would occur, the differences seem to be due to differences in programmer skill levels between the groups. Some groups had more proficient programmers than others, and this led to lower resource consumption values. Fortunately, the group factor is controlled for in the experimental design, and does not affect the results obtained.

The residual term shows significance in metrics LINES and COST, indicating that important interactions between the factors program number, grouping, and modularity practice are present. As the Latin square design is inadequate in these cases, the results obtained for the main effects are somewhat suspect. In fact, given that interactions are present, the error term used in the analysis of variance is incorrect. Further analysis of these metrics would require that the analysis of variance be repeated, using the residual term as the error term. However, it was decided that the metrics LINES and COST should be excluded from further analysis, since the Latin square model is not fully appropriate in these cases.

Only four of the remaining nine metrics showed significant differences due to modularity. These four were HOURS, DESIGNHRS, DEBUGHRS, and HRSWOCK. It should be noted that all of the data for these metrics was obtained from the survey. These metrics are somewhat related to each other, as each is an estimate of time spent on a portion of the assignment. The results show that the amount of time spent on the programming assignments (as estimated by the programmers) did vary between the three treatments.

The contrast results (Table XIV) show that control-flow modularity was significant for two metrics (HOURS, DESIGNHRS), while all four metrics showed significant differences due to data-flow modularity. By inspecting Table II, it can be seen that treatment ML tends to give much larger values than the other two treatments. Therefore, results of the control-flow contrast show that for metrics HOURS and DESIGNHRS, the "more modular" treatments are more expensive than the "less modular" treatment. Due again to the much larger values due to treatment ML, the data-flow modularity contrasts show for each of the four metrics that the "more modular" approach (local variables and parameter passing) is more expensive than the "less modular" approach (global variables). For this experiment, it seems that data-flow modularity is a more important factor than control-flow modularity, at least in how resource consumption is affected.

In general, Table XV shows that the significant differences occur because of treatment ML being paired with one of the other treatments. Significant differences between treatments NG and MG did not show up in any of the four metrics. These analyses tend to support the result that data-flow modularity impacts resource consumption more heavily than control-flow modularity.

B. SIGNIFICANCE OF RESULTS

As was mentioned previously, the desirability of modular design is voiced by almost everyone. However, the previously described experimental results indicate that modularity does not always favorably impact resource consumption. For the relatively small programs created within the experiment, there was either no difference in resource consumption between treatments, or increasing modularity served to actually increase consumed resources. This result is important as it establishes a baseline result. However, extending these results to larger programs may or may not be appropriate. Only further experimentation can provide the "correct" answer. Some suggestions for further research are made in the next section.

Metrics which showed significance due to modularity practice were those which were obtained from the surveys completed after each assignment. This information is somewhat subjective, as it is the programmers' perceptions of expended effort. None of the "objective" metrics (those relying on system-generated information) showed significant differences between modularity practices. Though somewhat subjective information (answers to survey questions) has been used as data in this experiment, the attitudes of the participants of the experiment have not been directly taken into account. Figure 6 shows a questionnaire that was given to the participants at the conclusion of the experiment. Table XVI summarizes the responses given in this final survey. Remember that when this survey was filled out, each participant had sampled each modularity practice. The results of the survey indicate:

- MG was felt to be the easiest methodology.
- NG was felt to be the most difficult methodology.
- The subjects were initially most familiar with the practices of treatment ML.
- Almost all participants (96%) felt that the experiment was a useful educational tool.

After you have completed your exam, please answer the following questions: (Those who did not participate in the experiment need not respond.)

1) Which group seemed _easiest_ to program in?

ML ________ (modular, local vars)	NG ________ (non-modular, global vars)	MG ________ (modular, global vars)

2) Which group was _most_ difficult to program in?

ML ________	NG ________	MG ________

3) Which group most closely corresponds to the way in which you _prefer_ to program?

ML ________	NG ________	MG ________

4) Which group most closely corresponds to the way in which you were _taught_ to program?

ML ________	NG ________	MG ________

5) Do you think that _YOU_ learned anything from participating in the experiment?

YES ________ NO ________

COMMENTS ____

____

____

____

____

*** PLEASE DO _NOT_ PUT YOUR NAME ON THIS SURVEY ***

Figure 6: Survey Completed at End of Experiment

TABLE XVI

RESULTS OF FINAL SURVEY OF ATTITUDES

	Percentage of Respondents			
Question	ML	NG	MG	No Answer
1	18%	20%	62%	0%
2	38%	55%	7%	0%
3	31%	8%	59%	2%
4	69%	8%	21%	2%

	YES	NO	No Answer
5	96%	3%	1%

These results indicate that:

- The programmers were initially somewhat familiar with modular design techniques.

- Control-flow modularity was seen as a desirable feature of a program, and to a lesser extent, data-flow modularity was also seen as desirable.

- Not only was a total lack of modularity (NG) seen as undesirable, but it was also viewed as a practice which led to difficulties in programming.

This last result is somewhat contradictory with respect to the experimental results, as treatment NG often led to less resources used than the other two practices.

Why did this disparity of results occur? It appears that, regardless of the attitudes of the programmers, the practices which required greater attention to detail required more total effort. There appears to be a substantial amount of effort that must be extended just to create a "modular" program. The incorporation of parameter passing within a program takes a certain amount of effort. The use of parameters in PL/I requires additional declarations of variables. The use of subroutines within a program requires advance planning relating to what subroutines should accomplish. Planning must also be done to figure out the hierarchical structure of all routines within the program. The various threads of control-flow must all be linked together through the design process.

All of these techniques would serve to increase the values obtained for resource consumption. If modularity is to impact resource consumption in a favorable sense, the effects of modularity on psychological complexity must outweigh the baseline effort needed just to use the modular design techniques. It appears that the programming assignments used in this study were not substantial enough for this to happen.

VI. CONCLUSIONS AND FURTHER RESEARCH DIRECTIONS

In the previous sections of this paper, it has been shown that for intermediate programmers and small to medium sized programs the use of modular design techniques can in fact increase software development costs. This result is quite interesting, as modularity is very widely accepted as a desirable, if not mandatory, feature of any software system. The authors believe that modularity should be present in large systems, as many have already stated. It seems likely that in order to minimize psychological complexity of very large programs, modularity in design is necessary. So, an interesting question arises concerning the size that a program must be before modularity contributes to minimizing costs in the design and development phases. Where is this "break-even" point in software system design? In addition to this question, there are at least three other experiments which could further illuminate the relationship between modularity and resource consumption:

- The same experiment as the one described herein should be replicated, using more substantial programming assignments.

- The same experiment could be replicated, using number of compile-time and run-time errors as metrics of resource consumption.

- A similar experiment could be performed using a factorial layout and treating control-flow and data-flow modularities as individual factors. This would allow some insight to be gained regarding the interaction effects of the two modularities, if any.

These experiments could be easily done in an academic environment, but it seems impractical to expect that such research could ever be done on "real-world" programs, as industry has already been convinced that modular design is necessary from a modifiability standpoint. It is unlikely that any company would allow the development of a large, monolithic software package, especially if modular design were being employed in a separate and independent development group.

BIBLIOGRAPHY

1. Frost, David, "Psychology and Program Design," Datamation 21,5 (May 1975), 137-138.

2. Schneiderman, Ben, Software Psychology, Winthrop, Cambridge, Massachusetts, 1980.

3. Weinberg, Gerald M., The Psychology of Computer Programming, Van Nostrand Reinhold, New York, 1971.

4. Sheil, B. A., "The Psychological Study of Programming," ACM Computing Surveys 13,1 (March 1981), 101-120.

5. Schneiderman, Ben, "Exploratory Experiments in Programmer Behavior," International Journal of Computer and Information Sciences 5,2 (1976), 123-143.

6. Love, Tom, "An Experimental Investigation of the Effect of Program Structure on Program Understanding," ACM SIGPLAN Notices 12,3 (March 1977), 105-113.

7. Basili, Victor R., and Robert W. Reiter, Jr., "A Controlled Experiment Quantitatively Comparing Software Development Approaches," IEEE Transactions on Software Engineering SE-7,3 (May 1981), 299-320.

BIBLIOGRAPHY (continued)

8. Wirth, Niklaus, "Program Development by Stepwise Refinement," Communications of the ACM 14,4 (April 1971), 221-227.

9. Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM 15,12 (December 1972), 1053-1058.

10. Jackson, M. A., _Principles of Program Design_, Academic Press, New York, 1975.

11. Myers, Glenford J., _Reliable Software through Composite Design_, Petrocelli, New York, 1975.

12. McNicholl, Daniel Gerard, _Predictive Modeling of Resource Consumption During the Programming Phase of Software Development_ (Ph.D. Dissertation, University of Missouri-Rolla, 1982).

13. Freund, Rudolf J., and Ramon C. Littell, _SAS for Linear Models: A Guide to the ANOVA and GLM Procedures_, SAS Institute Inc., Cary, North Carolina, 1981.