Factors affecting programmer productivity during application development

by A. J. Thadhani

The effects of good computer services on programmer and project productivity during application program development are examined. Programmers' terminal activity and the nature of terminal work are analyzed. The discussion includes the effects of short response times, programmers' skills, and program complexity on productivity.

The demand for new applications far exceeds the supply capability of the data processing industry,[1] thus creating a large backlog of application programs. Technological advances have significantly reduced the cost of computer hardware. People costs, such as those associated with programming, however, have increased and are far greater than hardware costs. Such costs are a major component of application development today. Current program development processes are labor-intensive and require highly skilled programming expertise. Unless major breakthroughs occur to significantly increase programmer productivity, the shortage of programming skills in this decade will severely restrict the implementation of applications.

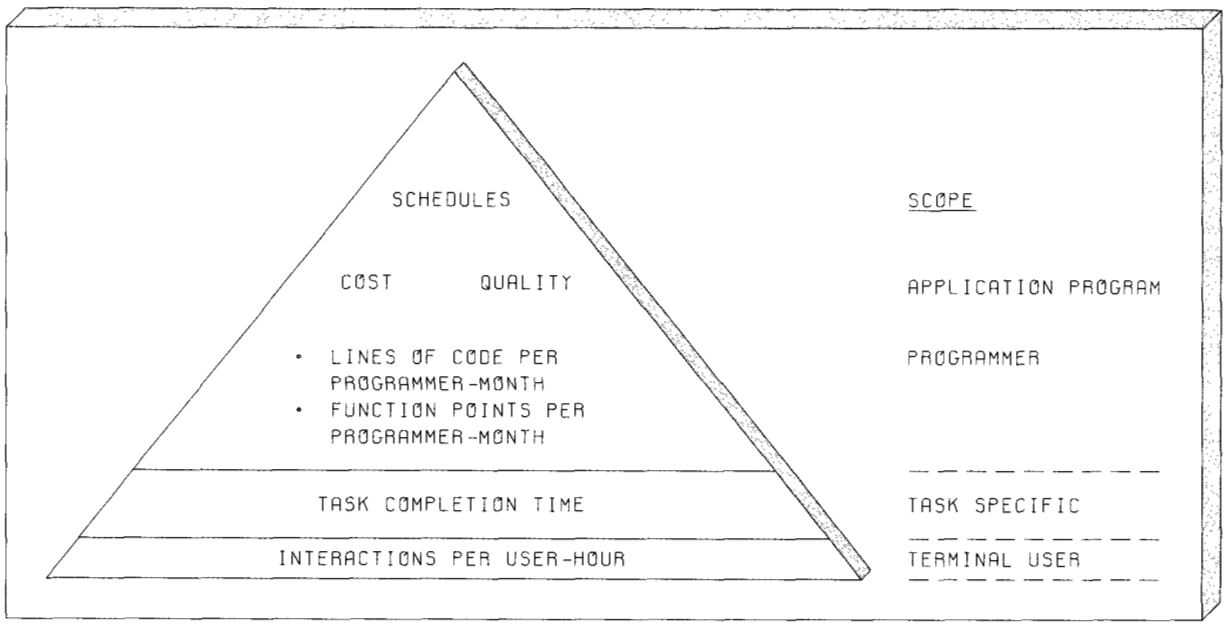
Programming can be considered to be still in its infancy, but the industry offers a wide choice of tools, techniques, and methodologies that can significantly affect the productivity of programmers and programming projects. Moreover, the application development process itself is in a state of flux. There is a search for efficient processes to improve the quality of application programs as well as the productivity of programmers.

Research on ways to improve productivity is focused in two broad areas. The first is to provide tools and techniques that increase productivity within the framework of the conventional program development process. The conventional process consists of the requirements or specification phase, the implementation phase, and the test and installation phase. Specific tools and techniques address productivity improvements in each of these phases. Specification languages, design aids, structured programming techniques, high-level procedural languages, and debugging aids fall into this category.

The second area is to alter the labor-intensive way of implementing application programs. Research in this area includes the use of high-level nonprocedural languages, i.e., the use of application generators, prototype methodologies, and languages. These methods focus on allowing the end user to work with the analyst in creating the application program or a prototype without first creating the specifications. A significant advantage of this process over the conventional one is the reduction in the programming skill level and resources in implementing application programs.

Although the above approach appears promising for long-term improvement in productivity, dramatic and immediate improvement in productivity is possible under the conventional approach. In this paper

Figure 1 Productivity measures



we provide empirical data gathered from an application development project using the conventional development process. The interactive patterns of programmers during the design, code, and unit test phase, abbreviated as DCUT, are analyzed, and the effects of good computer services on programmer productivity and project productivity are discussed.

Productivity measures

Programmers interact with a computer by means of their terminals to accomplish units of work called tasks. Programming requires the completion of one to several thousand tasks. The delivered application program consists of the program code along with documentation describing its method of use and aids to assist in the installation and the maintenance of the program.

In Figure 1 are shown the scope and measures of productivity that have been used in the past. These measures form levels on which the cost and quality of the program are based. A description of each of these measures follows.

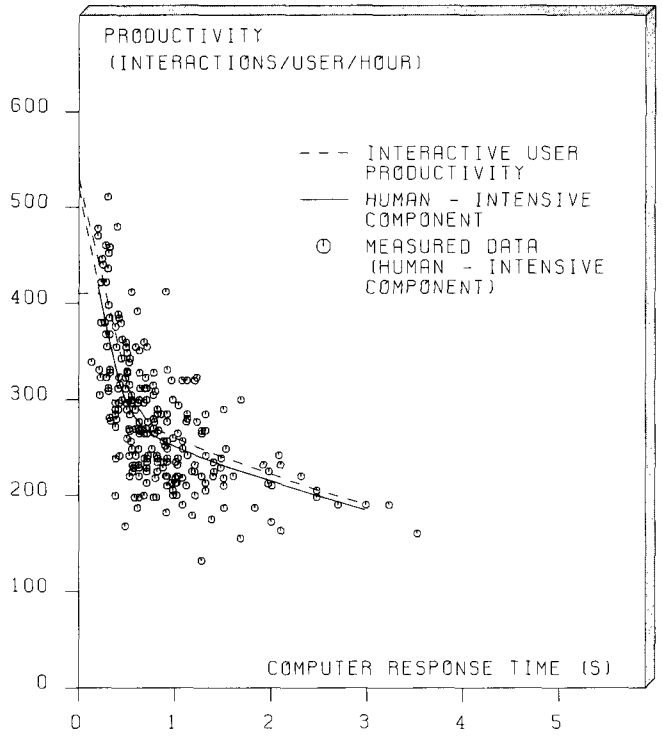
Interactive user productivity. A terminal user's work is defined in terms of the number of interactions between the user and the system. Interactive user productivity, a measure of productivity during the time users interact with the system, has been defined as interactions per user per hour. A prior statistical study has established a strong correlation between system response time and interactive user productivity.[2]

Task completion time. Terminal users interact with the system to accomplish specific tasks defined within a larger project. For example, implementation of a program module may be considered a programming task. At the task level, the time to complete a task is a measure of productivity. In a controlled experiment, task completion time is shown to be related to system response time. Results from the statistical study and the controlled experiment are summarized in a later section.

Lines of code or function points per programmer-month. Productivity at the project level can be measured in terms of shorter schedules, lower cost and development effort, improved quality, or some combination of these factors. Two measures of productivity have been used for programmer productivity—the number of lines of code per programmer-month and function points per programmer-month.[3]

Innumerable factors at the application program level affect productivity. Some of these are software tools, implementation language, modern programming

Figure 2 Interactive user productivity versus computer response time for human-intensive interactions



practices, complexity, and programmer and team capability.[4] In this paper we explore some of these factors. The productivity of six programmers during DCUT is compared, and productivity differences due to skill level are examined. The time spent by a programmer on individual module implementation is examined and proposed as a measure of complexity relative to other modules implemented by the same programmer. These data show that the skill and experience of the development team and the complexity of the program are significant factors affecting programmer and project productivity.

Because of all the factors affecting programmer and project productivity, it is not possible, in general, to conduct a controlled experiment with the quality of computer services being the only difference between two development projects. To show the effects of good computer services on programmer productivity, our methodology focuses instead on the nature of programmers' work and programmers' terminal session times. The trace of four programmers' interactions during DCUT is extensively analyzed. Their activities at the terminal are classified into human-intensive and computer-intensive interactions, and their dependencies on system response times are

Interactive user productivity is the interaction rate between the terminal user and the system.

discussed. But first, we present the results of some prior studies of the effects of response times on terminal users' productivity.

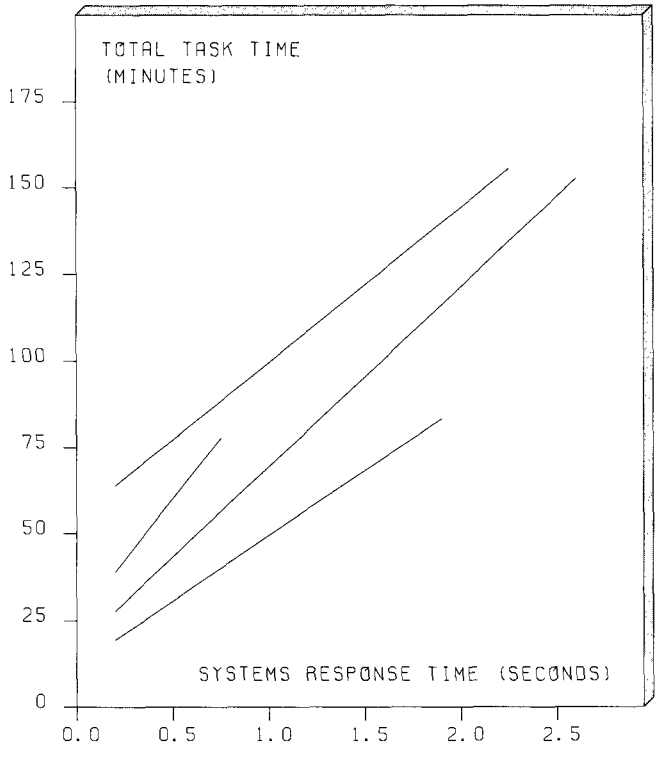
Results of prior studies

Response time and interaction rate. Interactive user productivity on a terminal was shown to be related to the system response time, SRT, in an earlier study.[2] This relationship is shown in Figure 2. Interactive user productivity is the interaction rate between the terminal user and the system and is a measure of productivity for the period when users are actively interacting with the system. We found that 95 percent of all user interactions were human-intensive. Human-intensive interactions consume small amounts of computer resources and complete in a short period of time. Most edit interactions fall into this class. Compiles and executes, examples of computer-intensive interactions, consume larger amounts of computer resources and complete in a longer period of time. These data show that interactive users are twice as productive when the computer response time for human-intensive interactions is 0.25 second instead of 2.0 seconds.[2]

The profound influence that system response time has on user behavior is observed by Doherty and Kelisky:[5]

> "This phenomenon seems to be related to an individual's attention span. The traditional model of a person thinking after each system response appears to be inaccurate. Instead, people seem to have a sequence of actions in mind, contained in a short-term mental memory buffer. Increases in SRT seem to disrupt the

Figure 3 Total elapsed time of users versus response time



thought processes, and this may result in having to rethink the sequence of actions to be continued."

There was a concern about using the count of interactions in a measure of productivity, with no consideration of the complexity or the end results of individual interactions. For example, terminal users may issue few but complex interactions at large response times and many but simple interactions at short response times. In both instances, they achieve the same end result. The study explored computer measures of complexity, e.g., CPU cycles and the number of I/O requests per interaction, and found no significant change in these within the response time range of 0.25 second to 3.0 seconds. These data suggested that the average terminal user did not change the type of interactions; hence, number of interactions per user-hour was an appropriate measure of terminal users' productivity.

Response time and task completion time. These statistical findings have been confirmed in controlled experiments with engineers.[6] A common task was defined for several engineers participating in the study. Their experience level varied from novice to expert user. The interaction rates of all users increased dramatically, particularly for response times under one second. Their elapsed time at the terminal to complete the task as a function of response time is shown in Figure 3.

For all the engineers, irrespective of their level of expertise, it took more than twice as long to complete the task at a response time of 2.0 seconds than at 0.25 second. The data also show that there was no significant difference in the number of interactions to complete the task, and hence, the terminal users did not change the type of interactions within this response time range. The conclusion of the study was that interactions per user-hour and task completion times are related.

Fast response time is just one component of good computer services. We define such services to include 24-hour continuous availability, fast response times to 95 percent of user interactions, response times of less than one minute to small foreground compiles and executions, and turnaround times of less than 15 minutes for small batch executions, including the time to distribute printed output to the user's bin.

The development environment

The development environment included two computers, one for program development and the other for testing. The development computer was an IBM 3031 uni-processor running the Virtual Machine/Conversational Monitor System (VM/CMS). All specification work, design, code, and unit test setup activity was done on the dedicated development machine. The utilization of this machine was intentionally kept below 50 percent to ensure good computer services to the developers. Response times to human-intensive interactions averaged under 0.2 second for more than six months during the period in which we measured the programmers. Response times to computer-intensive interactions like compiles averaged between 10 seconds and 20 seconds, varying as a function of program size and system load.

The test facility was an IBM System/370 Model 158 processor running the Multiple Virtual System operating system with the Time Sharing Option (MVS/TSO). Unit tests were submitted as batch jobs to the test facility. The developers shared this computer and had no preferential treatment over other test

groups. A simple interface allowed programmers to submit jobs from the development machine for execution on the test machine. After execution the results were sent back to the programmer on the development machine. This turnaround time was generally less than 15 minutes. The developer either browsed the results directly at his terminal or requested a hard copy printout for debugging at his desk. Print turnaround times for hard copy output at the programmer's bin averaged between 30 minutes and one hour. However, all developers had access to the computer room if they needed faster access to their printed output.

The quality of computer services is one of several factors that affect programmer and project productivity.

Project results

The delivered application program has 210 000 lines of PL/I code. We define our measure of lines of code in a later section. For this discussion it is sufficient to view a line of code as a unit measure of programmer work output. Implementation, from receipt of the user specifications until shipment of the product, took 16 months. Development programmer effort for PL/I and Application Development Facility (ADF) code was 300 programmer-months. In addition, a 75-person-month effort was spent in test case design and in the writing and execution of test cases for function and system test. Function test and system test were done by the user group. Productivity computed over the entire project, not just during DCUT, was 700 lines of PL/I code per development programmer-month. With the inclusion of the test effort, productivity was 560 lines of code per person-month. These achieved productivity rates are significantly higher than those achieved on similar projects within IBM.[7]

ADF code was not included in the count of lines of code. There were 130 000 lines of ADF code. In addition, there were 160 000 comment lines. Adding these lines to the PL/I code resulted in half a million newly developed program statements. If we include the analysts' and managers' time, the project productivity was 1050 program statements per project person-month. The product, when integrated with prior functions, had approximately three quarters of a million source statements.

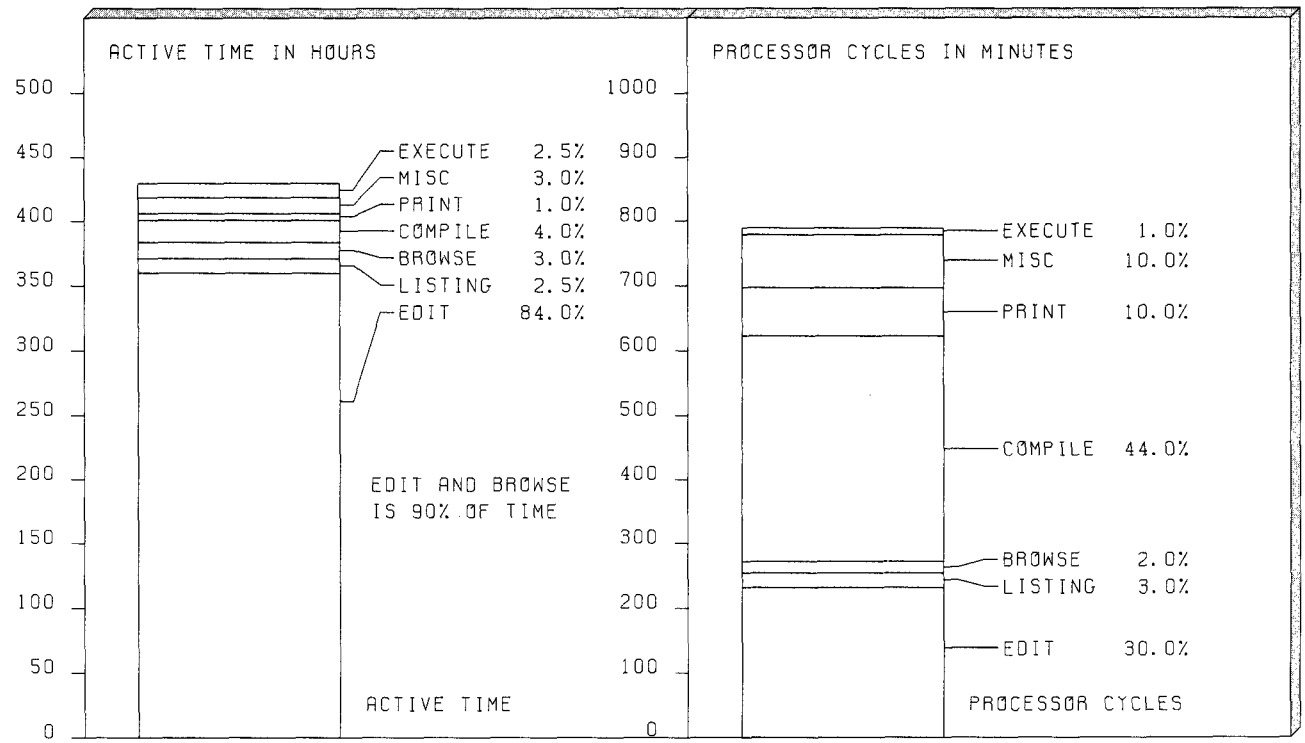
Readers are cautioned against using these absolute numbers for comparison with their own or other projects. There are many pitfalls when such comparisons are made. We use the absolute values only to provide the reader with an insight into the project size.

Costs—dedicated versus central services. The differences in costs of hardware and associated items between the dedicated development computer and central site services were marginal. But the overhead costs for running the dedicated computer facility were significantly lower. The lower overhead resulted because the user population was small and their requirements on the computer facility were not as diverse as those at the central site. For example, since processor utilizations were below 50 percent, no full-time staff was maintained to tune the performance of the system. The data processing staff was small, and third-shift and weekend service was provided with no operator coverage.

Effect of computer services on programmer productivity. As stated earlier, the quality of computer services is one of several factors that affect programmer and project productivity.[4] To isolate and understand the effects of computer services on programmer productivity, the terminal activity of programmers is extensively analyzed. Their terminal workload is characterized in terms of human-intensive and computer-intensive activity. Also, the time they spend at their terminals during DCUT is measured and compared with their other activities.

Classification of programmer active times. The work done at a terminal during DCUT by a programmer called A is classified into seven groups in each of the columns shown in Figure 4. Invocations of any editor such as ESPF, XEDIT, EDGAR, etc. were all classified as edit commands. Invocation of any editor to operate on compiler output listings was classified separately under the category of listing. All system commands and utilities were grouped under miscellaneous. The other classifications are self-explanatory.

Figure 4 Programmer A's active time on terminal and processor cycles consumed



On the one hand, human-intensive work was defined earlier as activities requiring large amounts of human time relative to a small consumption of computer time. Over 90 percent of programmer A's time at the terminal is in the human-intensive functions of edit and browse. Yet the consumption of processor cycles is relatively small. On the other hand, computer-intensive work like compile and print accounts for 55 percent of the processor usage, and these occur during less than five percent of programmer A's time at the terminal.

These characterizations, however, mask the dynamics of user interactions. Examining individual programmer traces, we discovered that compiles were followed immediately by browsing the compiler listing, browsing related modules, editing the source, and recompiling. This process was iterated until success was achieved. The quick compile response time allowed programmers to remain focused on a small set of related program modules without having to fill large response-time gaps with unrelated activities. It has been argued that programmers operating in this mode make fewer mistakes, resulting in superior program quality.

Programmer A spent 430 hours at the terminal. Over half this time was spent on the program modules. The rest of the time was spent in generating test cases, documentation, and other overhead activity. Both types of work are dominated by human-intensive activity.

Three other programmers were extensively analyzed. Their work patterns were similar. Irrespective of the experience level of the programmers as well as the complexity of the functions they implemented, they all spent between 90 and 95 percent of their time at the terminal on human-intensive activities.

Programmer time at the terminal. Each programmer had a terminal which remained connected to the computer for most of the day. Connect time contains many periods of inactivity when the programmer is not interacting with the computer. Thus, it is not an accurate reflection of the time a programmer is actively using the computer. Active time is defined to exclude these inactive periods. A minute of active time excludes all inactive periods greater than one minute and represents time when the programmer is intensely interacting with the computer.

Programmer A's one minute of active time, project overhead time, and nonproject overhead time are shown in Figure 5. Programmers' active times on the VM system were accumulated based on programmer activity. The active times on the TSO system are estimates based on system-accumulated connect times, whereas project and nonproject overhead times are estimates provided by programmers each week.

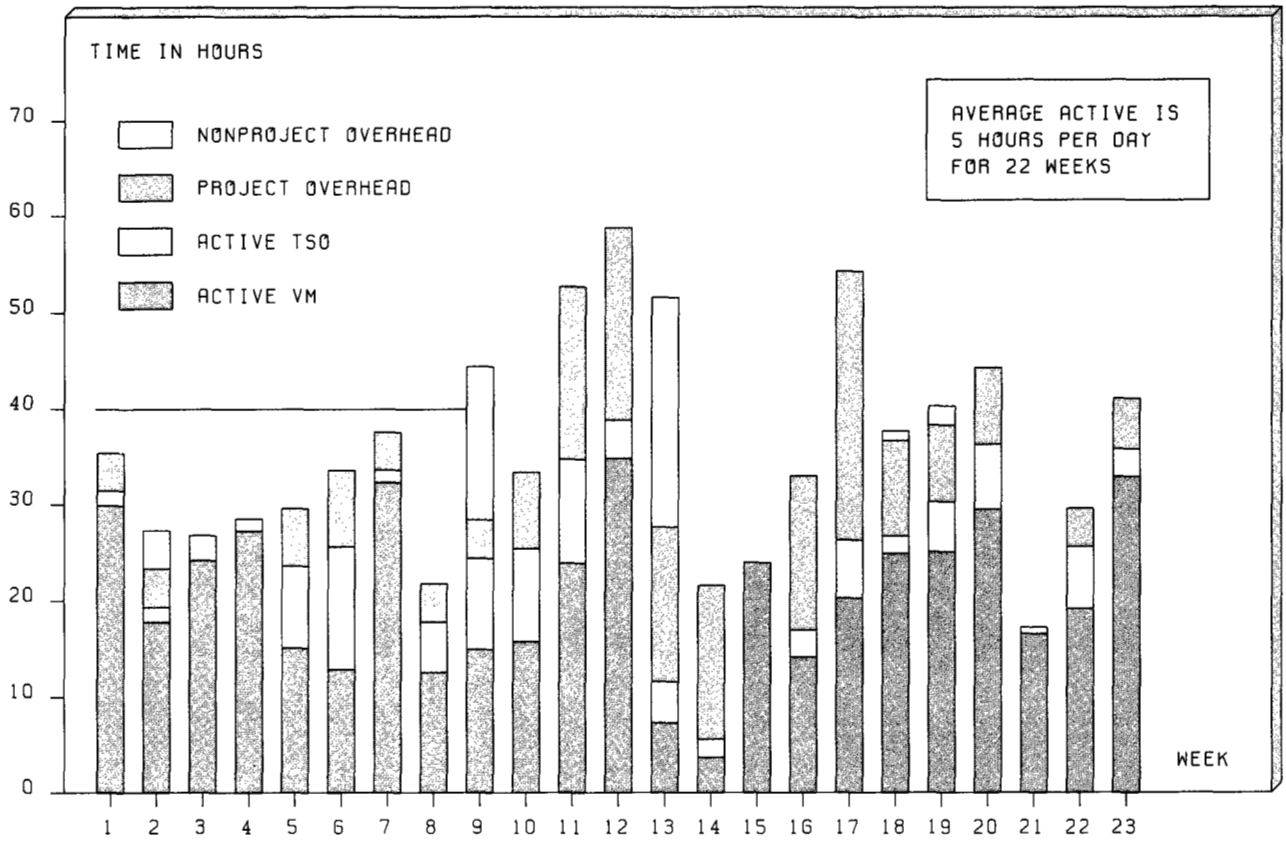
Project overhead consists of the time during which a programmer participates in meetings, walkthroughs, education, travel, and other project-related activities. Other activities might include reviewing another programmer's design, documenting and communicating an idea for review by other programmers, etc. Project overhead should not imply that the activity is not useful. It is classified as overhead only because the programmer cannot be actively generating code, our measure of work output, during this time. Meetings and other project-related activity are the major components of project overhead for programmer A.

Nonproject overhead consists of absence, vacation, and time spent on activities not related to the project. The major component of nonproject overhead for programmer A is vacation and holidays.

Total hours worked per week are shown only for the first two months for programmer A. Accurate overtime estimates were not kept for the remaining weeks for this programmer. During this remaining time, the programmer performs a host of activities, some of which may be design work at his desk, checking output listings, interacting with other programmers, etc.

The 23-week time profile shown in Figure 5 represents the period from when programmer A began

Figure 5 Time spent by Programmer A on terminal and nonterminal activity each week



detailed design from user specifications to the completion of unit test. Activities during this time include

- Functional partitioning
- Detailed design and pseudo code
- Coding in PL/I
- Successful compilation of individual modules
- Creation of unit test cases for individual modules and for the combination of several modules into functions
- Execution of unit test cases and program modification until achievement of success

Specifically not included are function integration and system test. These were done by the user group responsible for test case generation and execution.

The activities of other programmers during the DCUT phase were similar. On average, programmers spent between 20 and 25 hours per week intensely interacting with the computer for extended periods of time.

Programmer perception. Eight programmers were interviewed soon after they completed their design, code, and unit test cycle. Most felt they were significantly more productive in their present environment because of good computer services. They estimated that it would have taken them 60 to 100 percent longer to complete the same work had computer services been poor. One commented, "I am operating at my maximum efficiency"; another, "computer services are so good I cannot recommend any further improvements."

Findings

Programmer productivity and computer services. As mentioned previously, in an earlier study interactive user productivity at a terminal was observed to be two times greater at a response time of 0.25 second than at 2.0 seconds.[2] These statistical findings were confirmed later in controlled experiments in which engineers, irrespective of their level of expertise, spent over twice as long at a terminal to complete the same task when response times were 2.0 seconds instead of 0.25 second.[6]

Our study analyzed programmer workload and time spent at a terminal during the DCUT phase of program implementation for four programmers. Response times for human-intensive interactions were less than 0.25 second. In this environment of good computer services, these programmers spent between 20 and 25 hours per week intensely interacting with the computer for extended periods of time. Furthermore, between 90 and 95 percent of their time at the terminal was in the human-intensive activities of editing and browsing programs and other files. This implies that editor enhancements to provide more efficient user interfaces may significantly improve productivity and should be further investigated.

Project and nonproject overheads are a function of project elapsed time.

If all other factors remained unchanged and the results of prior studies were applied to the four programmers analyzed, it would have taken them twice as long to complete the same task if the response time were two seconds. For programmer A, this might mean that he would take one year instead of six months to do the same work. The programmers perceived that this was in fact the case. Project and nonproject overheads are a function of project elapsed time and do not depend on the progress made by programmers in implementation. The longer the elapsed time to project completion, the larger the project and nonproject overhead.

Note, however, that with the sample of programmers analyzed being small, these conclusions cannot be generalized for all programmers in all phases of program development. Additional analysis of programmer work patterns during DCUT and during the other phases of development such as specifications, function, and system test are areas for further investigation.

Though our quantification was based on response times, the other factors played an important role. We believe that factors contributing to high productivity were system availability, reliability, accessibility, response times, and batch and print turnaround times. During the six months that the system was measured, it was operational 24 hours a day, seven

days a week. On two occasions during prime shift, a system crash occurred because of a power outage, but even then, the system was brought on line in less than two hours. The flexibility of accessing the reliable system at any time, including the weekends, and receiving fast response times and 15-minute batch turnaround times at all times of the day, were all factors affecting programmer productivity.

Project productivity

The evidence provided in prior sections showed that good computer services can improve programmer productivity. However, improving programmer productivity may not improve project productivity if computer services are not the constraining factor in the project. A project is defined as being computer-constrained if on removal of the constraint, that is, by providing good computer services, improved programmer productivity leads to improved project productivity. In the case where computer services is the only constraint, increasing programmer productivity by a factor of two would result in a similar increase in project productivity. That is, the project would complete in one-half the elapsed time.

A noncomputer-constrained project, on the other hand, is one in which project schedules remain unaffected irrespective of the speed at which programmers complete their tasks. Projects with dependencies on external factors, for instance, may not improve project schedules by improving programmer productivity. Depending on the severity of these noncomputer constraints, project schedule reductions would span a range from no improvement to reducing project completion times by one half.

In a recent study, most other factors affecting project productivity were held invariant.[8] The same team of six programmers, completing implementation of the first release of the program, were provided good computer services for the second release. Project effort was in the 20- to 30-person-month range, and implementation took between three and four months. Project productivity was reported to be directly related to response times. Response times were reduced from 2.2 seconds to 0.8 second. Programmer terminal interaction rate increased by 60 percent. Programmer work output, measured in function points per programmer-month, increased linearly by 58 percent. Code quality, measured in trouble reports per function point, improved by over a factor of two at the shorter response time.

In their environment of good computer services, programmers did not have to switch between unrelated tasks. They were able to begin a set of related activities and see it through completion before beginning other activities. Their ability to concentrate on a small set of related activities enhanced their productivity, and they made fewer mistakes.

Programmers' work assignments and their work outputs change during the life of the project.

We did not do a comparative study for reasons cited earlier. The project we report on took 16 months in implementation, with normal turnover of programmers on the team. Moreover, the "experience level" of the team would be significantly different 16 months later for the second release. Instead, we examined how programmer skill and program complexity might affect project productivity.

Productivity index

The number of lines of code, LOC, written by a programmer was selected as the measure of work output. Programmer productivity was computed as LOC per person-month. There has been considerable debate on the usefulness of lines of code as a measure of programmer work output. Function points, a measure of the functions provided by the program, have been proposed as an alternative.[3] Lines of code were used strictly to be compatible with existing records and allow a comparison with other similar projects that have been tracked at the development laboratory.

Two commonly used measures for LOC are

- The executable lines of code, ELOC
- ELOC plus declarative statements plus commas within declares, CLOC

Figure 6 shows a simple example. Notice that comments are not counted in either measure.

Figure 6 Example of lines of code as measure of work output

	COMMENTS	SEMICOLON	COMMA	ACCUMULATED COUNTS CLOC	ACCUMULATED COUNTS ELOC
/* DECLARATION FOR INPUT */	1			0	0
DECLARE					
RECORD CHAR (116),			1	1	0
INPUT FILE INPUT RECORD;		1		2	0
DECLARE REC-COUNT FIXED BIN INIT (0);		1		3	0
DECLARE ERR-COUNT FIXED BIN;		1		4	0
/* HANDLE ERROR CONDITION */	1				
ON ERROR BEGIN;		1		5	1
ERR-COUNT = ERR-COUNT +1;		1		6	2

Normalized time in design, code, and unit test. Programmers' work assignments and their work outputs change during the life of a project. For example, some programmers may be responsible for the specifications only, others for high-level design. Specification documents, design documents, and effective communications with programmers doing coding, not lines of code, are the work output of these programmers. Furthermore, the type of work a programmer does during the project changes. For example, a programmer may initially review specifications, then learn about the tools used on the project, may even develop some new ones, code some modules, and assist the test group to set up function integration and system test.

The productivity over different time periods of a programmer labeled D is shown in Figure 7. Programmer D's assignments included the development of user specifications as well as the design and implementation of common tools for the development project. None of these activities are represented in the 3250 CLOCs of shippable code written by D. Although specifications for D's code were complete in October 1981, D continued to work on the common tools for the project. After spending approximately one month on the design, he began coding in February 1982. In all, he spent 13 weeks exclusively in design, code, and unit test, and his productivity was 1083 CLOCs per month.

Counting the lines of code completed over the project life, which was 16 months, is not very appropriate in comparing productivity rates of individual programmers. Instead, to make productivity comparisons, we select the design, code, and unit test phase, DCUT, during which programmers are doing similar work.

Normalized time is defined as that needed to overcome differences in overtime and nonproject overhead among programmers. Normalized time is computed by taking the total hours worked including overtime, subtracting the nonproject overhead, and then normalizing the amount to a 40-hour week. This results in D spending 15.5 normalized weeks and a productivity of 908 CLOCs per normalized month.

Productivity comparison. Productivity computed using normalized time for six programmers and using both measures for the count of the lines of code is shown in Figure 8. Using two measures provides a broader base from which to make comparisons of programmer productivity. No claim is made about which measure is "best."

The classification of programmers as more "skilled and experienced" is subjective. Some factors such as the number of years as a programmer and prior experience in developing large programs are shown in Figure 8.

Except for programmer D, none of the other programmers wrote their own specifications. Programmer D felt that this was an advantage and had a positive effect on his productivity. In his words, "there is so much underlying the specifications that is not documented, that if I had to implement from a specification written by someone else, my productivity would certainly have been lower."

Programmer B employed labor-saving techniques. Rather than replicate message generation data structures and program code in many modules, B wrote two macros to accomplish this function. We cannot quantify how this affected his productivity. However, the technique does make code easier to modify. Programmer D also made use of these macros in his modules. However, none of the other programmers in this comparison made use of these macros. In fact, after programmer A had completed unit test of his modules, the lead programmer recommended that A's messages, not being in a format consistent with the rest of the group, should be modified—a minor change, in A's words, not a logic problem. However, it required changes to all 40 of A's modules.

The complexity of function and code is yet another factor that can significantly affect programmer productivity.[9] Complexity is discussed in more detail in a later section. For this comparison, a subjective measure of complexity is included. Three categories, complex, C, moderate, M, and simple, S, for the functions assigned to the six programmers are shown in Figure 8. These categories represent the consensus of opinion of three senior project people knowledgeable in the functions being implemented.

The less-experienced programmers are in the 200-ELOC to 600-CLOC per person-month range, whereas the more-experienced are in the 800-ELOC to 1150-CLOC range. The experienced programmers are two to four times more productive than the less-experienced, depending on whether the CLOC or the ELOC measure is used for comparison. These differences would be even greater if the comparison were restricted to functions of equivalent complexity.

Programmer effort and processor resources per CLOC. Four programmers' trace of commands were extensively analyzed. Data for the two less-experienced programmers were averaged together and compared with the average for the two more-skilled and experienced programmers in Figure 9. These comparisons show that to create equivalent amounts of code, the less-experienced programmers spent twice as much time at the terminal and submitted three times more compile and print jobs than the more-experienced programmers. In the process, the

Figure 7 Productivity over different time periods

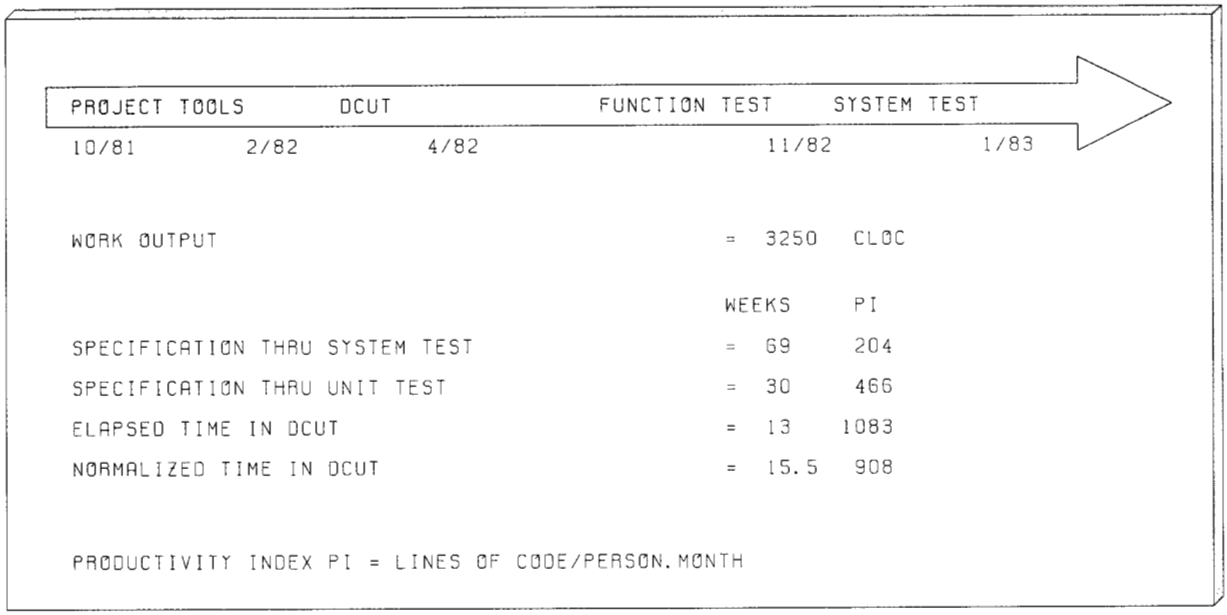


Figure 8 Productivity as lines of code for a normalized month

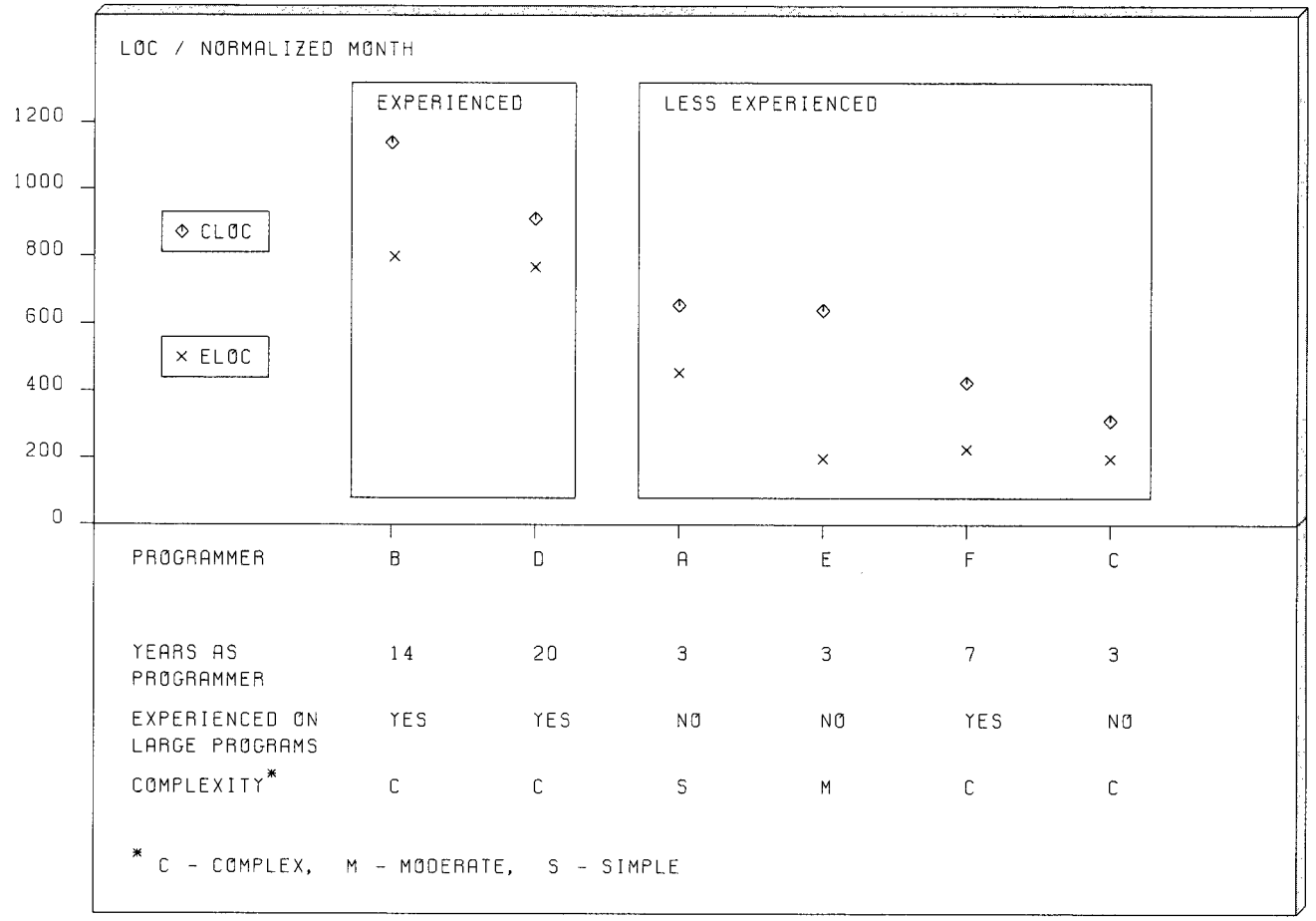
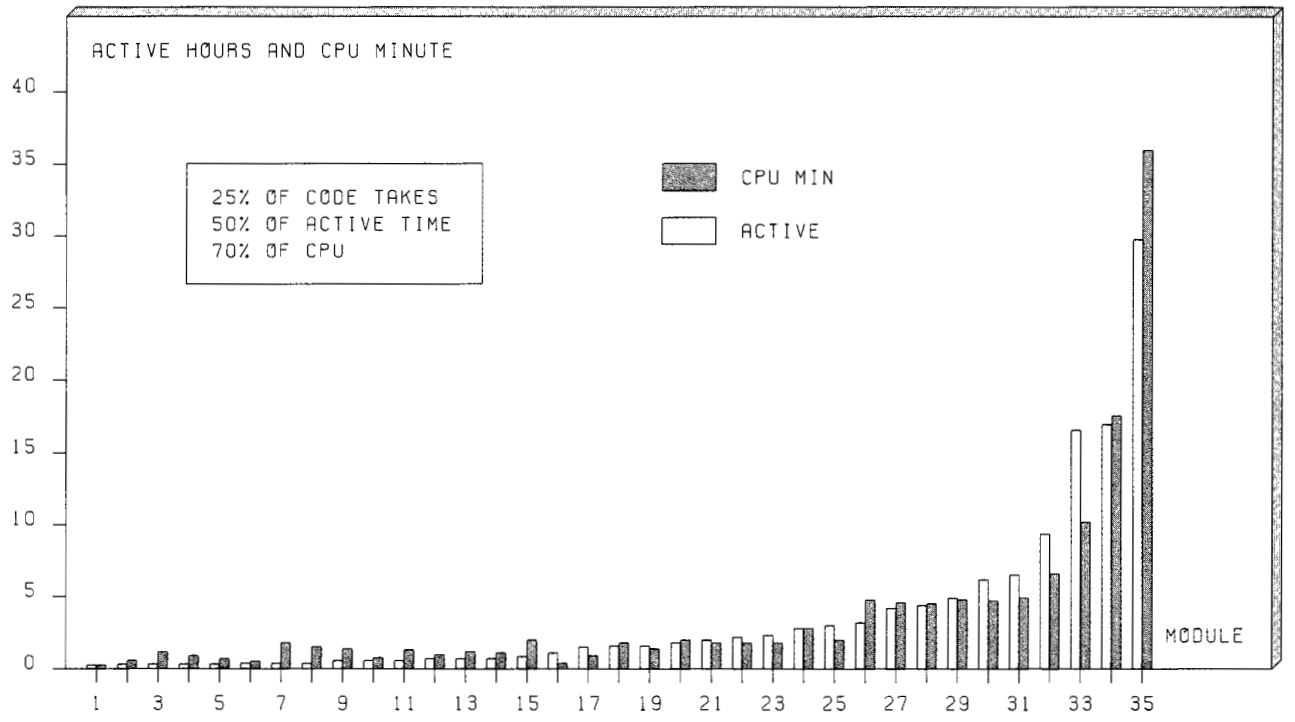


Figure 9 Resource consumption per thousand lines of code for four programmers

	LESS EXPERIENCED	EXPERIENCED	LESS EXPERIENCED ÷ EXPERIENCED
CLOC	3000	3300	1
ACTIVE HOURS	115	60	2
COMPILE	580	173	3.5
EXECUTE	156	15	10
PRINT	370	108	3.5
UNIT TEST	155	54	3
CPU MINUTES	512	180	3

Figure 10 Module statistics for Programmer D; active time in hours and CPU minutes



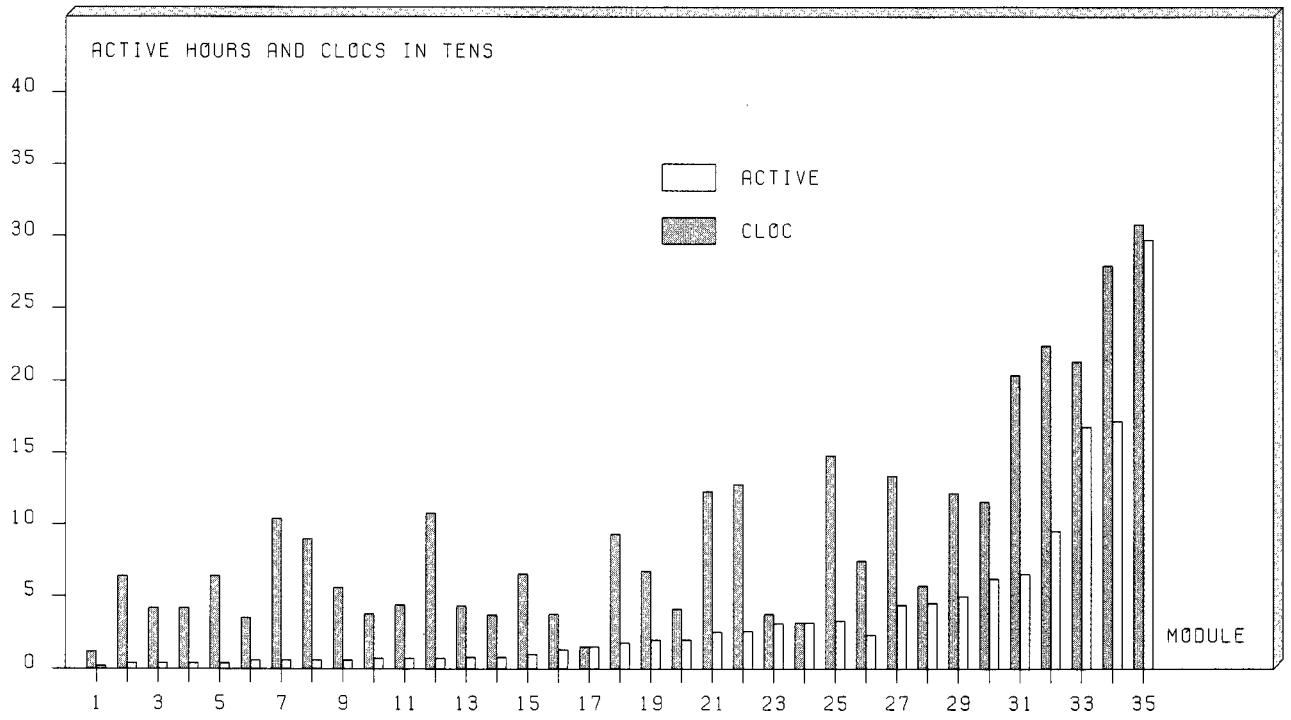
less-experienced programmers consumed three times as many processor cycles. Moreover, each module was individually tested for correctness by the less-experienced programmers before being tested together during unit test. The experienced programmers did not do an individual module test. Instead, they went directly to unit test. This accounts for the difference of a factor of ten in the number of executes. If this extensive testing of individual modules resulted in fewer unit test jobs, it may have been worthwhile to expend the additional effort. But it did not. The less-experienced programmers submitted three times more batch test jobs before achieving success compared to the experienced programmers. Furthermore, there was no discernible difference in code quality measured in trouble reports per line of code between the two groups.

Discussion

Programmer techniques and processes. Experienced programmers were two to four times more productive than the less-experienced. Furthermore, they consumed only one-third of the computer cycles to generate an equivalent number of lines of code. The extensive use of computer resources by the less-experienced programmers may be partly due to the different implementation techniques and processes adopted. For instance, the more-experienced programmers, being confident of the solution, may have completed module implementation before beginning the iterative process of compile and program correction. They stated that this was in fact the case. They may have also employed extensive desk-checking to minimize iterations with the computer. Yet the extensive use of computer resources by the less-experienced programmers may be explained by their use of incremental compile techniques—to let the computer find the bugs, instead of resorting to labor-intensive checking that reduces computer iterations. A study of the differences between these techniques and methods would be useful in understanding the implications for productivity. Furthermore, education to improve programming skills and techniques may have a significant payback in improved productivity of the less-skilled programmers.

Module statistics and programmer effort. The relation between a programmer's effort and computer resources expended in the development of modules

Figure 11 Module statistics for Programmer D; time and module size



is shown in Figure 10. The relationship is intuitively obvious—the longer the programmer spends on a module, the greater the computer resources consumed. However, no linear relation between module size in CLOC and programmer time is observed in Figure 11. In fact, the relationship is quite skewed. Fifty percent of the programmer's time and 70 percent of the processor cycles were expended on four modules, numbers 32 through 35. The four modules comprise 25 percent of the code. In the programmer's words, these modules represent the "heart" of the functions. These modules initialized parameters and invoked submodules in the right sequence for functional correctness and hence were more complex.

Complexity. There are at least two aspects of complexity. The first is intrinsic complexity, which may be defined in terms of the number of parts and the relations and connections between parts. A problem may be more complex if it has more parts, more relations, more interconnections, etc. Software science metrics[9] defines complexity of a program in just such terms, i.e., in terms of the number of operators, operands, and their repetitive use in a program. Moreover, programmer effort is defined to be related to program complexity. The larger the difficulty factor, the longer it takes a programmer to write the program. Furthermore, it is reported that the number of errors in a program is statistically related to program complexity.

There is, however, another aspect of complexity which we call perceived complexity. That is, irrespective of the intrinsic complexity of a problem, different people will perceive the same problem to be either less complex or more complex, depending on their expertise and past experiences. Consider as an example a mathematical problem. A mathematician may perceive the problem to be simple and provide a solution in a short period of time. A person with little background in mathematics may perceive the problem to be difficult and may spend a significantly longer period of time to arrive at a solution. Thus, the same problem requires different amounts of effort and time, depending on the perceived complexity.

One probable explanation for the data shown in Figures 10 and 11 is that the modules that take significantly longer to develop are the ones found to

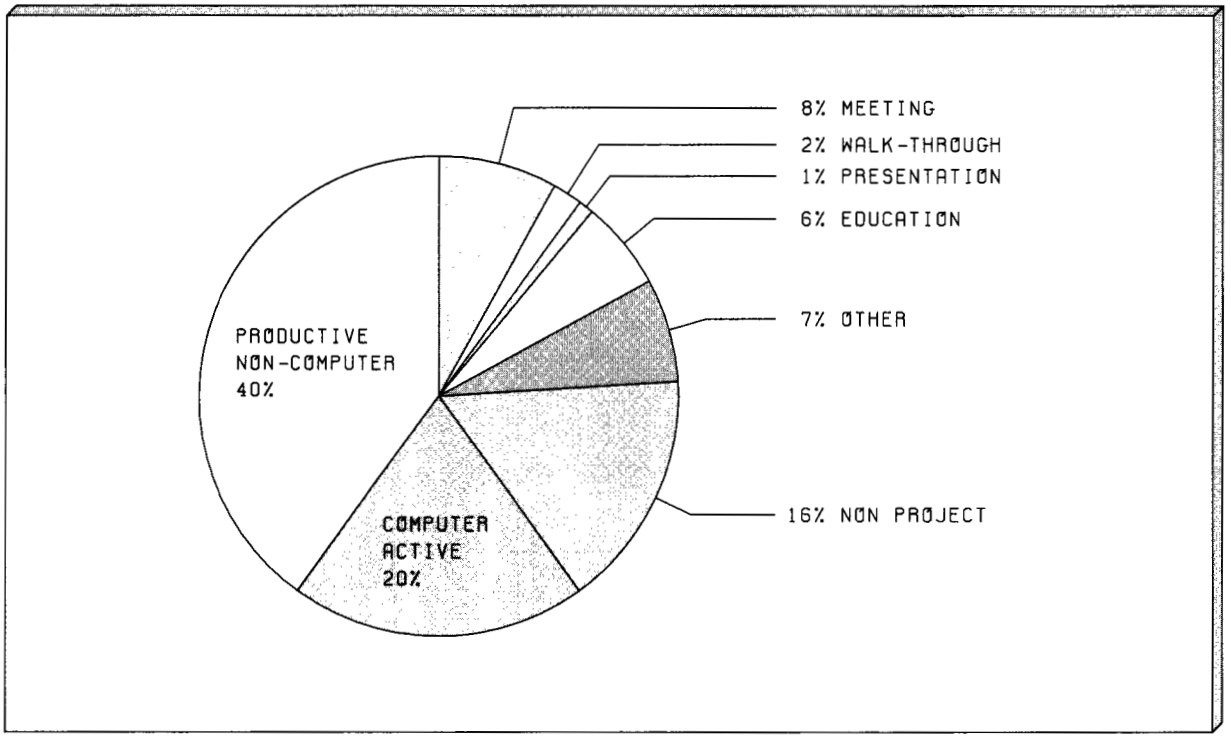
be more complex by the developer, i.e., greater perceived complexity. Furthermore, errors discovered during function test tended to cluster around these modules.

Although the data are limited, we can conclude that the time spent by a programmer developing a module may indicate a level of difficulty experienced by the programmer relative to his other modules and may be a useful indicator of perceived complexity. This measure may be more useful in estimating programmer and project completion time than lines of code. These measures could serve as early warnings for management in tracking the progress of a project. Furthermore, a useful testing strategy would be to test extensively those modules on which a programmer has spent significantly larger amounts of time in their development.

Project time. Weekly data on project and nonproject overheads were collected via an on-line activity collection mechanism. Programmers provided data on their activities and overheads each week. Weekly data on computer active times were collected automatically by the computer system based on actual programmer usage. These data, summarized in Figure 12, show that 40 percent of total time is in project and nonproject overheads. The effective utilization of people on the project is 60 percent. Twenty percent of project time is spent interacting with the computer. These are averages for all project personnel excluding managers on the project. In earlier sections, it was seen that programmers who were doing design, code, and unit test spent between 50 and 60 percent of their time active at the terminal. This activity is in contrast to others on the project

Twenty percent of project time is spent interacting with the computer.

Figure 12 Summarization of project time during design code and unit test for six months



whose usage of and hence dependence on computer services were significantly smaller. Many of the more skilled and experienced programmers were not writing any code. They were responsible for developing project tools, user specifications, high-level design, and consulting with and assisting the less-experienced programmers. Their computer usage was minimal.

These data show that the process employed for software development is quite labor-intensive. Only 20 percent of total project time is spent interacting with the computer. Furthermore, approximately 80 percent of project cost is people-related, and 20 percent is for computer services. With programmer costs escalating and computer costs decreasing, providing on-line computer solutions for many of the current labor-intensive processes should lead to higher development productivity.

Summary

A dedicated computer facility was installed to help in understanding the effects of good computer services on programmer and project productivity. Programmers were provided less than 0.2 second response time to human-intensive interactions like edit commands, between 10 and 20 seconds response time for compiles, and less than 15 minutes of batch turnaround time for unit test jobs, including the time to print the results. System availability was exceptional, and 24-hour continuous service was provided. The hardware and data processing center costs to provide these services were not significantly different from central service charges, since overheads were significantly lower. Project productivity was significantly higher than that achieved on comparable projects in IBM in the past.

Programmers' work patterns and their use of interactive facilities during the design, code, and unit test phase, DCUT, of program development were examined. Programmers spent a significant part of their workday at their terminals, between four and five hours, with no evidence of fatigue. Moreover, their interactive work was dominated by human-intensive work like file edit and browse. During DCUT, 90 to 95 percent of programmer terminal time was human-intensive, with productivity dependent on short response times.[2] Furthermore, 80 to 90 percent of a programmer's terminal time was spent in some editor. Editor enhancements that provide a more efficient interface and significantly improve productivity, as well as additional analysis of programmers' work patterns during DCUT and the other phases of development, are areas for further investigation.

Computer services is one of several factors that affect programmer and project productivity. Because of the innumerable other factors affecting this productivity, it is not possible, in general, to conduct a controlled experiment with the quality of computer services being the only difference between two development projects. Instead, we examined how programmer skill and experience and program complexity might affect productivity.

Experienced programmers were two to four times more productive than the less-experienced. Furthermore, they consumed one-third as many computer cycles to generate an equivalent number of lines of code. The extensive use of computer resources by the less-experienced programmers may be partly due to different implementation techniques and processes adopted, i.e., the use of incremental compile techniques to let the computer find the bugs, instead of labor-intensive checking that reduces computer iterations. A study of the differences between these techniques and methods would be useful in understanding the implications for productivity. Furthermore, education to improve programming skills and techniques may have a significant payback in improved productivity of the less-skilled programmers.

Our limited data suggest that the time spent by a programmer in developing a module may indicate a level of difficulty experienced by the programmer relative to his other modules and may be a useful indicator of perceived complexity. This measure may be more useful in estimating programmer and project completion time than lines of code. These measures could serve as early warnings for management in the tracking of project progress. Furthermore, a useful testing strategy would be to extensively test those modules on which the programmer has spent significantly larger amounts of time.

Finally, we examined how developers spent time on a project. Less than 20 percent of the time was spent interacting with the system. This finding leads us to conclude that there is significant room for further automation of the software development process. With programmer costs escalating and computer costs decreasing, providing on-line computer solutions for many of the current labor-intensive processes should lead to higher development productivity.

Acknowledgments

I am deeply grateful to Norm Pass, the function manager for the project, and Donald Edwards, the project manager, for ideas they contributed during the project and for their support and encouragement throughout the project. I wish to thank several people who have made significant contributions to this project. Matt Korn provided invaluable assistance in the debugging of the system instrumentation package and wrote and maintained several of the data collection and data reduction programs. John Godwin installed and maintained the system instrumentation package on the development system and wrote and maintained programs to collect programmer overhead times. Dave Smith modified the system instrumentation package to run on our system. Roger Wolfe and George Greene wrote several of the analysis programs. Bucky Pope contributed ideas during the project and the writing of this paper. John Bennett reviewed and provided comments on this paper. My thanks also go to R. Parady and J. Orsley for their support and review of this paper, to the programmers who participated in the experiment, and to their managers, K. Shintani and H. Morkner.

Cited references

1. J. Martin, *Application Development Without Programmers*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1982).
2. A. J. Thadhani, "Interactive user productivity," *IBM Systems Journal* **20,** No. 4, 407–423 (1981).
3. A. J. Albrecht, "Measuring application development productivity," SHARE-GUIDE (1979), pp. 83–92.
4. B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1981).
5. W. J. Doherty and R. P. Kelisky, "Managing VM/CMS systems for user effectiveness," *IBM Systems Journal* **18,** No. 1, 143–163 (1979).
6. *The Economic Value of Rapid Response Time*, GE20-0752-0, IBM Corporation; available through IBM branch offices.
7. C. E. Walston and C. P. Felix, "A method of programming measurement and estimation," *IBM Systems Journal* **16,** No. 1, 54–73 (1977).
8. G. N. Lambert, "A comparative study of system response time on program developer productivity," *IBM Systems Journal* **23,** No. 1, 36–43 (1984, this issue).
9. M. H. Halstead, *Elements of Software Science*, Elsevier, New York (1977).

Arvind J. Thadhani *IBM General Products Division, Santa Teresa Laboratory, P.O. Box 50020, San Jose, California 95150.* Mr. Thadhani is senior programmer manager, responsible for information systems strategies at the Santa Teresa Laboratory. He joined IBM in 1968 as a junior engineer in Poughkeepsie, where he was involved in the development of the System/360 Model 155. He then held system design, system analysis, and technical planning positions for large systems. In 1976, he was assigned to the San Jose Research Laboratory, where he was involved in the investigation of storage system architecture. Since 1978, he has been with the General Products Division, working in technical planning, and recently in productivity. Mr. Thadhani received a B.S. in electrical engineering from the Indian Institute of Technology, Bombay, in 1966. He received an M.S. in electrical engineering from Cornell University in 1968 and an M.S. in computer science from the University of Wisconsin in 1972.

Reprint Order No. G321-5207.