B. RANDELL, Editor

Exploratory Experimental Studies Comparing Online and Offline Programming Performance

H. Sackman, W. J. Erikson, and E. E. Grant
System Development Corporation
Santa Monica, California

Two exploratory experiments were conducted at System Development Corporation to compare debugging performance of programmers working under conditions of online and offline access to a computer. These are the first known studies that measure programmers' performance under controlled conditions for standard tasks.

Statistically significant results of both experiments indicated faster debugging under online conditions, but perhaps the most important practical finding involves the striking individual differences in programmer performance. Methodological problems encountered in designing and conducting these experiments are described; limitations of the findings are pointed out; hypotheses are presented to account for results; and suggestions are made for further research.

KEY WORDS AND PHRASES: online vs. offline performance, programmer/computer communication, programming experimental-empirical studies, programming cost effectiveness, programming performance, debugging effectiveness, time sharing vs. batch processing, factor analysis application, programmer trainee performance, basic programming knowledge test, experienced programmer study, analysis of variance, programmer individual differences

CR CATEGORIES: 2.40

Introduction

Computer programming is a multibillion dollar industry. Major resources are being expended on the development of new programming languages, new software techniques, and improved means for man-computer communications. As computer power grows and computer hardware costs go down because of the advancing computer technology, the human costs of computer programming continue to rise and one day will probably greatly exceed hardware costs.

This research was sponsored by the Advanced Research Projects Agency Information Processing Techniques Office and was monitored by the Electronic Systems Division, Air Force Systems Command, under contract F 1962867C0004, Information Processing Techniques, with the System Development Corporation.

Amid all these portents of the dominating role that computer programming will play in the emerging computer scene, one would expect that computer programming would be the object of intensive applied scientific study. This is not the case. There is, in fact, an applied scientific *lag* in the study of computer programmers and computer programming—a widening and critical lag that threatens the industry and the profession with the great waste that inevitably accompanies the absence of systematic and established methods and findings and their substitution by anecdotal opinion, vested interests, and provincialism.

The problem of the applied scientific lag in computer programming is strikingly highlighted in the field of online versus offline programming. The spectacular increase in the number of time-shared computing systems over the last few years has raised a critical issue for many, if not most, managers of computing facilities. Should they or should they not convert from a batch-processing operation, or from some other form of noninteractive information processing, to time-shared operations? Spirited controversy has been generated at professional meetings, in the literature, and at grass roots, but virtually no experimental comparisons have been made to test and evaluate these competing alternatives objectively under controlled conditions. Except for related studies by Gold 1967 [4], and by Schatzoff, Tsao, and Wiig 1967 [11], the two experimental studies reported in this paper are, to our knowledge, the first on this central issue to have appeared. They illustrate the problems and pitfalls in doing applied experimental work in computer programming. They spell out some of the key dimensions of the scientific lag in computer programming, and they provide some useful guidelines for future work.

Time-sharing systems, because of requirements for expanded hardware and more extensive software, are generally more expensive than closed-shop systems using the same central computer. Time-sharing advocates think that such systems more than pay for themselves in convenience to the user, in more rapid program development, and in manpower savings. It appears that most programmers who have worked with both time-sharing and closed-shop systems are enthusiastic about the online way of life.

Time sharing, however, has its critics. Their arguments are often directed at the efficiency of time sharing; that is, at how much of the computational power of the machine is actually used for productive data processing as opposed to how much is devoted to relatively nonproductive functions (program swapping, idle time, etc.). These critics (see Patrick 1963 [8], Emerson 1962 [2], and Macdonald 1965 [7]) claim that the efficiency of time-sharing systems is questionable when compared to modern closed-shop methods, or with economical small computers. Since online systems are presumably more expensive than offline systems, there is little justification for their use except in those situations where online access is mandatory for system operations (for example, in realtime command and control systems). Time-sharing advocates respond to these charges by saying that, even if time sharing is more costly with regard to hardware and operating efficiency, the savings in programmer man-hours and in the time required to produce working programs more than offset such increased costs. The critics, however, do not concede this point either. Many believe that programmers grow lazy and adopt careless and inefficient work habits under time sharing. In fact, they claim that instead of improving, programmer performance is likely to deteriorate.

The two exploratory studies summarized here are found in Grant and Sackman 1966 [5] and in Erikson 1966 [3]. The original studies should be consulted for technical details that are beyond the scope of this paper. They were performed by the System Development Corporation for the Advanced Research Projects Agency of the Department of Defense. The first study is concerned with online versus offline debugging performance for a group of twelve experienced programmers (average of seven years' experience). The second investigation involved nine programmer trainees in a comparison of interactive versus noninteractive program debugging. The highlights of each study are discussed in turn, and the composite results are interpreted in the concluding section. For easier reference, the first experiment is described as the "Experienced Programmer" study, and the second as the "Programmer Trainee" study.

The two experiments were conducted using the SDC Time-Sharing System (TSS) under the normal online condition and simulated offline or noninteractive conditions. TSS is a general purpose system (see Schwartz, Coffman, and Weissman 1964 [14], and Schwartz and Weissman 1967 [15]) similar in many respects to the Project MAC system (see Scherr 1966 [12]) at the Massachusetts Institute of Technology. Schwartz 1965 [13] has characterized this class of time-sharing system as providing four important properties to the user: "instantaneous" response, independent operation for each user, essentially simultaneous operation for several users, and general purpose capability.

TSS utilizes an IBM AN/FSQ-32 computer. The following is a general description of its operation. User programs are stored on magnetic tape or in disk file memory. When a user wishes to operate his program, he goes to one of several teletype consoles; these consoles are direct input/output devices to the Q-32. He instructs the computer, through the teletype, to load and activate his program. The system then loads the program either from the disk file or from magnetic tape into active storage (drum memory). All currently operating programs are stored on drum memory and are transferred, one at a time, in turn, into core memory for processing. Under TSS scheduling control, each program is processed for a short amount of time (usually a fraction of a second) and is then replaced in active storage to await its next turn. A program is transferred to core only if it requires processing; otherwise it is passed up for that turn. Thus, a user may spend as much time as he needs thinking about what to do next without wasting the computational time of the machine. Although a time-sharing system processes programs sequentially and discontinuously, it gives users the illusion of simultaneity and continuity because of its high speed.

1. Experienced Programmer Study

1.1 Experimental Design

The design used in this experiment is illustrated in Figure 1.

	Online		*Offline*	
GROUP I	Algebra	(6)	Maze	(6)
GROUP II	Maze	(6)	Algebra	(6)
Totals		(12)		(12)

Fig. 1. Experimental design for the experienced programmer study

The 2 × 2 Latin-square design with repeated measures for this experiment should be interpreted as follows. Two experimental groups were employed with six subjects in each; the two experimental treatments were online and offline program debugging; and the Algebra and Maze problems were the two types of programs that were coded and debugged. Repeated measures were employed in that each subject, serving as his own control, solved one problem task under online conditions and the other under offline conditions. Note in Figure 1 that each of the two program problems appears once, and only once, in each row and column to meet the requirements of the 2 × 2 Latin-square. Subjects were assigned to the two groups at random, and problem order and online/offline order were counterbalanced.

The statistical treatment for this design involves an analysis of variance to test for the significance of mean differences between the online and offline conditions and between the Algebra and Maze problems. There are two analyses of variance, corresponding to the two criterion measures—one for programmer man-hours spent in debugging and the other for central processor time. A leading advantage of the Latin-square design for this experiment is that each analysis of variance incorporates a total of 24 measurements. This configuration permits maximum pooled sample size and high statistical efficiency in the analysis of the results—especially desirable features in view of the small subject samples that were used.

1.2 Method

A number of problems were encountered in the design and conduct of this experiment. Many are illustrative of problems in experimenting with operational computer systems, and many stemmed from lack of experimental precedent in this area. Key problems are described below.

1.2.1 *Online and Offline Conditions.* Defining the online condition posed no problems. Programmers debugging online were simply instructed to use TSS in the normal fashion. All the standard features of the system were available to them for debugging. Defining the offline condition proved more difficult. It was desired to provide a controlled and uniform turnaround time for the offline condition. It was further desired that this turnaround time be short enough so that subjects could be released to their regular jobs and the experiment completed in a reasonable amount of time; on the other hand, the turnaround time had to be long enough to constitute a significant delay. The compromise reached was two hours—considerably shorter than most offline systems and yet long enough so that most of the programmer-subjects complained about the delay.

It was decided to simulate an offline system using TSS and the Q-32 by requiring the programmer to submit a work request to a member of the experimental staff to have his program operated. The work request contained specific instructions from the programmer on the procedures to be followed in running the program—essentially the same approach used in closed-shop computer facilities. Strictly speaking, then, this experiment was a comparison between online and *simulated* offline operations.

Each programmer was required to code his own program using his own logic and to rely on the specificity of the problem requirements for comparable programs. Program coding procedures were independent of debugging conditions; i.e., regardless of the condition imposed for checkout—online or offline—all programmers coded offline. Programmers primarily wrote their programs in JTS (Jovial Time-Sharing—a procedure-oriented language for time sharing).

1.2.2 *Experimental Problems.* Two program problem statements were designed for the experiment. One problem required the subjects to write a program to interpret teletype-inserted, algebraic equations. Each equation involved a single dependent variable. The program was required to compute the value of the dependent variable, given teletype-inserted values for the independent variables, and to check for specific kinds of errors in teletype input. All programmers were referred to a published source (Samelson and Bauer 1960 [10]) for a suggested workable logic to solve the problem. Programs written to solve this problem were referred to as Algebra programs.

The other problem called for writing a program to find the one and only path through a 20×20 cell maze. The programs were required to print out the designators of the cells constituting the path. Each cell was represented as an entry in a 400-item table, and each entry contained information on the directions in which movement was possible from the cell. These programs were referred to as Maze programs.

1.2.3 *Performance Measures.* Debugging time was considered to begin when the programmer had coded and compiled a program with no serious format errors detected by the compiler. Debugging was considered finished when the subject's program was able to process, without errors, a standard set of test inputs. Two basic criterion measures were collected for comparing online and offline debugging—programmer man-hours and central processor (CPU) time.

Man-hours for debugging were actual hours spent on the problem by the programmer (including turnaround time). Hours were carefully recorded by close personal observation of each programmer by the experimental staff in conjunction with a daily time log kept by the subjects. Discrepancies between observed time and reported time were resolved by tactful interviewing. TSS keeps its own accounting records on user activity; these records provided accurate measures of the central processor time used by each subject. The recorded CPU time included program execute time, some system overhead time, and times for dumping the contents of program or system registers.

A variety of additional measures was obtained in the course of the experiment to provide control data, and to obtain additional indices of programmer performance. Control measures included: TSS experience, general programming experience (excluding TSS experience), type of programming language used (JTS or machine language), and the number of computer runs submitted by each subject in the offline condition. Additional programmer performance measures included: man-hours spent on each program until a successful pass was made through the compiler (called coding time), program size in machine instructions, program running time for a successful pass through the test data, and scores on the Basic Programming Knowledge Test (BPKT)—a paper-and-pencil test developed by Berger, et al., 1966 [1] at the University of Southern California.

1.3 Results

1.3.1 *Criterion Performance.* Table I shows the means and standard deviations for the two criterion variables, debug man-hours and CPU time. These raw score values show a consistent and substantial superiority for online debug man-hours, from 50 percent to 300 percent faster than the offline condition. CPU time shows a reverse trend; the offline condition consistently required about 30 percent less CPU time than the online mode. The standard deviations are comparatively large in all cases, reflecting extensive individual differences. Are these results statistically significant with such small samples?

Table II shows three types of analysis of variance applied to the Latin-square experimental design. The first is a straightforward analysis of raw scores. The second is

TABLE I. Experienced Programmer Performance

Debug Man-Hours

	Algebra		Maze	
	Online	Offline	Online	Offline
Mean	34.5	50.2	4.0	12.3
SD	30.5	58.9	4.3	8.7

CPU Time (sec)

	Algebra		Maze	
	Online	Offline	Online	Offline
Mean	1266	907	229	191
SD	473	1067	175	136

TABLE II. Comparative Results of Three Analyses of Variance

Performance measures	Significance levels		
	Raw Scores	Square root	Square root with covariance
1. Debug Man-Hours			
Online vs. Offline	None	.10	.025
Algebra vs. Maze	.025	.001	.10
2. CPU Time			
Online vs. Offline	None	None	None
Algebra vs. Maze	None	.001	.05

TABLE III. Range of Individual Differences in Programming Performance

Performance measure	Poorest score	Best score	Ratio
1. Debug hours Algebra	170	6	28:1
2. Debug hours Maze	26	1	26:1
3. CPU time Algebra (sec)	3075	370	8:1
4. CPU time Maze (sec)	541	50	11:1
5. Code hours Algebra	111	7	16:1
6. Code hours Maze	50	2	25:1
7. Program size Algebra	6137	1050	6:1
8. Program size Maze	3287	651	5:1
9. Run time Algebra (sec)	7.9	1.6	5:1
10. Run time Maze (sec)	8.0	.6	13:1

an analysis of square root transformed scores to obtain more normal distributions. The third is also an analysis of variance on the square root scores but with the covariance associated with programmer coding skill parceled out statistically; that is, individuals were effectively equated on coding skill so that online/offline differences could be tested more directly.

These applications resulted in six analyses of variance (three for each criterion measure) as shown in Table II. The columns in Table II represent the three kinds of analysis of variance; the rows show the two criterion measures. For each analysis of variance, tests for mean differences compared online versus offline performance and Algebra versus Maze differences. The entries in the cells show the level of statistical significance found for these two main effects for each of the six analyses of variance.

The results in Table II reveal key findings for this experiment. The first row shows results for online versus offline performance as measured by debug man-hours. The raw score analysis of variance shows no significant differences. The analysis on square root transformed scores shows a 10 percent level of significance in favor of online performance. The last analysis of variance, with covariance, on square root scores, shows statistically significant differences in favor of the online condition at the .025 level. This progressive trend toward more clearcut mean differences for shorter debug man-hours with online performance reflects the increasing statistical control over individual differences in the three types of analyses. In contrast to debug man hours, no significant trend is indicated for online versus offline conditions for CPU time. If real differences do exist along the lines indicated in Table I for more CPU time in the online mode, these differences were not strong enough to show statistical significance with these small samples and with the large individual differences between programmers, even with the square root and covariance transformations.

The results for Algebra versus Maze differences were not surprising. The Algebra task was obviously a longer and harder problem than the Maze task, as indicated by all the performance measures. The fairly consistent significant differences between Algebra and Maze scores shown in Table II reflect the differential effects of the three tests of analysis of variance, and, in particular, point up the greater sensitivity of the square root transformations over the original raw scores in demonstrating significant problem differences.

1.3.2 *Individual Differences.* The observed ranges of individual differences are listed in Table III for the ten performance variables measured in this study. The ratio between highest and lowest values is also shown.

Table III points up the very large individual differences, typically by an order of magnitude, for most performance variables. To paraphrase a nursery rhyme:

When a programmer is good,
He is very, very good,
But when he is bad,
He is horrid.

The "horrid" portion of the performance frequency distribution is the long tail at the high end, the positively skewed part which shows that one poor performer can consume as much time or cost as 5, 10, or 20 good ones. Validated techniques to detect and weed out these poor performers could result in vast savings in time, effort, and cost.

To obtain further information on these striking individual differences, an exploratory factor analysis was conducted on the intercorrelations of 15 performance and control variables in the experimental data. Coupled with visual inspection of the empirical correlation matrix, the main results were:

a. A substantial performance factor designated as "programming speed," associated with faster coding and debugging, less CPU time, and the use of a higher order language.

b. A well-defined "program economy" factor marked by shorter and faster running programs, associated to some extent with greater programming experience and with the use of machine language rather than higher order language.

This concludes the description of the method and results of the first study. The second study on programmer trainees follows.

2. Programmer Trainee Study

2.1 Experimental Design

A 2 × 2 Latin-square design was also used in this experiment. With this design, as shown in Figure 2, the Sort Routine problem was solved by Group I (consisting of four subjects) in the noninteractive mode and by Group II (consisting of the other five subjects) in the interactive mode. Similarly, the second problem, a Cube Puzzle, was worked by Group I in the interactive mode and by Group II in the noninteractive mode.

	Interactive	*Noninteractive*
GROUP I (4)	Cube Puzzle	Sort Routine
GROUP II (5)	Sort Routine	Cube Puzzle
Total	9 Subjects	

Fig. 2. Experimental design for the programmer trainee study

Analysis of variance was used to test the significance of the differences between the mean values of the two test conditions (interactive and noninteractive) and the two problems. The first (test conditions) was the central experimental inquiry, and the other was of interest from the point of view of control.

2.2 Method

Nine programmer trainees were randomly divided into two groups of four and five each. One group coded and debugged the first problem interactively while the other group did the same problem in a noninteractive mode. The two groups switched computer system type for the second problem. All subjects used Tint (Kennedy 1965 [6]) for both problems. (Tint is a dialect of Jovial that is used interpretively with TSS.)

2.2.1 *Interactive and Noninteractive Conditions.* "Interactive," for this experiment, meant the use of TSS and the Tint language with all of its associated aids. No restrictions in the use of this language were placed upon the subjects.

The noninteractive condition was the same as the interactive except that the subjects were required to quit after every attempted execution. The subjects ran their own programs under close supervision to assure that they were not inadvertently running their jobs in an interactive manner. If a member of the noninteractive group immediately saw his error and if there were no other members of the noninteractive group waiting for a teletype, then, after he quit, he was allowed to log in again without any waiting period. Waiting time for an available console in the noninteractive mode fluctuated greatly but typically involved minutes rather than hours.

2.2.2 *Experimental Problems.* The two experimental tasks were relatively simple problems that were normally given to students by the training staff. The first involved writing a numerical sort routine, and the second required finding the arrangement of four specially marked cubes that met a given condition. The second problem was more difficult than the first, but neither required more than five days of elapsed time for a solution by any subject. The subjects worked at each problem until they were able to produce a correct solution with a run of their program.

2.2.3 *Performance Measures.* CPU time, automatically recorded for each trainee, and programmer man-hours spent debugging the problem, recorded by individual work logs, were the two major measures of performance. Debugging was assumed to begin when a subject logged in for the first time, that is, after he had finished coding his program at his desk and was ready for initial runs to check and test his program.

2.3 Results

2.3.1 *Criterion Performance.* A summary of the results of this experiment is shown in Table IV. Analysis of variance showed the difference between the raw score mean values of debug hours for the interactive and the noninteractive conditions to be significant at the .13 level. The difference between the two experimental conditions for mean values of CPU seconds was significant at the .08 level. In both cases, better performance (faster solutions) was obtained under the interactive mode. In the previous experiment, the use of square root transformed scores and the use of coding hours as a covariate allowed better statistical control over the differences between individual subjects. No such result was found in this experiment.

If each of the subjects could be directly compared to himself as he worked with each of the systems, the problem of matching subjects or subject groups and the need for extensive statistical analysis could be eliminated. Unfortunately, it is not meaningful to have the same subject code and debug the same problem twice; and it is extremely difficult to develop different problems that are at the same level of difficulty. One possible solution to this problem would be to use some measure of problem difficulty as a normalizing factor. It should be recognized that the use of any normalizing factor can introduce problems in analysis and interpretation. It was decided to use one of the more popular of such measures, namely, the number of instructions in the program. CPU time per instruction and debug man-hours per instruction were compared on the two problems for each subject for the interactive and noninteractive conditions. The results showed that the interactive subjects had significantly lower values on both compute seconds per instruction (.01 level) and debug hours per instruction (.06 level).

TABLE IV. Programmer Trainee Performance

Debug Man-Hours

	Sort Routine		*Cube Puzzle*	
	Interactive	*Noninteractive*	*Interactive*	*Noninteractive*
Mean	0.71	4.7	9.2	13.6
SD	0.66	3.5	4.2	7.0

CPU Time (sec)

	Sort routine		*Cube puzzle*	
	Interactive	*Noninteractive*	*Interactive*	*Noninteractive*
Mean	11.1	109.1	290.2	875.3
SD	9.9	65.6	213.0	392.6

2.3.2 *Individual Differences.* One of the key findings of the previous study was that there were large individual differences between programmers. Because of differences in sampling and scale factors, coefficients of variation were computed to compare individual differences in both studies. (The coefficient of variation is expressed as a percentage: it is equal to the standard deviation divided by the mean, multiplied by 100.) The overall results showed that coefficients of variation for debug man-hours and CPU time in this experiment were only 16 percent smaller than coefficients of variation in the experienced programmer study (median values of 66 percent and 82 percent, respectively). These observed differences may be attributable, in part, to the greater difficulty level of the problems in the experienced programmer study, and to the much greater range of programming experience between subjects which tended to magnify individual programmer differences.

In an attempt to determine if there are measures of skill that can be used as a preliminary screening tool to equalize groups, data were gathered on the subject's grades in the SDC programmer training class, and as mentioned earlier, they were also given the Basic Programming Knowledge Test (BPKT). Correlations between all experimental measures, adjusted scores, grades, and the BPKT results were determined. Except for some spurious part-whole correlations, the results showed no consistent correlation between performance measures and the various grades and test scores. The most interesting result of this exploratory analysis, however, was that class grades and BPKT scores showed substantial intercorrelations. This is especially notable when only the first of the two BPKT scores is considered. These correlations ranged between .64 and .83 for Part I of the BPKT; two out of these four correlations are at the 5 percent level and one exceeds the 1 percent level of significance even for these small samples. This implies that the BPKT is measuring the same kinds of skills that are measured in trainee class performance. It should also be noted that neither class grades nor BPKT scores would have provided useful predictions of trainee performance in the test situation that was used in this experiment. This observation may be interpreted three basic ways: first, that the BPKT and class grades are valid and that the problems do not represent general programming tasks; second, that the problems are valid, but that the BPKT and class grades are not indicative of working programmer performance; or third, that interrelations between the BPKT and class grades do in fact exist with respect to programming performance, but that the intercorrelations are only low to moderate, which cannot be detected by the very small samples used in these experiments. The results of these studies are ambiguous with respect to these three hypotheses; further investigation is required to determine whether one or any combination of them will hold.

3. Interpretation

Before drawing any conclusions from the results, consider the scope of the two studies. Each dealt with a small number of subjects—performance measures were marked by large error variance and wide-ranging individual differences, which made statistical inference difficult and risky. The subject skill range was considerable, from programmer trainees in one study to highly experienced research and development programmers in the other. The programming languages included one machine language and two subsets of Jovial, a higher order language. In both experiments TSS served as the online or interactive condition whereas the offline or noninteractive mode had to be simulated on TSS according to specified rules. Only one facility was used for both experiments—TSS. The problems ranged from the conceptually simple tasks administered to the programmer trainees to the much more difficult problems given to the experienced programmers. The representativeness of these problems for programming tasks is unknown. The point of this thumbnail sketch of the two studies is simply to emphasize their tentative, exploratory nature—at best they cover a highly circumscribed set of online and offline programming behaviors.

The interpretation of the results is discussed under three broad areas, corresponding to three leading objectives of these two studies: comparison of online and offline programming performance, analysis of individual differences in programming proficiency, and implications of the methodology and findings for future research.

3.1 Online vs. Offline Programming Performance

On the basis of the concrete results of these experiments, the online conditions resulted in substantially and, by and large, significantly better performance for debug man-hours than the offline conditions. The crucial questions are: to what extent may these results be generalized to other computing facilities; to other programmers; to varying levels of turnaround time; and to other types of programming problems? Provisional answers to these four questions highlight problem areas requiring further research.

The online/offline comparisons were made in a time-shared computing facility in which the online condition was the natural operational mode, whereas offline conditions had to be simulated. It might be argued that in analogous experiments, conducted with a batch-processing facility, with real offline conditions and simulated online conditions, the results might be reversed. One way to neutralize this methodological bias is to conduct an experiment in a hybrid facility that uses both time-sharing and batch-processing procedures on the same computer so that neither has to be simulated. Another approach is to compare facilities matched on type of computer, programming languages, compilers, and other tools for coding and debugging, but different in online and offline operations. It might also be argued that the use of new and different programming languages, methods, and tools might lead to entirely different results.

The generalization of these results to other programmers essentially boils down to the representativeness of the experimental samples with regard to an objective and well-defined criterion population. A universally accepted classification scheme for programmers does not exist, nor are there accepted norms with regard to biographical, educational and job experience data.

In certain respects, the differences between online and offline performance hinge on the length and variability of turnaround time. The critical experimental question is not whether one mode is superior to the other mode, since, all other things equal, offline facilities with long turnaround times consume more elapsed programming time than either online facilities or offline facilities with short turnaround times. The critical comparison is with online versus offline operations that have short response times. The data from the experienced programmer study suggest the possibility that, as offline turnaround time approaches zero, the performance differential between the two modes with regard to debug man-hours tends to disappear. The programmer trainee study, however, tends to refute this hypothesis since the mean performance advantage of the interactive mode was considerably larger than waiting time for computer availability. Other experimental studies need to be conducted to determine whether online systems offer a man-hour performance advantage above and beyond the elimination of turnaround time in converting from offline to online operations.

The last of the four considerations crucial to any generalization of the experimental findings—type of programming problem—presents a baffling obstacle. How does an investigator select a "typical" programming problem or set of problems? No suitable classification of computing systems exists, let alone a classification of types of programs. Scientific versus business, online versus offline, automated versus semiautomated, realtime versus nonrealtime—these and many other tags for computer systems and computer programs are much too gross to provide systematic classification. In the absence of a systematic classification of computer programs with respect to underlying skills, programming techniques and applications, all that can be done is to extend the selection of experimental problems to cover a broader spectrum of programming activity.

In the preceding discussion we have been primarily concerned with consistent findings on debug man-hours for both experiments. The opposite findings in both studies with regard to CPU time require some comment. The results of the programmer trainee study seem to indicate that online programming permits the programmer to solve his problem in a direct, uninterrupted manner, which results not only in less human time but also less CPU time. The programmer does not have to "warm up" and remember his problem in all its details if he has access to the computer whenever he needs it. In contrast, the apparent reduction of CPU time in the experienced programmer study under the offline condition suggests an opposing hypothesis; that is, perhaps there is a deliberate tradeoff, on the part of the programmer, to use more machine time in an exploratory trial-and-error manner in order to reduce his own time and effort in solving his problem. The results of these two studies are ambiguous with respect to these opposing hypotheses. One or both of them may be true to different degrees under different conditions. Then again, perhaps these explanations are too crude to account for complex problem-solving in programming tasks. More definitive research is needed.

3.2 Individual Differences

These studies revealed large individual differences between high and low performers, often by an order of magnitude. It is apparent from the spread of the data that very substantial savings can be effected by successfully detecting low performers. Techniques measuring individual programming skills should be vigorously pursued, tested and evaluated, and developed on a broad front for the growing variety of programming jobs.

These two studies suggest that such paper-and-pencil tests may work best in predicting the performance of programmer trainees and relatively inexperienced programmers. The observed pattern was one of substantive correlations of BPKT test scores with programmer trainee class grades but of no detectable correlation with experienced programmer performance. These tentative findings on our small samples are consistent with internal validation data for the BPKT. The test discriminates best between low experience levels and fails to discriminate significantly among highest experience levels. This situation suggests that general programming skill may dominate early training and initial on-the-job experience, but that such skill is progressively transformed and displaced by more specialized skills with increasing experience.

If programmers show such large performance differences, even larger and more striking differences may be expected in general user performance levels with the advent of information utilities (such as large networks of time-shared computing facilities with a broad range of information services available to the general public). The computer science community has not recognized (let alone faced up to) the problem of anticipating and dealing with very large individual differences in performing tasks involving man-computer communications for the general public.

In an attempt to explain the results of both studies in regard to individual differences and to offer a framework for future analyses of individual differences in programmer skills, a differentiation hypothesis is offered, as follows: when programmers are first exposed to and indoctrinated in the use of computers, and during their early experience with computers, a general factor of programmer proficiency is held to account for a large proportion of observed individual differences. However, with the advent of diversified and extended experience, the general programming skill factor differentiates into separate and relatively independent factors related to specialized experience.

From a broader and longer range perspective, the trend in computer science and technology is toward more diversified computers, programming languages, and computer applications. This general trend toward increasing variety is likely to require an equivalent diversification of human skills to program such systems. A pluralistic hypothesis, such as the suggested differentiation hypothesis, seems more appropriate to anticipate and deal with this type of technological evolution, not only for programmers, but for the general user of computing facilities.

3.3 Future Research

These studies began with a rather straightforward objective—the comparison of online and offline programmer debugging performance under controlled conditions. But in order to deal with the online/offline comparison, it became necessary to consider many other factors related to man-machine performance. For example, it was necessary to look into the characteristics and correlates of individual differences. We had to recognize that there was no objective way to assess the representativeness of the various experimental problems for data processing in general. The results were constrained to a single computing facility normally using online operations. The debugging criterion measures showed relationships with other performance, experience, and control variables that demanded at least preliminary explanations. Programming languages had to be accounted for in the interpretation of the results. The original conception of a direct statistical comparison between online and offline performance had to give way to multivariate statistical analysis in order to interpret the results in a more meaningful context.

In short, our efforts to measure online/offline programming differences in an objective manner were severely constrained by the lack of substantive scientific information on computer programming performance—constrained by the applied scientific lag in computer programming, which brings us back to the opening theme. This lag is not localized to computer programming; it stems from a more fundamental experimental lag in the general study of man-computer communications. The case for this assertion involves a critical analysis of the status and direction of computer science which is beyond the scope of this article; this analysis is presented elsewhere (Sackman 1967 [9]). In view of these various considerations, it is recommended that future experimental comparisons of online and offline programming performance be conducted within the broad framework of programmer performance and not as a simple dichotomy existing in a separate data-processing world of its own. It is far more difficult and laborious to construct a scientific scaffold for the man-machine components and characteristics of programmer performance than it is to try to concentrate exclusively on a rigorous comparison of online and offline programming.

Eight broad areas for further research are indicated:

a. Development of empirical, normative data on computing system performance with respect to type of application, man-machine environment, and types of computer programs in relation to leading tasks in object systems.

b. Comparative experimental studies of computer facility performance, such as online, offline, and hybrid installations, systematically permuted against broad classes of program languages (machine-oriented, procedure-oriented, and problem-oriented languages), and representative classes of programming tasks.

c. Development of cost-effectiveness models for computing facilities, incorporating man and machine elements, with greater emphasis on empirically validated measures of effectiveness and less emphasis on abstract models than has been the case in the past.

d. Programmer job and task analysis based on representative sampling of programmer activities, leading toward the development of empirically validated and updated job classification procedures.

e. Systematic collection, analysis, and evaluation of the empirical characteristics, correlates, and variation associated with individual performance differences for programmers, including analysis of team effectiveness and team differences.

f. Development of a variety of paper-and-pencil tests, such as the Basic Programming Knowledge Test, for assessment of general and specific programmer skills in relation to representative, normative populations.

g. Detailed case histories on the genesis and course of programmer problem-solving, the frequency and nature of human and machine errors in the problem-solving process, the role of machine feedback and reinforcement in programmer behavior, and the delineation of critical programmer decision points in the life cycle of the design, development and installation of computer programs.

h. And finally, integration of the above findings into the broader arena of man-computer communication for the general user.

More powerful applied research on programmer performance, including experimental comparisons of online and offline programming, will require the development in depth of basic concepts and procedures for the field as a whole—a development that can only be achieved by a concerted effort to bridge the scientific gap between knowledge and application.

REFERENCES

1. Berger, Raymond M., et al. Computer personnel selection and criterion development: III. The basic programming knowledge test. U. of S. California, Los Angeles, June 1966.
2. Emerson, Marvin. The "small" computer versus time-shared systems. *Comput. Autom.*, Sept. 1965.
3. Erikson, Warren J. A pilot study of interactive versus noninteractive debugging. TM-3296, System Development Corporation, Santa Monica, Calif., Dec. 13, 1966.
4. Gold, M. M. Methodological for evaluating time-shared computer usage. Doctoral dissertation, Alfred P. Sloan School of Management, M.I.T., 1967.
5. Grant, E. E., and Sackman, H. An exploratory investigation of programmer performance under online and offline conditions. SP-2581, System Development Corp., Santa Monica, Calif., Sept. 2, 1966.
6. Kennedy, Phyllis R. TINT users guide. TM-1933/00/03, Syst. Develop. Corp., Santa Monica, Calif., July. 1965.
7. Macdonald, Neil. A time shared computer system—the disadvantages. *Comput. Autom.* (Sept. 1965).
8. Patrick, R. L. So you want to go online? *Datamation 9*, 10 (Oct. 1963), 25–27.
9. Sackman, H. *Computers, System Science, and Evolving Society*, John Wiley & Sons, New York, (in press) 1967.
10. Samelson, K., and Bauer, F. Sequential formula translation. *Comm. ACM 3* (Feb. 1960), 76–83.
11. Schatzoff, M., Tsao, R., and Wiig, R. An experimental comparison of time sharing and batch processing. *Comm. ACM 10* (May 1967), 261–265.
12. Scherr, A. L. Time sharing measurement. *Datamation 12*, 4 (April 1966), 22–26.
13. Schwartz, J. I. Observations on time shared systems. Proc. ACM 20th Nat. Conf., 1965, pp. 525–542.
14. ——, Coffman, E. G., and Weissman, C. A general purpose time sharing system. Proc. AFIPS 1964 Spring Joint Comp. Conf., Vol. 25, pp. 397–411.
15. ——, and Weissman, C. The SDC time sharing system revisited. Proc. ACM 22nd Nat. Conf., 1967, pp. 263–271.

Scientific Applications

Corrigenda

James C. Howard, "Computer Formulation of the Equations of Motion Using Tensor Notation," *Comm. ACM 10*, 9 (Sept. 1967), 543–548.

Page	*Now reads*	*Should read*
545, Eq. (13)	$\frac{d\mathbf{A}}{\partial x^k} = \left(\frac{\partial A^i}{dx^k} + \cdots\right.$	$\frac{\partial \mathbf{A}}{\partial x^k} = \left(\frac{\partial A^i}{\partial x^k} + \cdots\right.$
545, Eq. (14)	$\left.\cdots A^j \frac{dx^k}{dt}\right) \mathbf{a}_i A_{.k}^{i} \cdots$	$\left.\cdots A^j \frac{dx^k}{dt}\right) \mathbf{a}_i = A_{.k}^{i} \cdots$
545, Eq. (15)	$\frac{\partial \mathbf{A}}{\partial x^k} = \left(\frac{\partial \mathbf{A}^i}{dx^k} - \cdots\right.$	$\frac{\partial \mathbf{A}}{\partial x^k} = \left(\frac{\partial \mathbf{A}_i}{\partial x^k} - \cdots\right.$
547, l.7	the following computer output . . .	the computer output shown in Figure 4 . . .
548, *Ack.*	. . . Howard Tasjian . . .	. . . Howard Tashjian . . .

All the changes above are due to printing errors.

Letters to the Editor—Continued from Page 1

(On Meeting Coverage in CACM)

that the technical sessions rate no more coverage. Were these sessions deficient, though, they shouldn't have been touted "vital." Rather ACM should stop breaking its arm patting its own back and start contemplating why, as a learned society, its sessions fall short of those of the catchall AFIPS, whose claim to learnedness is certainly less.

I had the dishonor of helping select papers for one of the sessions, whose cancellation I subsequently recommended. The session came on anyway; the meeting must go on, and if the field is moribund, well lower your sights and follow it to the grave.

Enough digression to the business of program chairman, let's return to the *Communications*. News is okay; ACM undoubtedly has news and makes it, and it's more convenient to have news bound with the *CACM* than to ship yet another mimeo around to the membership. Still, if we are going to tell the news, let's tell it well. *Science* magazine is a good model, and I think James P. Titus does a comparable job for his section of the *CACM*. But meeting reports are vastly better done by *Science*. One might even do *Science* one better by treating meeting reports in the style of book reviews, full of opinion; this could have a compelling effect on session organization. Whatever the ultimate style may be, the *CACM* should be professional and honest in all departments, and if no one can be found to do a department right, drop it.

M. Douglas McIlroy
Oxford University
Computing Laboratory
45 Banbury Road
Oxford, England

On CR Format

Editor:

I have nothing to add to Dr. T. Gabay's proposal concerning journal formats [Letter to the Editor, *Comm. ACM 10*, 9 (Sept. 1967), 531], only that he missed one more publication, *Computing Reviews*.

Here is my proposal for the *Reviews*:

(1) All sections, e.g. 1.0 General. 4.3 Supervisory Systems, etc., should start at the top of an odd-numbered page. This page should not contain any information from the previous section.

(2) Left-hand margins should be wide enough for punching holes; pages may be perforated for easy tear-out.

(3) Extended running numbers should be used, including the classifying number, also cross references.

All this would permit the collection of each section separately.

Costs might be kept at the same level, if paper of lesser quality than the present were used. Most important with *CR* is a quick access to general information about a given section, and this could most easily be achieved with separate collecting.

Gerald Kaiser
D-7809 Denzlingen
Schwarzwaldstrasse 39
West Germany

Protection of Computer Programs – A Bibliography (Page 64, CACM, this issue)
Reprints Available from ACM Headquarters
Single copies to individuals, free; Single copies to companies, 50¢;
Multiple copies: first 10, 50¢ ea.; next 100, 25¢ ea.; all over 110, 10¢ ea.
All orders must be prepaid.