Seeking to demonstrate increased programmer productivity, a functional organization of specialists led by a chief programmer has combined and applied known techniques into a unified methodology.

Combined are a program production library, general-to-detail implementation, and structured programming. The overall methodology has been applied to an information storage and retrieval system.

Experimental results suggest significantly increased productivity and decreased system integration difficulties.

Chief programmer team management of production programming

by F. T. Baker

Production programming projects today are often staffed by relatively junior programmers with at most a few years of experience. This condition is primarily the result of the rapid development of the computer and the burgeoning of its applications. Although understandable, such staffing has at least two negative effects on the costs of projects. First, the low average level of experience and knowledge frequently results in less-than-optimum efficiency in programming design, coding, and testing. Concurrently, the more experienced programmers, who have both the insight and knowledge needed to improve this situation, are frequently in second- or third-level management positions where they cannot effectively or economically do the required detailed work of programming.

Another kind of ineffectiveness appears on many projects, which derives from the typical project structure wherein each programmer has complete responsibility for all aspects of one or a small set of modules. This means that, in addition to normal programming activities such as design, coding, and unit testing, the programmer maintains his own decks and listings, punches his own corrections, sets up his own runs, and writes reports on the status of all aspects of his subsystem. Furthermore, since there are few if any guidelines (let alone standards) for doing any of these essentially clerical tasks, the results are highly individual-

ized. This frequently leads to serious problems in subsystem integration, system testing, documentation, and inevitably to a lack of concentration and a general loss of effectiveness throughout the project. Because such clerical work is added to that of programming, more programmers are required for a given size system than would be necessary if the programming and clerical work were separated. There are also many more opportunities for misunderstanding when there is a larger number of interpersonal interfaces. This approach to multiprogrammer projects appears to have evolved naturally, beginning in the days when one-programmer projects were the rule rather than the exception. With the intervening advances in methods and technology, this is not a necessary, desirable, or efficient way to do programming today.

chief programmer teams

H. D. Mills has studied the present large, undifferentiated, and relatively inexperienced team approach to programming projects and suggests that it could be supplemented—perhaps eventually replaced—by a smaller, functionally specialized, and skilled team.[1] The proposed organization is compared with a surgical team in which chief programmers are analogous to chief surgeons, and the chief programmer is supported by a team of specialists (as in a surgical team) whose members assist the chief, rather than write parts of the program independently.

A chief programmer is a senior level programmer who is responsible for the detailed development of a programming system. The chief programmer produces a critical nucleus of the programming system in full, and he specifies and integrates all other programming for the system as well. If the system is sufficiently monolithic in function or small enough, he may produce it entirely.

Permanent members of a team consist of the chief programmer, his backup programmer, and a programming librarian. The backup programmer is also a senior-level programmer. The librarian may be either a programmer technician or a secretary with additional technical training. Depending on the size and character of the system under development, other programmers, analysts, and technicians may be required.

The chief programmer, backup programmer, and librarian produce the central processing capabilities of the system. This programming nucleus includes job control, linkage editing, and some fraction of source-language programming for the system—including the executive and, usually, the data management subsystems.

Specific functional capabilities of the system may be provided by other programmers and integrated into the system by the chief programmer. Functional capabilities might involve very

complex mathematical or logical considerations and require a variety of programmers and other specialists to produce them.

Thus the team organization directly attacks the problems previously described. By organizing the team around a skilled and experienced programmer who performs critical parts of the programming work, better performance can be expected. Also, because of the separation of the clerical and the programming activities, fewer programmers are needed, and the number of interfaces is reduced. The results are more efficient implementation and a more reliable product.

a team experiment

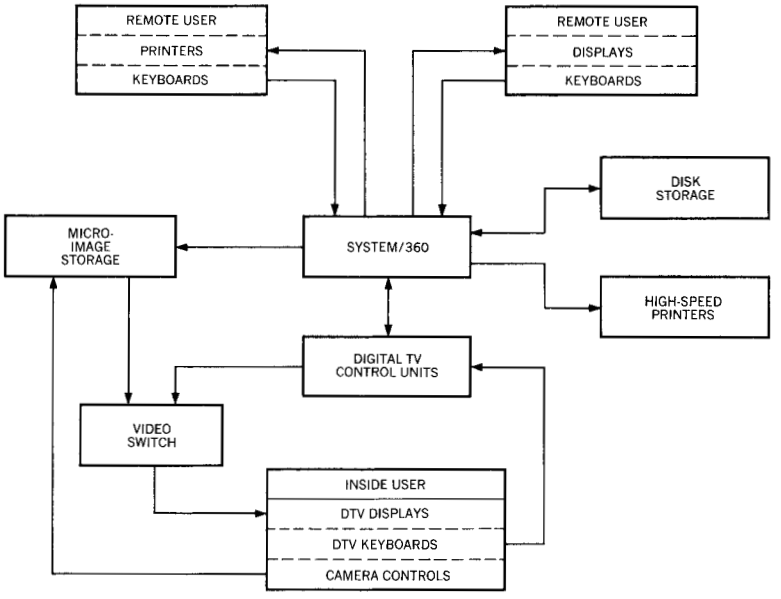
Programming for *The New York Times* information bank was selected as a project suitable for testing the validity of the chief programmer team principles. Since the programming had to interface with non-IBM programs and non-IBM hardware, this experiment involved most of the types of problems generally encountered in large system development. Besides serving as a proving ground for chief programmer team operational techniques, the project sheds light on three key questions bearing on the utility of the approach: (1) Is the team a feasible organization for production programming?, (2) What are the implications of the wide deployment of teams?, and (3) How can a realistic evolution be made? The main theme of this paper is a discussion of these questions. Before beginning, however, we present a technical description of the project, which was performed under a contract between The New York Times Company and the IBM Federal Systems Division.

Information bank system

The heart of the information bank system is a conversational subsystem that uses a data base consisting of indexing data, abstracts, and full articles from *The New York Times* and other periodicals. Although a primary object of the system is to bring the clipping file (morgue) to the editorial staff through terminals, the system may also be made available to remote users. This is a dedicated, time-sharing system that provides document retrieval services to 64 local terminals (IBM 4279/4506 digital TV display subsystems) and up to one hundred twenty remote lines with display or typewriter terminals.

Figure 1 is a diagram of the data flow in the conversational subsystem, which occupies a 200 to 240K byte partition of a System/360 (depending on the remote line configuration) under the System/360 Disk Operating System (DOS/360). Most of the indexing data and all of the system control data are stored on an IBM 2314 disk storage facility. Abstracts of all articles are stored on an IBM 2321. The full text of all articles is photographed and

Figure 1 Conversational subsystem data flow



placed on microfiche, and is accessible to the system through four TV cameras contained in a microfiche retrieval device called the RISAR that was developed by Foto-Mem. A video switch allows the digital TV display consoles to receive either computer-generated character data from the control unit or article images from the RISAR. Users have manual scan and zoom controls to assist in studying articles and can alternate between abstract and article viewing through interaction with the CPU.

Users scan the data base via a thesaurus of all descriptors (index terms) that have been used in indexing the articles. This thesaurus contains complete information about each descriptor, often including scope notes and suggested cross references. Descriptors of interest may be selected and saved for later use in composing an inquiry. Experienced users, who are familiar with the thesaurus, may key in precise descriptors directly. When the descriptor specification is complete, inquirers supply any of the following known bibliographic data that further limits the range of each article in which they are interested:

- Date or date range
- Publication in which the articles appeared
- Sources other than staff reporters from which an article has been prepared
- Types of article (e.g., editorial or obituary)
- Articles with specific types of illustrations (e.g., maps and graphs)

- Section number where an article was published
- Pages (e.g., front-page articles)
- Columns
- Relative importance of the article desired (on an eight-point scale)

Users may further specify their retrieval by combining descriptors that must appear in eligible articles by relating them in AND, OR, and NOT Boolean logic expressions.

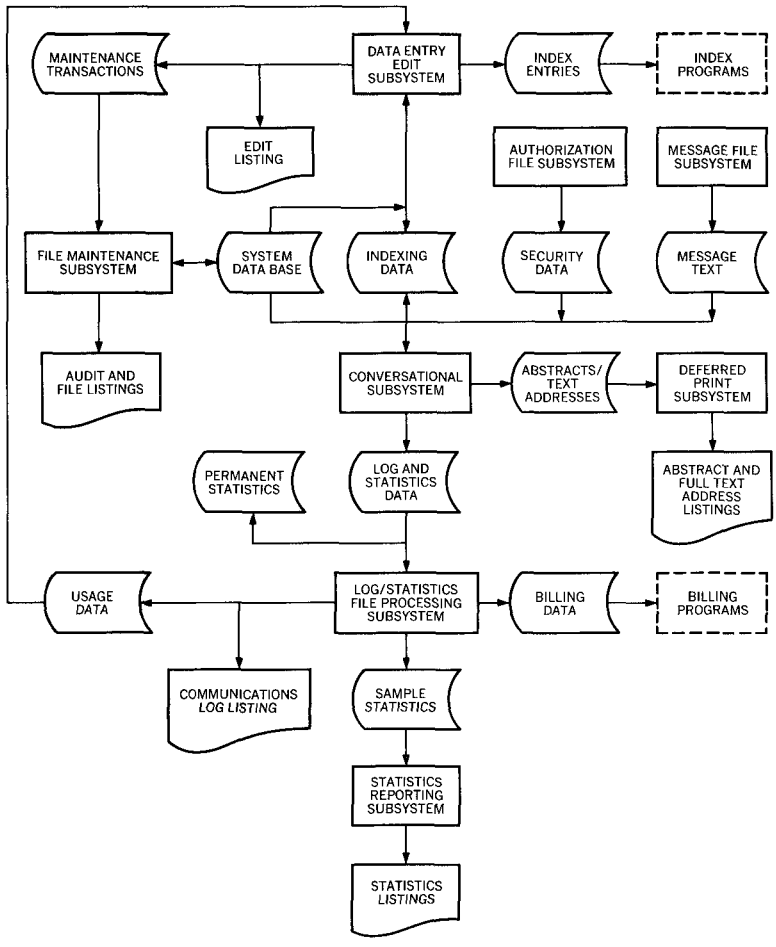
The article search is performed in two phases. An inverted index derives an initial list of articles that satisfy the Boolean inquiry statement. Articles on this list are then looked up in a file of bibliographic data and further culled on the basis of any other specified data. When the search is complete, the inquirer may elect to sort the article references into ascending or descending chronological order before he begins viewing.

Because there are only four cameras available in the RISAR, the system limits article viewing to reduce contention. Thus the inquirer views abstracts of the retrieved articles and selects the most relevant ones for full viewing when a camera becomes available. Inquirers may also request hard copies of specified abstracts and articles. Remote users cannot view the full articles directly. The references in displayed abstracts, however, identify the corresponding articles for off-line retrieval from other sources or through the mail.

A few other significant features of the conversational subsystem may be of interest. It incorporates several authorization features that inhibit unauthorized access to the system and fulfill the conditions of copyright law and other legal agreements. Inquirers who need assistance may key a special code and be placed in keyboard communication with an expert on system files and operations. This expert may also broadcast messages of general interest to all users. Several priority categories exist to allocate resources to inquirers and to control response time. In addition to inquirer facilities, the conversational subsystem allows indexers using the digital TV terminals to compose and edit indexing data for articles being entered into the system data base.

Figure 2 shows the relationship of the conversational subsystem to the supporting subsystems. The indexing data previously mentioned is processed by the data entry edit subsystem and produces transactions for entering data into or modifying the system files. Also produced is a separate set of transactions for preparing a published index. The file maintenance subsystem modifies the six interrelated files that constitute the system data base, and also prepares file backups. Security data used by the conversational subsystem to identify users and determine their

Figure 2 Information bank system



authority is prepared by the authorization file subsystem. The conversational subsystem interacts with users by presenting messages on one of three levels ranging from concise to tutorial, and the message file subsystem prepares and maintains the message file. During operation of the conversational subsystem, users may request hard copy of abstracts and/or articles. The abstracts and the microfiche addresses of the designated articles are printed by the deferred print subsystem. The conversational subsystem also transmits a variety of data on its operation to the log/statistics file, and the corresponding subsystem. A log containing a summary of operations is printed. Billing data for subscribers are passed to billing programs written by *The Times*. Usage data are passed back to be added to the data base. Usage statistics are passed to the statistics reporting subsystem, which produces detailed reports on overall system usage, descriptor (index term) usage, abstract usage, and full article usage.

Team organization and methodology

The methods discussed in this paper have been individually tried in other projects. What we have done is to integrate, consistently apply, and evaluate the following four programming management techniques that constitute the methodology of chief programmer teams:

- Functional organization
- Program production library
- Top-down programming
- Structured programming

functional organization

Since our contracts have more legal, financial, administrative, and reporting requirements associated with them than internal projects of corresponding size, a project manager coordinates these activities in all except the smallest contracts. Administrative and technical problems are jointly handled by the chief programmer and the project manager, thereby permitting the team and especially the chief programmer to concentrate on the technical aspects of the project.

A functional organization also segregates the creative from the clerical work of programming. Because the clerical work is similar in all programming projects, standard procedures can be easily created so that a secretary performs the duties of program maintenance and computer scheduling.

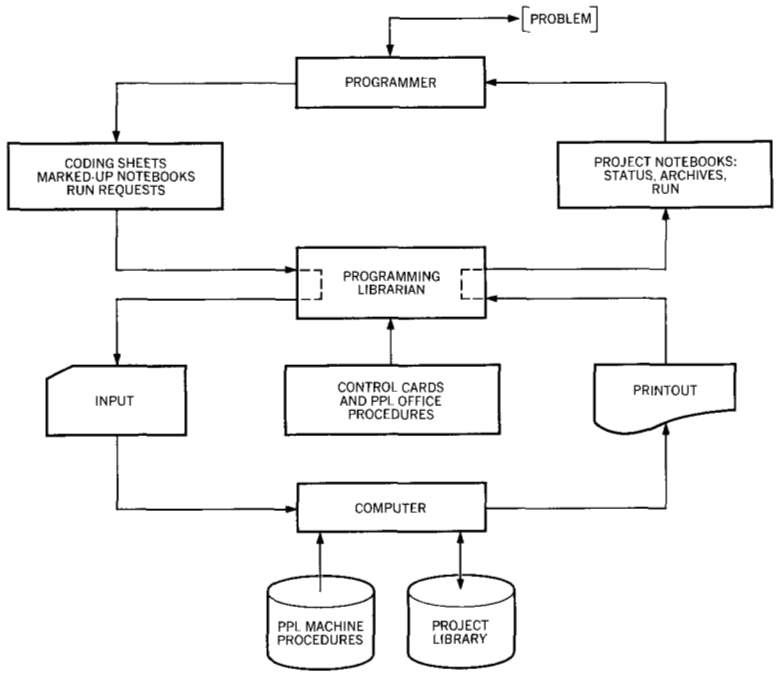
program production library

We have developed a program library system to isolate clerical work from programming and thereby enhance programmer productivity. The system currently in use is the Programming Production Library (PPL). The PPL, shown in Figure 3, includes both machine and office procedures for defining the clerical duties of a programming project. The PPL procedures promote efficiency and visibility during the program development stages.

The PPL comprises four parts. The machine-readable *internal library* is a group of sublibraries, each of which is a data set containing all current project programming data. These data may be source code, relocatable modules, linkage-editing statements, object modules, job control statements, or test information. The status of the internal library is reflected in the human-readable *external library* binders that contain current listings of all library members and archives consisting of recently superseded listings. The *machine procedures* consist of standard computer steps for such procedures as the following:

- Updating libraries
- Retrieving modules for compilations and storing results
- Linkage editing of jobs and test runs

Figure 3 Programming production library



- Backing up and restoring libraries
- Producing library status listings

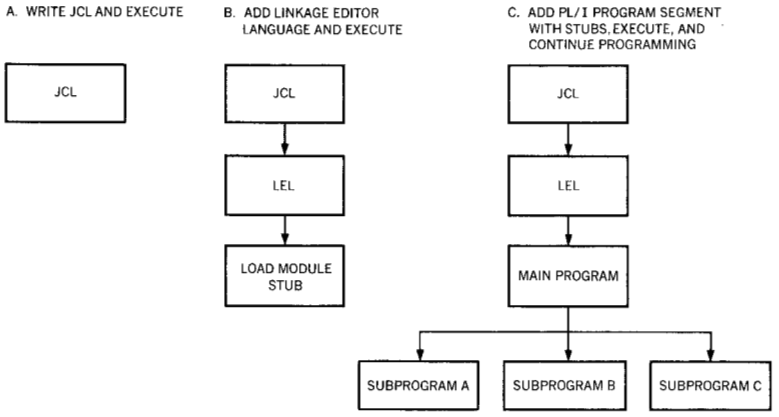
Office procedures are clerical rules used by librarians to perform the following duties:

- Accepting directions marked in the external library
- Using machine procedures
- Filing updated status listings in the external library
- Filing and replacing pages in the archives

A programmer using the PPL works only with the external library. Using standard conventions, he enters directly into the external library binders the changes to be made or work to be done. He then gives these changes to the librarian. Later he receives the updated external library binders, which reflect the new status of the internal library. The external library is always current and is organized to facilitate use by programmers. A chronological history of recent runs contained in the archive binders is retained to assist in disaster recovery. The programmers are thus freed from handling decks, filing listings, keypunching, and spending unnecessary time in the machine area.

The PPL procedures are similar to other library maintenance systems and consist solely of Job Control Language (JCL) state-

Figure 4 Top-down system development



ments and standard utility control statements. By combining standard machine procedures, standard office procedures, and project libraries, the trained librarians provide a versatile programming service that allows a team to make more effective use of its time. The PPL also assists in improving productivity and quality by providing visibility of the work, thereby allowing team members to be aware of the status of modules that they are integrating. Such visibility also permits members to be certain of interface requirements. The internal working languages of a team are the code and statements in the libraries, rather than a separate set of documents that lag behind actual status. Programmers read each other's code in order to communicate definitions, interfaces, and details of operation. Only when a question arises that cannot be resolved by reading code, is it necessary to consult another programmer directly.

top-down programming

The third technique implemented and tested is that of top-down programming. Although most programming system design is done from the top down, most implementations are done from the bottom up. That is, units are typically written and integrated into subsystems that are in turn integrated at higher and higher levels into the final system. The top-down approach inverts the order of the development process. Figure 4 depicts the essence of the top-down approach. Following system design, all JCL and link-edit statements are written together with a base system. The second-level modules are then written while the base system is being checked out with dummy second-level modules and dummy files where necessary. Third-level modules are then written while the second-level modules are being integrated with the base system. This development cycle is repeated for as many levels as necessary. Even within a module, the top-down approach is used by writing and running a nucleus of control code

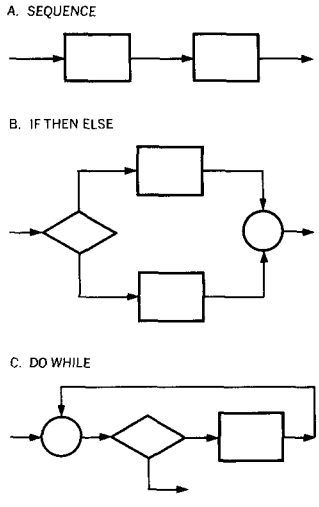
first. Then functional code is added to the control code in an incremental fashion.

structured programming

Structured programming, also used in the information bank project, is a method of programming according to a set of rules that enhance a program's readability and maintainability. The rules are a consequence of a structure theorem in computer science described by Böhm and Jacopini.[2] The rules state that any proper program—a program with one entry and one exit—can be written using only the following programming progressions that are also illustrated in Figure 5.

A. Sequence
B. IF THEN ELSE
C. DO WHILE

Figure 5 Structured programming



Although these rules may seem restrictive and may require a programmer to exercise more thought when first using them, several advantages ensue. With the elimination of GO TOs, one can read a program from top to bottom with no jumps and one can see at a glance the conditions required for modifying a block of code. For the same reason, tests are easier to specify. Further, the rules assist in allowing a program unit to be written using the top-down approach by writing control statements first and then function statements. The use of CALLs to dummy subroutines or INCLUDEs of empty members permits compilation and debugging at a much earlier stage of programming. Finally, if meaningful identifiers are used, a program becomes self-documenting and the need for lengthy comments and flow charts is reduced.

Conventions to support the use of structured programming are required. A set of rules has been developed to format source code so that indentation corresponds to logical depth. If extensive change is necessary, a program is available to reformat the source code.[3] To make minor changes such as moving some code a few columns, a utility program may be written or an existing one modified. Also, the lengths of individual blocks of source code are small to enhance readability and encourage a top-down approach. The objective is to have no block exceed a single listed page, or about fifty lines. Finally, by extending the range of structured programming progressions, efficiency of object code can be significantly improved, and source code readability is not impaired. Thus, iterative DOs with or without a WHILE clause and a simulated ALGOL-like CASE statement based on a subscripted GO TO statement and a LABEL array were permitted in our project.

Structured programming has been described in terms of languages with block structures such as PL/I, ALGOL, or JOVIAL. It is possible to introduce a simulated block structure into other

types of languages and then to develop structuring rules for them also. This has been done for System/360 Assembler Language, a low level language, through a set of macros that introduce and delimit blocks and provide DO WHILE, IF THEN ELSE and CASE-type figures. Further, if the long identifiers permitted by Assembler H are used, the source code is even more readable.

System development

This section discusses how the previously described techniques have been used in developing the information bank. The project was originally staffed with a chief programmer, a backup programmer, a system analyst (who was also a programmer), and a project manager. Since a project requirement was that the information bank operate under the System/360 Disk Operating System (DOS/360), the backup programmer began developing a version of the programming production library (PPL) that would operate under DOS/360. In parallel, the chief programmer and the system analyst began developing a detailed set of functional specifications. The first product of the team was a book of specifications that served as a detailed statement of the project objectives.

The team, at this point, reoriented itself from an analysis group into a development group, and a programmer technician was added to serve as a librarian. The system analyst began detailed design of system externals, such as the messages, communication log, and statistics reports. The chief programmer and backup programmer worked together on designing the various subsystems and their interfaces.

file maintenance subsystem

Since the system is heavily file oriented, efficient retrieval and the capability of adding large volumes of new material daily were requirements. Therefore, the chief and backup programmers initially emphasized the development of an interrelated set of six files that provide the necessary file attributes. Declarations of structures for these files were the first members placed in the library. Detailed file maintenance and retrieval algorithms were developed before any further design was done.

A substantial amount of data already existed on magnetic tape. Therefore, to begin building files for debugging and testing the system, it was desirable that the file maintenance subsystem be developed. This subsystem was designed to consist of two major programs and several minor ones. The chief programmer and backup programmer each began work on one of the major programs. Working in top-down fashion, control nuclei for each major program were developed. Functional code was gradually added to these nuclei to handle different types of file mainte-

nance transactions until the programs were complete. The minor programs were then produced similarly.

Because of the early need for the file maintenance programs, an independent acceptance test was held for this subsystem. One of the functions performed by the backup programmer was the development of a test plan that specified all functions of the subsystem requiring testing and an orderly sequence for performing the test using actual data and transactions. An indication of the quality achievable by the chief programmer team is afforded by the fact that no errors were detected during the subsystem test. In fact, no errors have been detected during fifteen months of operation subsequent to the test.

data entry subsystem

While the file maintenance subsystem was being developed, the chief programmer and system analyst designed an on-line system for keying and correcting indexing data destined for information bank files and for *The New York Times Index*. This indexing system became the data entry subsystem and additions to the conversational subsystem. The *Index* had previously been prepared by a programming system from data obtained by keying a complex free-form indexing language onto paper tape. The existing language was, therefore, extended to include the fields needed by the conversational subsystem and formalized by expressing it in Backus-Naur form. Because it was likely that the language would be modified as the project evolved, we decided to perform the editing of indexing data using syntax-direct techniques. (Another programmer was added to the team to develop the data entry subsystem around the syntax-directed editor.)

After the file maintenance subsystem had been delivered and the externals of the system specified, the system analyst programmed the authorization file subsystem, the message file subsystem, the log/statistics file processing subsystem, and the deferred print subsystem. (Another programmer was added, who wrote the statistics reporting subsystem.)

The chief programmer and backup programmer developed the conversational subsystem. Again, operating in top-down fashion, first programmed was the nucleus consisting of a time-sharing supervisor and the part of the terminal-handling package required to support the digital TV terminals. This nucleus was debugged with a simple function module that echoed back to a display material that was typed on the keyboard. After the nucleus was operational, development of the functions of the retrieval system itself commenced. System functions were programmed in retrieval order, so that new functions could be debugged and tested using existing operational functions, and an inquiry could proceed as far as programming existed to support it. All debugging was done in the framework of the conversa-

tional subsystem itself, and because of the time-sharing aspects of the system, several programmers could debug their programs simultaneously. The ability to modify tests as results were displayed at a terminal was helpful in checking out new code. Two programmers were added to the team to write functional code. A third programmer was added to extend the terminal-handling package for the 2260 and 2265 display terminals, and for the 2740 communication terminal. These programmers rapidly acquired sufficient knowledge of the interface with the time-sharing supervisor to write functional code despite their short participation on the team.

system testing

During this development process, the backup programmer prepared a test plan for the rest of the system to be used with realistic inquiries for the test. Although some errors were found during a five-week period of functional and performance testing, all were relatively small, and did not involve the basic logic of the system. Most errors were found in the functional code that had been most recently added to the system and had been the least exercised. The performance parts of the testing measured both sustained load handling and peak load handling. In spite of the fact that the performance tests were run on a System/360 Model 40 with three 2314 disk storage facilities as files, instead of on the System/360 Model 50 with seven disk storage facilities for which the performance objectives had been developed, performance objectives were successfully met.

Productivity

A key objective of the chief programmer team approach was to demonstrate increased productivity of the team over an equal number of conventionally organized programmers. This section discusses data on the productivity of the team and their strategy for using their time. Typical productivity measures are computed to facilitate comparison with other projects. Table 1 breaks down the staff months applied on the project, and Table 2 displays measures of amounts of source code produced.

Standardized definitions have been used in preparing these tables and achieving comparable measures of productivity. *Source lines* are eighty-character records in the library that have been incorporated into the information bank and consist of the following kinds of statements:

- Programming language
- Linkage-editor control
- Job control

Source coding has been broken into the following three levels of difficulty, which are summarized in Table 2:

Table 1 Analysis of project staffing by time and type of work

Work type	*Staff time (man months)*											
	Chief	*Backup*	*Analyst*	*Programmer* 1	2	3	4	5	*Technician*	*Manager*	*Sec'y*	*Total*
Requirements Analysis	2.5	1.0	8.0	0.5	–	–	–	–	–	–	–	12.0
System design	4.0	4.0	4.5	1.0	–	–	–	–	–	–	–	13.5
Unit design, programming, debugging, and testing	12.0	14.0	10.0	13.0	4.5	2.8	3.7	4.5	–	–	–	64.5
Documentation	2.0	2.0	4.5	1.5	0.2	0.2	0.3	0.3	–	–	–	11.0
Secretarial	–	–	–	–	–	–	–	–	–	–	7.0	7.0
Librarian	–	–	–	–	–	–	–	–	5.5	–	2.0	7.5
Manager	3.5	2.0	–	–	–	–	–	–	–	11.0	–	16.5
Total	24.0	23.0	27.0	16.0	4.7	3.0	4.0	4.8	5.5	11.0	9.0	132.0

Table 2 Lines of source coding by difficulty and level

Difficulty	*Level* *High*	*Low*	*Total*
Hard	5034	–	5034
Standard	44247	4513	48760
Easy	27897	1633	29530
Total	77178	6146	83324

- *Easy coding* has few interactions with other system elements. (Most of the support programs are in this category.)
- *Standard coding* has some interactions with other system elements. (Examples are the functional parts of the conversational subsystem and the data entry edit subsystem.)
- *Difficult coding* has many interactions with other system elements. (This category is limited to the control elements of the conversational subsystem.)

Source coding types have been categorized as one of the following:

- *High-level* coding in a language such as PL/I, COBOL, or JCL
- *Low-level* coding such as assembler language and linkage-editor control statements

Table 3 presents some simple measures of programmer productivity based on the same coding used for producing Tables 1 and 2. The first row includes work done on unit design, coding, debugging, and acceptance testing. The second row summarizes

Table 3 Programmer productivity

Organization	*Source lines per programmer day*
Unit design, programming debugging, and testing	65
All professional	47
With librarian support	43
Entire team	35

professional work, which includes system design and documentation, but not librarian support. The third row includes all programming and librarian support. The last row presents the productivity of the entire team on the completed system (excluding requirements analysis).

Team experience and conclusions

The chief programmer team approach appears to be desirable for the type of project discussed in this paper because programmer efficiency was substantially improved. The quality of the programming was demonstrated by nearly error-free acceptance testing with real data, by successful operation after delivery, and by its acceptance by system users.

The information bank system was specified, developed, and tested during a 132 man-month project. The team, in this experiment, was a relatively experienced one, and it performed at an above-average level. Comparing results of this experiment with results for comparable projects that were organized more conventionally, we believe that chiet programmer teams applying the methods described in this paper should probably be able to double normal productivity. In addition, the quality of the completed programs should be superior to conventionally produced programs in terms of lower levels of errors remaining, self-documentation, and ease of maintenance.

Another valuable experience of the chief programmer team approach was its manageability. The team had a lower than usual ratio of professional-to-support personnel. Because the number of people actually doing professional work was small, communications problems were significantly reduced. The chief programmer was more knowledgeable about the progress of the work than programming managers generally are because of his direct involvement in it and because the techniques used (particularly the Programming Production Library, top-down programming, and structured programming) made the status of the work highly

visible and understandable. This knowledge allowed both him and his management to react to problems sooner and more effectively than might have been the case had they been more detached from the work.

The relatively small size of the team made it highly responsive to change. The original functional specification went through six revisions, yet it was possible to adapt readily to major changes, even those occurring after programming was well along. Improved communication achieved through the consistent application of top-down programming, structured programming, and the PPL all contributed to team adaptability.

A functional organization was applied both within the team and to the project organization as a whole. Within the team, the functional distribution of work allowed team members to concentrate on those aspects of the job for which they were best equipped and most productive. At the project level, the functional organization allowed the chief programmer to concentrate on technical progress of the programming, both internally and in his relations with the system users. A very effective relationship was established between the chief programmer and the project manager, and no problems arose from the dual interface with the users—who fully understood the responsibilities of each of the managers. During a period when the chief programmer was off of the project, the backup programmer successfully ran the project.

The functional organization effectively broadened the range of career opportunities in the programming field by allowing senior programmers to continue to be productive in a technical capacity. Downward, the team approach offers programming related clerical opportunities to nonprogramming personnel. The team, as originally constituted, included a programmer technician for the clerical function, but two problems arose with this approach. The work did not require a programmer technician because the PPL procedures were well enough defined that no programming knowledge was required to operate it. Also, neither librarian support nor secretarial support became full-time jobs on the project. We, therefore, combined the two functions and trained a secretary to perform them. With two weeks of on-the-job training, the secretary was capable of acting as librarian by using the PPL. Combining the two jobs also worked well from a work load standpoint because when programming work was heavy then documentation was light, and vice versa.

The programming techniques and standards used by the team to enhance productivity and visibility also worked as planned. Top-down programming was similarly successful. System logic for one of the major programs ran correctly the first time and never

required a change as the program was expanded to its full size. This was helpful in debugging, since programs usually ran to completion, and the rare failures were readily traceable to newly added functions. Top-down programming also alleviated the interface problems normally associated with multiprogrammer projects, because interfaces were always defined and coded before any coding functions that made use of the interfaces.

The Programming Production Library run by the librarian-secretary achieved its objectives of removing many of the clerical aspects of programming from the programmer and of making the project more visible and, hence, more manageable. It also encouraged modularity of the programs and made top-down programming practical and effective.

Whereas the experiment was successful, there are still some unanswered questions and unsolved problems. Most obvious, perhaps, is whether the approach can be extended to larger projects. The best estimate at this time is that it probably can, but it needs to be tried. The general approach would be to begin a project with a single high-level team to do overall system design and nucleus development. After the nucleus is functioning, programmers on the original team could become chief programmers on teams developing major subsystems. The original team would assume control, review, validation, and testing duties and perform integration of the subsystems into the overall system. The process could be repeated at lower levels if necessary. It might appear that such a top-down evolution of the development process would increase the project time vis-à-vis the bottom-up approach. This is not necessarily true because of parallel development and integration, and it may take even less time. In any case, the risk should be substantially reduced because of the better visibility and management control in the team methodology.

A second major question concerns team composition and training. Because the team is a close-knit unit producing a large system at a faster-than-usual pace, close cooperation and good communication are essential. It is, therefore, desirable that team members be experienced professionals trained in the techniques described. Although a team may include one or possibly two less experienced programmers, larger teams would force the chief programmer to spend too high a percentage of his time in detailed training and supervision thereby reducing his own productivity. One solution may be to place newly trained programmers in program maintenance or in projects that are extending existing systems before placing them on teams that are developing new systems.

The selection of the chief programmer from among several candidates may be more difficult than was at first anticipated. The

chief programmer is responsible for team management and for technical representation of the project to a customer and to his own management. Therefore, management ability and experience are necessary qualifications. A chief programmer must also possess the creativity and drive to make significant technical contributions of his own and to assist other team members in making their contributions. This essential combination of skills rarely appears in the same individual. Thus the use of aptitude testing should probably be considered as part of the selection process. Potential chief programmers should of course first serve as backup programmers to obtain first-hand experience before taking on their own projects.

One final question that has frequently been asked is whether chief programmers are willing to accept the technical and managerial challenges of large projects with few people. Experienced chief programmers have responded to the challenges and have found that it leads to a degree of satisfaction that is hard to match.

To summarize, there is little in the chief programmer team organization and methodology that has not been previously tried. Laid bare, it is basically a functional organization of programming projects coupled with the use of tried and true tools to improve productivity and quality. It works well when it all fits snugly together and is applied in a consistent fashion over an entire project. Continuing evolution shows promise of making the programming production process more economical and more manageable.

CITED REFERENCES

1. *Chief Programmer Teams: Principles and Procedures*, Report No. FSC 71-5108, may be obtained from International Business Machines Corporation, Federal Systems Division, Gaithersburg, Maryland 20760.
2. C. Böhm and G. Jacopini, "Flow diagrams, Turing machines and languages with only two formation rules," *Communications of the ACM* **9**, No. 3, 366–371 (May 1966).
3. K. Conrow and R. G. Smith, "NEATER2: a PL/I source statement reformatter," *Communications of the ACM* **13**, No. 11 (November 1970).