Organizing for Structured Programming

F. T. Baker, IBM Federal Systems Division, Gaithersburg, Maryland, USA

ABSTRACT

A new type of programming methodology, built around structured programming ideas, has been gaining widespread acceptance for production programming. This paper discusses how this methodology has been introduced into a large production programming organization. Finally it analyzes the advantages and disadvantages of each component of the methodology and recommends ways it can be introduced in a conventional programming environment.

INTRODUCTION

At this point in time, the ideas of structured programming have gained widespread acceptance, not only in academic circles, but also in organizations doing production programming. An issue[1] of Datamation, one of the leading business data processing oriented magazines in the U.S., featured several articles on the topic. The meetings of SHARE and GUIDE, two prominent computer user groups, have had an increasing number of sessions on subjects related to structured programming. The IBM Systems Science Institutes are offering courses and holding seminars, and several books on the topic are in print.

What is perhaps not so widely appreciated, however, is that the organizations, procedures and tools associated with the implementation of structured programming are critical to its success. This is particularly true in production programming environments, where program systems (rather than single programs) are developed, people come and go, and the attainment of reliable, maintainable software on time and within cost estimates is a prime management objective. In this environment,

module level coding and debugging activities typically account for about 20% of the effort spent on software development[2]. Thus, narrow applications of structured programming ideas limited only to these activities have correspondingly limited effects. It is therefore desirable to adopt a broad, integrated approach incorporating the ideas into every aspect of the project from concept development to program maintenance to achieve as many quality improvements and cost savings as possible.

BACKGROUND

The IBM Federal Systems Division (FSD) is an organization involved in production programming on a large scale. Although much of its software work is performed for federal, state and local governmental agencies, the division also contracts with private business enterprises for complex systems development work. Work scope ranges from less than a man-year of effort on small projects to thousands of man-years spent on the development and maintenance of large, evolutionary, long-term systems such as the Apollo/Skylab ground support software. Varying customer requirements cause the use of a wide variety of hardware, programming languages, software tools, documentation procedures, management techniques, etc. Problems range from software maintenance through pure applications programming using commercially available operating systems and program products to the concurrent development of central processors, peripherals, firmware, support software and applications software for avionics requirements. Thus, within this single organization can be found a wide range of software development efforts.

FSD has always been concerned with the development of improved software tools, techniques and management methods. Most recently, FSD has been active in the development of structured programming techniques [3]. This has led to organizations, procedures and tools for applying them to production programming projects, particularly with a new organization called a Chief Programmer Team.[4] The Team, a functional organization

based on standard support tools and disciplined application of structured programming principles, had its first trial on a major software development effort in 1969-71.[5],[6] In the three years since the completion of that experimental project, FSD has been incorporating structured programming techniques into most of its software development projects. Because of the scope and diversity of these projects, it was impossible to adopt any single set of tools and procedures or any rigid type of organization to all or even to a majority of them. And because of the ongoing nature of many of these systems, it was necessary to introduce these techniques gradually over a period of many years. The approach which was adopted, the problems which have been encountered and the results which were achieved, are the subject of this paper. It is believed that any software development organization can improve the quality and reduce the costs of its software projects in a similar way.

PLAN

To introduce the ideas into FSD work practices and to evaluate their use, a plan with four major components was implemented. First, a set of guidelines was established to define the terminology associated with the ideas with sufficient precision to permit the introduction and measurement of individual components of the overall methodology. These guidelines were published, and directives regarding their implementation were issued. Second, support tools and methodologies were developed, particularly for projects using commercial hardware and operating systems. For those projects where these were not employed, standards based on the developed tools enabled them to provide their own support. Third, documentation of the techniques and tools, and education in their use, were both carried out. These were done on a broad scale covering management techniques, pro-

gramming methodologies and clerical procedures. Fourth, a measurement program was established to provide data for technology evaluation and improvement. This program included both broad measurements which were introduced immediately, and detailed measurements which required substantial development work and were introduced later. The next four sections cover the components of this plan and their implementation in detail.

GUIDELINES

A number of important considerations influenced the establishment of a set of guidelines for the application of structured programming technology within FSD. First and most important, they had to permit adaptation to the wide variety of project environments described above. This required that they be useful in program maintenance situations where unstructured program systems were already in being, as well as in those where completely new systems were to be developed. Second, they had to allow for the range of processors and operating systems in use. This necessitated the description of functions to be provided instead of specific tools to be used. Third, they had to allow for differences in organizations and methodology (e.g., specifications, documentation, configuration management) required or in use on various projects.

The guidelines resulting from these considerations are a hierarchical set of four components, graphically illustrated in Figure 1. Use of the component at any level presupposes use of those below it. Thus, by beginning at a level which a project's environment and status permit, and then progressing upward, projects can evolve gradually toward full use of the technology.

1. Development Support Libraries

The introductory level is the Development Support Library, which is a tool designed with two key principles in mind:

a. Keep current project status organized and visible at all times. In this way, any programmer, manager or user can find out the status or study an approach directly without depending on anyone else.

b. Make it possible for a trained secretary to do as much library maintenance as possible, thus separating clerical from intellectual activity.

A DSL is normally the primary responsibility of a Programming Librarian. Programmers interface with the computer primarily through the library and the Programming Librarian. This allows better control of computer activity and ensures that the library is always complete and current. Programmers are always working with up-to-date versions of programs and data, so that misunderstandings and inconsistencies are greatly reduced. A version and modification level are associated with all material in the library to permit change control and assist in configuration management. In general, the library system is the prime factor in increasing the visibility of a developing project and thus reducing risk and increasing reliability.

The guidelines provide that a Development Support Library is being used if the following conditions prevail:

a. A library system providing the functional equivalent of the PPL (see TOOLS below) is being used.

b. The library system is being used throughout the development process, not just to store debugged source or object code, for example.

c. Visibility of the current status of the entire project, as well as past history of source code activities and run executions, is provided by the external library.

d. Filing procedures are faithfully adhered to for all runs, whether or not setup was performed by a librarian.

e. The visibility of the code is such that the code itself serves as the prime reference for questions of data formats, program operation, etc.

f. Use of a trained librarian is recommended.

2. Structured Programming

In order to provide for use of structured programming techniques on maintenance as well as development projects, it was necessary to depart from the classical use of the terminology and adopt a narrower definition. In FSD, then, we distinguish between those practices used in system development (Top-Down Development) and those used in coding individual program modules (Structured Programming). Our use of the term "structured programming" in the guidelines thus refers primarily to coding standards governing control flow, and module organization and construction. They require three basic control flow figures and permit two optional ones, as shown in Figure 2. They refer to a Guide[7] (see DOCUMENTATION below) which contains general information and standards for structured programming, as well as

detailed standards for use of various programming languages. They also require that code be reviewed by someone other than the developer. The detailed guidelines for structured programming are as follows:

a. The conventions established in the Structured Programming Guide are being followed. Exceptions to conventions are documented. If a language is being used for which conventions have not been published in the Guide, then use of a locally generated set of conventions consistent with the rules of structured programming is acceptable.

b. The code is being reviewed for functional integrity and for adherence to the structured programming conventions.

c. A Development Support Library is being used.

3. Top-Down Development

Top-down development refers to the process of concurrent design and development of program systems containing more than a single compilable unit. It requires development to proceed in a way which minimizes interface problems normally encountered during the integration process typical of "bottom-up development" by integrating and testing modules as soon as they are developed. Other advantages are that:

a. It permits a project to man up more gradually and should reduce the total manpower required.

b. Computer time requirements tend to be spread more evenly over the development period.

c. The user gets to work with major portions of the system much earlier and can identify gross errors before acceptance testing.

d. Most of the system has been in use long enough by the time it is delivered that both the user and the developer have confidence in its reliability.

e. The really critical interfaces between control and function code are the first ones to be coded and tested and are in operation the longest.

The term "top-down" may be somewhat misleading if taken too literally. What top-down development really implies in everyday production programming is that one builds the system in a way which ideally eliminates (or more practically, minimizes) writing any code whose testing is dependent on other code not yet written, or on data which is not yet available. This requires careful planning of the development sequence for a large system consisting of many programs and data sets, since some programs will have to be partially completed before other programs can be begun. In practice, it also recognizes that exigencies of customer requirements or schedule may force deviations from what would otherwise be an ideal development sequence. The guidelines for top-down development are as follows:

a. Code currently being developed depends only on code already operational, except in those portions where deviations from this procedure are justified by special circumstances.

b. The project schedule reflects a continuing integration, as part of the development process, leading directly to system test, as opposed to a development, followed by integration, followed by system test, cycle.

c. Structured Programming is being used.

d. A Development Support Library system is being used. (While ongoing projects may not be able to meet this criterion, an implementation of structured coding practice is acceptable in these cases.)

e. The managers of the effort have attended a structured programming orientation course (see EDUCATION below).

4. Chief Programmer Teams

A Chief Programmer Team (CPT) is a functional programming organization built around a nucleus of three experienced professionals doing well-defined parts of the programming development process using the techniques and tools described above. It is an organization uniquely oriented around them and is a logical outgrowth of their introduction and use. Described in detail in [4], [5], and [6], it has been used extensively in FSD on projects ranging up to approximately 100,000 lines of source code and is being experimented with on larger projects. The guidelines for CPT's are as follows:

a. A single person, the chief programmer, has complete technical responsibility for the effort. He will ordinarily be the manager of the other people.

b. There is a backup programmer prepared to assume the role of chief programmer.

c. Top-Down Development, Structured Programming and a Development Support Library are all being used.

d. Top level code segments and the critical control paths of lower level segments are being coded by the chief and backup programmers.

e. The chief and backup programmers are reviewing the code produced by other members of the team.

f. Other programmers are added to the team only to code specific well defined functions within a framework established by the chief and backup programmers.

TOOLS

Tools are necessary in order to permit effective implementation of, and achieve maximum benefits from the ideas of structured programming. Development Support Libraries, introduced above, are a recognized and required component of the methodology employed in FSD. Standards are necessary to ensure a consistent approach and to help realize benefits of improved project communications and manageability. Procedures are required for effective use of the tools and to permit functional breakup and improved overall efficiency in the programming process. Finally, other techniques of design, programming, testing and management can be helpful in a structured programming environment as well as in a conventional one.

1. Development Support Libraries

The need for and value of Development Support Library (DSL) support, both as a necessity for structured programming and as a vehicle for project communication and control, has been thoroughly covered in [4], [5], [6], [7], and [8]. Early work on DSL's centered on the provision of libraries for projects using IBM's System/360 Operating System and Disk Operating System. The OS/360 Programming Production Library (PPL) is typical of those we are using in batch programming development situations. It consists of internal (computer-readable) and external (human-readable) libraries, and office and machine procedures. Similar concepts and approaches apply to our other library systems, including those working in an online environment.

The PPL keeps all machineable data on a project - source code, object code, linkage editor language, job control language, test data, and so on - in a series of data sets which comprise the internal library (see Figure 3). Since all data is kept internally and is fully backed up, there is no need for programmers to generate or maintain their own personal copies. Corresponding to each type of data in the internal library there is a set of current status binders which comprise the external library (see Figure 4). These are filed centrally and used by all as a standard means of communication. There is also a set of archives of superseded status pages which are retained to assist in disaster recovery, and a set of run books containing run results. Together, these record the activities - current and historical - of an entire project and keep it completely organized.

The machine procedures, as the name implies, are cataloged procedures which perform internal library maintenance, back-up, expansion and so on. Most of them are used by Programming Librarians by means of simple control cards they have been trained to prepare. A complete list is given in Table 1.

The office procedures are a set of "clerical algorithms" used by the Programming Librarian to invoke the machine procedures, to prepare the input and file the output. Once new code has been created and placed in the library initially, a programmer makes corrections to it by marking up pages in the external library and giving them to the Programming Librarian to make up control and data cards to cause the corresponding changes or additions to be made to the internal library. As a result, clerical effort and wasted time on the part of the programmers are significantly reduced. Figure 5 shows the work flow and the central role of the Programming Librarian in the process. Because programmers are served by the Librarian and the PPL, they are freed from interruptions and can work on more routines in parallel than they previously did. The PPL machine and office procedures are documented for programmers in [7] and for librarians in [9].

Subsequent work on DSL's in FSD has extended the support to some of the non-System/360 equipment in use and also introduced interactive DSL's for use both by librarians and programmers. Furthermore, a study of general requirements for DSL's has been performed under contract to the U. S. Air Force and has been published in [10]. DSL's are now available for and in use on most programming projects in FSD.

2. Standards

To support structured programming in the various languages used, programming standards were required. These covered both the implementation of the control flow figures in each language as well as the conventions for formatting and indenting programs in that language.

There are four approaches which can be taken to provide the basic and optional control flow figures in a programming language, and each was used in certain situations in FSD.

a. The figures may be directly available as statements in the language. In the case of PL/1, all of the basic figures were of this variety. In COBOL, the IFTHENELSE (with slight restrictions) and the DOUNTIL (as a PERFORM) were present.

b. The figures may be easily simulated using a few standard statements. The CASE statement may be readily simulated in PL/1 using an indexed GOTO and a LABEL array, with each case implemented via a DO-group ending in a GOTO to a common null statement following all cases.

c. A standard pre-processor may be used to augment the basic language statements to provide necessary features. The macro assembler has been used in FSD to add structuring features to System/360, System/370 and System/7 Assembler Languages.

d. A special pre-processor may be written to compile augmented language statements into standard ones, which may then be processed by the normal computer. This was done

for the FORTRAN language, which directly contains almost none of the needed features.

The result of using these four approaches was a complete set of figures for PL/1, COBOL, FORTRAN and Assembler. Using these as a base, similar work was also done for several special-purpose languages used in FSD.

To assist in making programs readable and in standardizing communications and librarian procedures, it was desirable that programs in a given language should be organized, formatted and indented in the same way. (This was true of the Job Control and Link Editor Languages as well as of the procedural languages mentioned above.) Coding conventions were developed for each covering the permitted control structures, segment formatting, naming, use of comments, labels, and indentation and formatting for all control flow and special (e.g., OPEN, CLOSE, DECLARE) statements.

3. Procedures

An essential aspect of the use of DSL's is the standardization of the procedures associated with them. The machine procedures used in setting up, maintaining and terminating the libraries were mentioned above in that connection. However, the office procedures used by librarians in preparing runs, executing them and filing the results are also quite extensive. These were developed and documented[9] in a form readily usable by non-programming oriented librarians.

4. Other

While the above constitute the bulk of the work originated by FSD, certain other techniques and procedures have been assimi-

lated into the methodology in varying degrees. These include management techniques, HIPO diagrams and structured walkthroughs.

FSD has been a leader in the development of management techniques for programming projects. A book[11] resulting from a management course and guide used in FSD has become a classic in the field. As top-down development and structured programming came into use, it became apparent that traditional management practices would have to be substantially revised (see IMPLEMENTATION EXPERIENCE below). An initial examination was done, and a report[12] was issued which has been very valuable in guiding managers into using the new methodology. This material is now being added to a revised edition of the FSC Programming Project Management Guide, from which the book[11] mentioned above was drawn.

A documentation technique called HIPO (Hierarchy plus Input-Process-Output) diagrams [8,13] developed elsewhere in IBM has proved valuable in supporting top-down development. HIPO consists of a set of operational diagrams which graphically describe the functions of a program system from the general to the detail level. Not to be confused with flowcharts, which describe procedural flow, HIPO diagrams provide a convenient means of documenting the functions identified in the design phase of a top-down development effort. They also serve as a useful introduction to the code contained in a DSL and as a valuable maintenance tool following delivery of the program system.

Structured walk-throughs[8] were developed on the second CPT project as a formal means for design and code reviews during the development process. Using HIPO diagrams and eventually the

code itself, the developer "walks through" his efforts for the reviewers. These latter may consist of the Chief or Backup Programmer (or lead programmer if a CPT is not being employed), other programmers and a representative from the group which will formally test the programs. Emphasis is on error avoidance and detection, not correction, and the attitude is open and non-defensive on the part of all participants (today's reviewer will be tomorrow's reviewee). The reviewers prepare for the walk-through by studying the diagrams or code before the meeting, and followup is the responsibility of the reviewee, who must notify the reviewers of corrective actions taken.

DOCUMENTATION AND EDUCATION

Once the fundamental tools and guidelines were established, it was necessary to begin disseminating them throughout FSD. Much experimental work had already been done in developing the tools and guidelines themselves, so that a cadre of people familiar with them was already in being.

Most of the documentation has been referred to above. The primary reference for programmers was the FSC Structured Programming Guide[7]. In addition to the standards for each language and for use of the PPL, it contained general information on the use of top-down development and structured programming, as well as the procedures for making exceptions to them when necessary. It also contained provisions for sections to be added locally when special-purpose languages or libraries were in use. Distributed throughout FSD, the Guide has been updated and is still the standard reference for programmers. The FSC Programming Librarian's Guide[9] serves a similar purpose for librarians and also has provisions for local sections where necessary. While

the use of the macros for System/360 Assembler Language was included in the Programming Guide, additional documentation[14] was available on them if desired. Finally, management documentation in the form of [11] and [12] was also available.

It was recognized that providing documentation alone was not sufficient to permit most personnel to begin applying the techniques. Structured programming requires substantial changes in the patterns and procedures of programming, and a significant mental effort and amount of practice is needed to overcome old habits and instill new ones. A series of courses (one for each major language) was set up to train experienced FSD programmers in structured programming and DSL techniques. Lasting twenty-five hours, these courses provided instruction and, more importantly, practice problems which forced the programmers to begin the transition process. Once all programmers had been retrained, these courses were discontinued, and structured programming is now included as part of the basic programmer training courses given to newly hired personnel.

The same situation held true for managers as well as programmers. Because FSD wished to apply the methodology as rapidly as possible, it was desirable to acquaint managers with it and its potential immediately. Thus, one of the first actions taken was to give a half-day orientation course to all FSD managers. This permitted them to evaluate the depth to which they could begin to use it on current projects, and to begin to plan for its use on proposed projects. This was then followed up by a twelve-hour course for experienced programming managers, acquainting them with management and control techniques peculiar to top-down development and structured programming. (It was ex-

pected that most of these managers would also attend one of the structured programming courses described above to acquire the fundamentals.) Again, now that most programming managers have received this form of update, the material has now been included in the normal programming management course given to all new programming managers.

MEASUREMENT

One of the problems of the production programming world is that it has not developed good measures of its activities. Various past efforts, most notably the System Development Corporation studies[15] have attempted to develop measurement and prediction techniques for production programming projects. The general results have been that a number of variables must be accurately estimated to yield even rough cost and schedule predictions, and that the biggest factors are the experience and abilities of the programmers involved. Nevertheless, it was felt in FSD that some measures of activity were needed, not so much for prediction as for evaluation of the degree to which the methodology was being applied and the problems which were experienced in its use. To these ends, two types of measurements were put into effect.

The first type of measurement, implemented immediately, was a monthly report required from each programming project. Each programming manager was required to state:

1. The total number of programmers on the project.

2. The number currently programming.

3. The number using structured programming.

4. The number of programming groups on the project.

5. The number of CPT's.

6. Whether a DSL was in use.

7. Whether top-down development was in use.

These figures were summarized monthly for various levels of FSD management and were a valuable tool in ensuring that the methodology was indeed being introduced.

The second type of measurement was a much more comprehensive one. It required a great deal of research in its preparation, and eventually took the form of a questionnaire from which data was extracted to build a measurement data base. The questionnaire contains 105 questions organized into the following eight sections:

1. Identification of the project.

2. Description of the contractual environment.

3. Description of the personnel environment.

4. Description of the personnel themselves.

5. Description of the technical environment.

6. Definition of the size, type and quality of the programs produced.

7. Itemization of the financial, computer and manpower resources used in their development.

8. Definition of the schedule.

The questionnaire is administered at four points during the lifetime of every project. The first point is at the beginning, in which all questions are answered with estimates. The next administration is at the end of the design phase, when the initial estimates are updated as necessary. It is again filled out halfway through development, when actual figures begin to be known. And it is completed for the last time after the system has been tested and delivered, and all results are in. The four points provide for meaningful comparisons of estimates to actuals, and allow subsequent projects to draw useful guidance for their own planning. The data base permits reports to be prepared automatically and statistical comparisons to be made.

IMPLEMENTATION EXPERIENCE

Each of the four components of the methodology which FSD has introduced has resulted in substantial benefits. However, experience has also revealed that their application is neither trivial nor trouble-free. This section presents a qualitative analysis of the experience to date, describing both the advantages and the problems.

1. Development Support Libraries

Most projects of any size have historically gravitated toward use of a program library system of some type. This was cer-

tainly true in FSD, which had some highly developed systems already in place when the methodology was introduced. These were primarily used as mechanisms to control the code, so that differing versions of complex systems could be segregated. In some cases they provided program development services such as compilation, testing and so forth. However, none were being used primarily to achieve the goals of improved communications or work functionalization which are the primary benefits of a DSL. In fact, the general attitude toward the services they provided was that they were there to be used when and if the programmers wished. Most code in them was presumed private, with the usual exceptions of macro and subroutine libraries.

One of the most difficult problems in the introduction of the DSL approach was to convince ongoing projects that their present library systems fulfilled neither the requirements nor the intents of a DSL. A DSL is as much a management tool as a programmer convenience. A Programming Librarian's primary responsibility is to management, in the sense of supporting control of the project's assets of code and data -- analogous to a controller's responsibility to management of supporting control of financial assets. The project as a whole should be entirely dependent on the DSL for its operation, and this, more than any other criterion, is the determining factor in whether a library system meets the guidelines as a DSL.

When all functions are provided, and a project implements a DSL, then a high degree of visibility is available. Programmers use the source code as a basic means of communication and rely on it to answer questions on interfaces or suggest approaches to their problems. Managers use the code itself (or the summary

features of more sophisticated DSL's) to determine the progress of the work. Users also benefit, even at an early stage of implementation, from the ready availability of the test data and the feasibility of beginning to use the developing system on an experimental basis.

The visibility in itself is valuable, even on a laissez-faire basis. But when it is coupled with well-managed code-reading procedures, it also provides quality improvements. The walk-throughs described above, or equivalent procedures, ensure that someone in addition to the developer reviews the code, verifying that the specifications have been addressed, checking the planned test coverage, assisting in standards compliance and last but not least, constructively criticizing the content. While the review procedure is obviously greatly facilitated by concomitant use of structured programming, it is possible without it and was included with the DSL guidelines to encourage its adoption.

The archives which are an integral part of a DSL provide an ability to refer to earlier versions of a routine - sometimes useful in tracing intent when a program is passed from hand to hand. More importantly, they give a project the ability to recover from a disaster in which part of its resources are destroyed. (It is perhaps obvious but worth mentioning that this will not be complete insurance unless project management sees to it that the backup data sets are stored physically separate from the working versions.) There was an initial tendency in FSD to over-collect and to over-formalize the archiving process. It appears unnecessary to retain more than a few generations of object code, run results and so forth. The source code and test

data generally warrant longer retention, but even here it rapidly becomes impractical to save all versions. In general, sufficient archives should be retained to provide complete recovery capability when used in conjunction with the backup data sets, plus enough additional to provide back references.

The separation of function introduced by the DSL office procedures has two main benefits. The obvious one is of lowered cost through the use of clerical personnel instead of programmers for program maintenance, run setup and filing activities. A significant additional benefit comes about through the resulting more concentrated use of programmers. By reducing interruptions, librarians afford the programmers a work environment in which errors are less likely to occur. Furthermore, they permit programmers to work on more routines in parallel than typically is the case.

The last major benefit derived from a DSL rests in its support of a programming measurement activity. By automatically collecting statistics of the types described above, they can enhance our ability to manage and improve the programming process. The early DSL's in FSD did not include measurement features, and the next generation is only beginning to come into use, so a full assessment of this support is not yet possible.

It was difficult to convince FSD projects in some cases that a well-qualified Programming Librarian could benefit a project as much as another programmer. In fact, there was an initial tendency to use junior programmers or programmer technicians to provide librarian support. This had two disadvantages and hence is not recommended. First, the use of programming-qualified

personnel is not necessary because of the well-defined procedures inherent in the DSL's. Use of overqualified individuals in some cases led to boredom and sloppy work with a resulting loss of quality. Second, such personnel cannot perform other necessary functions when needed. One of the advantages of using secretaries as librarians is that they can use both skills effectively over the lifetime of a typical project. During design and documentation phases, they can provide typing and transcription services; while during coding and testing phases, they can perform the needed librarian work.

Two problems remain in defining completely the role of librarians. First, the increasing use of interactive systems for program development is forcing an evolution of librarian skills toward terminal operation and test support rather than coding of changes and extensive filing. The most effective division of labor between programmer and librarian in such an environment remains to be determined. It also appears possible to use librarians to assist in documentation, such as in preparation of HIPO diagrams. Second, FSD has a number of small projects in locations remote from the major office complexes and support facilities - frequently on customer premises. Here it is not always possible to use a librarian cost-effectively. In this situation, better definition of the programmer/librarian relationship in the interactive system development environment may permit development and librarian support to some extent from the central facility instead of requiring all personnel to be on-site.

2. Structured Programming

Recall that the FSD use of the term "structured programming" is a narrow one, adopted to permit ongoing projects to use some of the methodology. In this usage, it is more like what might be called "structured coding" and is limited to those techniques used in developing a single compilable unit (module). Combined with usage of a DSL, it provides enhanced readability of code, enforces modularity and thus encourages changeability and maintainability, simplifies testing, and permits improved manageability and accountability. These are all well-known properties of structured programming and need not be elaborated on here. An additional, unplanned for, benefit of structured programming is that it tends to encourage the property of "locality of reference", which improves performance in a virtual systems environment.

Reflecting on the advantages attributed to structured programming and the use of DSL's, one is struck by the fact that the techniques fundamentally are directed toward encouraging programming discipline. Historically, programming has been a very individualistic, undisciplined activity. Thus, introducing discipline in the form of practices which most programmers recognize as beneficial, yields double rewards -- the advantages inherent in the methodology itself, plus those due to better standardization and control.

The introduction of structured programming was not easily achieved in FSD. The broad variety of projects, languages and support has already been mentioned, and the development of DSL's, the Guides[7,9] and the education program were necessary before widespread application of the methodology could take

place. Furthermore, the ongoing nature of many of the systems meant that structured programming could take place only as modules were rewritten or replaced.

This gradual introduction created a problem of educational timing. Practically, it was most expedient to have programmers attend the education courses between assignments. The nature of the courses was such that they introduced the techniques and provided some initial practice. Yet they required substantial work experience using the techniques to be fully effective. Structured programming requires the development of a whole new set of personal patterns in programming. Until old habits are unlearned and replaced by new ones, it is difficult for programmers to fully appreciate the advantages of structured programming. For best results, this work experience and the overcoming of the natural reluctance to change habits should follow the training immediately. This was not always feasible and resulted in some loss of educational effectiveness.

A second problem arose because of the real-time nature of a significant fraction of FSD's programming business. Here the difficulty was one of demonstrating that structured programming was not detrimental to either execution speed or core utilization. While it is difficult to verify the advantages quantitatively, a working consensus has arisen. Simply stated, it is that the added time and thought required to structure a program pay off in better core utilization and improved efficiency which generally are comparable to the effects achieved in unstructured programs by closer attention to detail. It is also useful to note that even in "critical" programs, a relatively small fraction of the code is really time- or core-sensitive, and this

fraction may not in fact be predictable a priori. Hence it is probably a better strategy to use structured programming throughout to begin with. Then, if performance bottlenecks do appear and cannot be resolved otherwise, at most small units of code must be hand-tailored to remedy the problems. In this way the visibility, manageability and maintainability advantages of structured programming are largely retained.

Perhaps the most difficult problem to overcome in applying structured programming is the purist syndrome, in which the goal is to write perfectly structured code in every situation. It must be emphasized that structured programming is not an end in itself, but is a means to achieving better, more reliable, more maintainable programs. In some cases (e.g., exiting from a loop when a search is complete, handling interrupt conditions), religious application of the figures allowed by the Guide may produce code which is less readable than that which might contain a GO TO (e.g., to the end of the loop block, or to return from the interrupt handler to a point other than the point of interrupt). Clearly the exceptions must be limited if discipline is to be maintained, but they must be permitted when desirable. Our approach in FSD has been to require management approval and documentation for each such deviation. This ensures that only cases which are clearly justified will be nominated as exceptions, since otherwise the requirements are prohibitive.

3. Top-Down Development

As defined above, top-down development is the sequencing of program system development to eliminate or avoid interface problems.

This permits development and integration to be carried out in parallel and provides additional advantages such as early availability discussed under GUIDELINES.

Top-down development is the most difficult of the four components to introduce, probably because it requires the heaviest involvement and changes of approach on the part of programming managers. Top-down development has profound effects on traditional programming management methodology. While the guidelines sound simple, they require a great deal of careful planning and supervision to carry out thoroughly in practice, even on a small project. The implementation of top-down development, unlike structured programming and DSL's, thus is fundamentally a management and not a programming problem.

Let us distinguish at this point between what might be called "top-down programming" and true top-down development. While they were originally used interchangeably and the guidelines do not distinguish between them, the two terms are valuable in delineating levels of scope and complexity as use of the methodology increases.

Top-down programming is primarily a single-program-oriented concept. It applies to the development of a "program", typically consisting of one or a few load modules and a number of independently compilable units, which is developed by one or a few programmers. At this level of complexity the problems are primarily ones of program design, and the approaches used are those of classical structured programming (here not the narrower FSD definition) such as "levels of abstraction"[16] and the use of Mills' Expansion Theorem[3]. Within this scope of development

external problems and constraints are not as critical, and while management involvement is needed, it need not be so pervasive as in top-down development. Many of FSD's successful projects have been of this nature, and the experience gained on them has been most valuable.

Top-down development, on the other hand, is a multiple-program oriented idea. It applies to the development of a "program system", typically consisting of many load modules and perhaps a hundred or more independently compilable units, which is developed by one or more programming departments with five or more people in each. Now the problems expand to those of system architecture, and external problems and constraints become the major ones. The programs in the system are usually interdependent and have a large number of interfaces, perhaps directly but also frequently through shared data sets or communications lines. They may operate in more than one processor concurrently - for example, in a System/7 "front end" and a System/370 "host".

The complexity of such a system makes management involvement in its planning and development essential even when external constraints are minimal. It involves all aspects of the project from its inception to its termination. For example, a proposal for a project to be implemented top-down should differ from one for a conventional implementation in the proposed manning levels and usage of computer time. Functions must be carefully analyzed during the system design phase to ensure that the requirements of minimum code and data dependency are met, and a detailed implementation sequence must be planned in accordance with the overall proposed plan and schedule. The design of the system

very probably should differ significantly from what it would have been if a bottom-up approach were to be used. During implementation, progress must be monitored via the DSL to ensure that this sequence is being followed, and that schedules are being met. The early availability of parts of the system must be coordinated with the user if he intends to use these parts for experimentation or production. An entirely different type of test plan must be prepared, for incremental testing over the entire period. Rather than tracking individual components, the manager is more concerned with the progress of the system as a whole, which is a more complicated matter to assess. This is normally not determinable until the integration phase in a bottom-up development, when it suddenly become a critical item; in top-down work it is a continuing requirement, but one which enables the manager to identify problems earlier and to correct them while there is still time to do so.

In a typical system development environment such as those in FSD, however, external constraints are the rule rather than the exception. A user will have schedule requirements which must be met. A particular data set must be designed to interface with an existing system. Special hardware may arrive late in a development cycle and may vary from that desired. These are typical of situations not directly under the developers' control which have profound effects on the sequence in which the system is produced. Now the manager's job becomes still more complex in planning and controlling development. Each of these external constraints may force a deviation from what would otherwise be a classical, no-dependency development sequence. Provision may have to be made for testing, documentation and delivery of

products at intermediate points in the overall cycle. This will typically change the schedule from the ideal one, and will probably increase the complexity of the management job. This is especially true on a very large project (several hundred thousand lines of source code or more), since any realistic schedule may well require that major subsystems be developed in parallel and integrated in a nearly conventional fashion (hopefully at an earlier point in time than the end of the project). This was carried out successfully on a project of 400,000 lines of source code, the largest known to the author to date.

When carried to its fullest extent, top-down development of a large system probably has greater effects on quality (and thus, indirectly, on productivity) than any other component of the methodology. Even when competent management is fully devoted to its implementation, there are two other problems which potentially can arise and must be planned for. These both relate to the overlapping nature of design, development and integration in a top-down environment.

The first of these concerns the nature of materials documenting the system to be delivered to and reviewed by the user. Typically, a user receives a program design document at the end of the design phase and must express his concurrence before development proceeds. This is impractical in top-down development because development must proceed in some areas before design is complete in others. To give a user a comparable opportunity, a detailed functional specification is desirable instead. This describes all external aspects of a system, as well as any processing algorithms of concern to a user, but does not address its internal

design. This type of specification is probably more readily assimilated by typical users, is more meaningful than a design document and should pose no problems in most situations. Where standardized procurement regulations (such as U.S. Government Armed Services Procurement Regulations) are in effect, then efforts must be made to seek exceptions. (As top-down development becomes more prevalent, then it is hoped that changes to such procedures will directly permit submission of this type of specification.)

The second problem is one of the most severe to be encountered in any of the components and is one of the most difficult to deal with. It has to do with the depth to which a design should be carried before implementation is begun. If a complete, detailed design of an entire system is done, and implementation of key code in all areas is carried out by the programmers who begin the project, then the work remaining for programmers added later is relatively trivial. In some environments this may be perfectly appropriate and perhaps even desirable; in others it may lead to dissatisfaction and poor morale on the part of the latecomers. It can be avoided by recognizing that design to the same depth in all areas of most systems is totally unnecessary. The initial system design work (the overworked term "architecture" still seems to be appropriate here) should concentrate on specifying all modules to be developed and all inter-module interfaces. Those modules which pose significant schedule, development or performance problems should be identified, and detailed design work and key code writing done only on these. This leaves scope for creativity and originality on the part of the newer programmers, subject obviously to review and concurrence through normal project design control procedures. On some projects, the design of entire support sub-

systems with interfaces to a main subsystem only through standard, straightforward data sets has been left to late in the project. Note that while this may solve the problems of challenge and morale, it also poses a risk that the difficulty has been underestimated. Thus, here again management is confronted with a difficult decision where an incorrect assessment may be nearly impossible to recover from.

4. Chief Programmer Teams

The introduction of CPT's should be a natural outgrowth of top-down development. The use of a smaller group based on a nucleus of experienced people tends to reduce the communications and control problems encountered on a typical project. Use of the other three components of the methodology enhances these advantages through standardization and visibility.

In order for a CPT to function effectively, the Chief Programmer must be given the time, responsibility and authority to perform the technical direction of the project. In some environments this poses no problems; in FSD it is sometimes difficult to achieve because of other demands which may be levied upon the chief. In a contract programming environment he may be called upon to perform three distinct types of activities: technical management - the supervision of the development process itself, personnel management - the supervision of the people reporting to him, and contract management - the supervision of the relationships with the customer. This latter in particular can be a very time-consuming function and also is the simplest to secure assistance on. Hence many FSD CPT's have a program manager who has the primary customer interface responsibility in all non-

technical matters. The Chief remains responsible for technical customer interface as well as the other two types of management; in most cases this makes the situation manageable, and if not then additional support can be provided where needed.

The Backup Programmer role is one that seems to cause people a great deal of difficulty in accepting, probably because there are overtones of "second-best" in the name. Perhaps the name could be improved, but the functions the Backup performs are essential and cannot be dispensed with. One of the primary rules of management is that every manager should identify and train his successor. This is no less true on a CPT and is a major reason for the existence of the Backup position. It is also highly desirable for the Chief to have a peer with whom he can freely and openly interact, especially in the critical stages of system design. The Backup is thus an essential check and balance on the Chief. Because of this, it is important that the Chief have the right of refusal on a proposed Backup; if he feels that an open relationship of mutual trust and respect cannot be achieved, then it is useless to proceed. The requirement that the Backup be a peer of the Chief also should not be waived, since it is always possible that a Backup will be called on to take over the project and must be fully qualified to do so.

One of the limits on a CPT is the scope of a project it can reasonably undertake. It is difficult for a single CPT to get much larger than eight people and still permit the Chief and Backup to exercise the essential amount of control and supervision. Thus, even at the higher-than-normal productivity rates achievable by CPT's it is difficult for a single Team to produce

much more than perhaps 20,000 lines of code in its first year and 30-40,000 lines thereafter. Larger projects must therefore look to multiple CPT's, which can be implemented in two ways. First, as mentioned above under Top-Down Development, interfaces may be established and independent subsystems may be developed concurrently by several CPT's and then integrated. Second, a single CPT may be established to do architecture and nucleus development for the entire system. It then can spin off subordinate CPT's to complete the development of these subsystems. The latter approach is inherently more appealing, since it carries the precepts of top-down development through intact. It is also more difficult to implement; the experiment under way by the author ran into problems because equipment being developed concurrently ran into definition problems and prevented true top-down development.

It is difficult to identify problems unique to CPT's which differ from those of top-down development discussed above. Perhaps the most significant one is the claim frequently heard that, "We've had Chief Programmer Teams in place for years - there's nothing new there for us." While it is certainly true that many of the elements of CPT's are not new, the identification of the CPT as a particular form of functional organization using a disciplined, precise methodology suffices to make it unique. In particular, the emphasis on visibility and control through management code-reading, formal structured programming techniques and DSL's differentiate true CPT's from other forms of programming teams[17]. And it is this same set of features which make the CPT approach so valuable in a production programming environment where close control is essential if cost and schedule targets are to be met.

MEASUREMENT RESULTS

It is not possible, because it would reveal valuable business information, to present significant amounts of quantitative information in this paper. At this time, the results of the measurement program do show substantial improvements in programming productivity where the new technology has been used. A graph has been prepared where each point represents an FSD project. The horizontal axis records the percentage of structured code in the delivered product, and the vertical axis records the productivity. (The latter includes all effort on the project, including analysis, design, testing, management, support and documentation as well as coding and debugging. It also is based only on delivered code, so that effort used to produce drivers, code written but replaced, etc., tends to reduce the measured productivity.) A weighted least squares fit to the points on the graph shows a better than 1.5 to 1 improvement in the coding rate from projects which use no structured programming to those employing it fully.

It is also possible, because the data has already been released elsewhere, to make one quantitative comparison between productivity rates experienced using various components of the technology on some of the programming support work which FSD has performed for the National Aeronautics and Space Administration's Apollo and Skylab projects. This comparison is especially significant because the only major change in approach was the degree to which the new methodology was used; the people, experience level, management and support were all substantially the same in each area.

Figure 6 shows the productivity rates and the components of the technology used. In the Apollo project, a rate of 1161

bytes of new code per man-month was experienced on the Ground Support Simulation work. (Again, all numbers are based on overall project effort.) This work used none of the components described in this paper. In the directly comparable effort on the Skylab project, a DSL, structured programming and top-down development were all employed, and a rate of 3756 bytes of new code per man-month was achieved -- almost twice as much new code was produced with slightly more than half the effort. It is interesting also to remark that this was achieved on the planned schedule in spite of over 1100 formal changes made during the development of that product, along with cuts in both manpower and computer time. Finally, while the improvement may rest to some extent on the similar work done previously, this was not demonstrated in the parallel Mission Operations Control work. There productivity dropped from 1547 to 841 bytes per man-month on comparable work which in neither case used anything other than a DSL.

In addition to making quality measurements and determining productivity rates, the measurement activity has served a number of other useful purposes. First, it has built up a substantial data base of information about FSD projects. As new data is added, checks are made to ensure its validity, and questionable data is reviewed before being added. The result is an increasingly consistent and useful set of data. Second, it has enabled FSD to begin studies on the value of the components of the methodology. Third, and related, it also permits the study of other factors (e.g., environment, personnel) affecting project activity. Fourth, it is used to assist in reviewing ongoing projects, where the objective data it contains has proved quite valuable. And fifth, it is used in estimating for proposed projects, where it affords an opportunity to compare the new work against similar work done in the past, and to identify risks which may exist.

CONCLUSIONS

It should be clear at this point that FSD's experience has been a very positive one. Work remains to be done, particularly in the management of top-down development and the formalization and application of CPT's. Nevertheless, FSD is fully committed to application of the methodology and is continuing to require its use.

In retrospect, the plan appears to have been a success and could serve as a model for other organizations interested in applying the ideas. The FSD experience shows that this is neither easy nor rapid. It takes substantial time and effort and, most important, commitments and support from management, to equip an organization to apply the methodology.

To summarize, it appears that once a base of tools, standards and education exists, it is most appropriate to begin with use of structured and top-down programming and DSL's. Where the people, knowhow and opportunity exist, then top-down development should be applied on a few large, complex projects to yield an experienced group of people and the required management techniques. It is likely that one or more of these may also present the opportunity to introduce a CPT. This is essentially the approach that FSD has taken, and it appears to be an excellent way to organize for structured programming.

REFERENCES

[1] Datamation, Vol. 19, No. 12, December, 1973, pp. 50-63

[2] B. W. Boehm, "Software and its Impact: A Quantitative Assessment", Datamation, Vol. 19, No. 5, May, 1973, p. 52

[3] H. D. Mills, Mathematical Foundations for Structured Programming, Report No. FSC 72-6012, IBM Corporation, Gaithersburg, Maryland, USA, February, 1972

[4] H. D. Mills, Chief Programmer Teams: Principles and Procedures, Report No. FSC 71-5108, IBM Corporation, Gaithersburg, Maryland, USA, June, 1971

[5] F. T. Baker, "Chief Programmer Team Management of Production Programming", IBM Systems Journal, Vol. 11., No. 1, 1972, pp. 56-73

[6] F. T. Baker, "System Quality Through Structured Programming", AFIPS Conference Proceedings, Vol. 41, Part I, 1972, pp. 339-343

[7] Federal Systems Center Structured Programming Guide, Report No. FSC 72-5075, IBM Corporation, Gaithersburg, Maryland, USA, July, 1973 (revised)

[8] Improved Technology for Application Development: Management Overview, IBM Corporation, Bethesda, Maryland, USA, August, 1973

[9] Federal Systems Center Programming Librarian's Guide, Report No. FSC 72-5074, IBM Corporation, Gaithersburg, Maryland, USA, April, 1972

[10] F. M. Luppino and R. L. Smith, Programming Support Library (PSL) Functional Requirements: Final Report, IBM Corporation, Gaithersburg, Maryland, USA, prepared under Contract #F30602-74-C-0186 with the U. S. Air Force HQ Rome Air Development Center, Griffiss Air Force Base, New York, USA, July, 1974 (Release subject to approval of Contracting Officer, Mr. Paul DeLorenzo)

[11] P. W. Metzger, Managing a Programming Project, Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1973

[12] R. C. McHenry, Management Concepts for Top Down Structured Programming, IBM Corporation, Gaithersburg, Maryland, USA, November, 1972

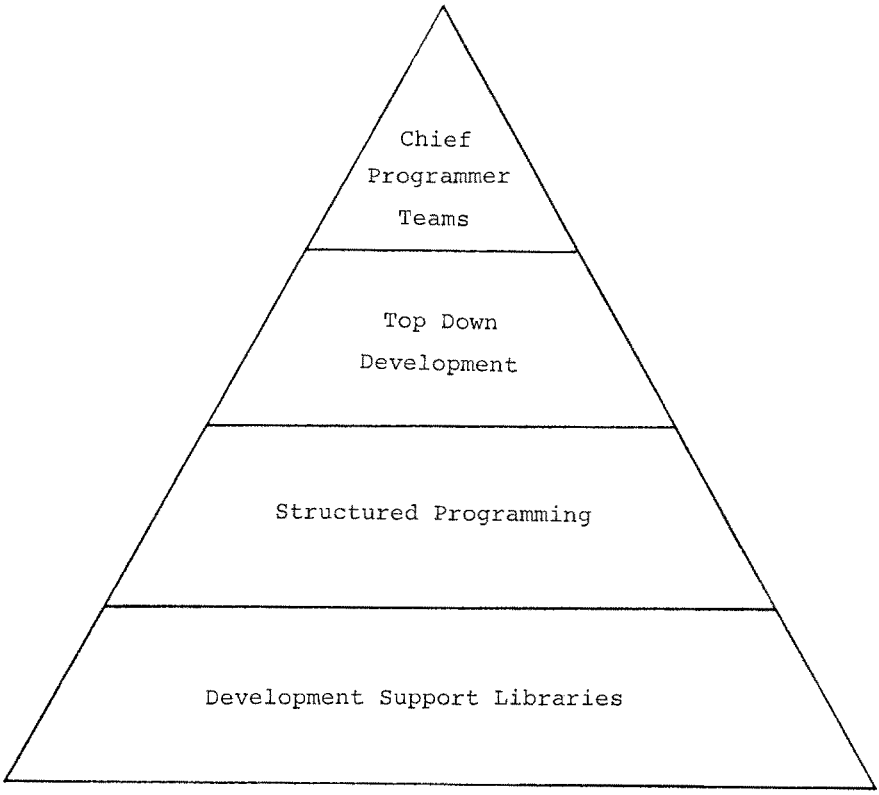
[13] HIPO - Hierarchical Input - Process - Output Documentation Technique: Audio Education Package, IBM Corporation, Form No. SR20-9413 (Available through any IBM Branch Office)

[14] M. M. Kessler, Assembly Language Structured Programming Macros, IBM Corporation, Gaithersburg, Maryland, USA, September, 1972

[15] G. F. Weinwurm et al, Research into the Management of Computer Programming: A Transitional Analysis of Cost Estimation Techniques, System Development Corporation, Santa Monica, California, USA, November, 1965 (available from the Clearinghouse for Federal Scientific and Technical Information as AD 631 259)

[16] E. W. Dijkstra, "The Structure of the THE Multiprogramming System", Communications of the ACM, Vol. 11., No. 5, May, 1968, pp. 341-346

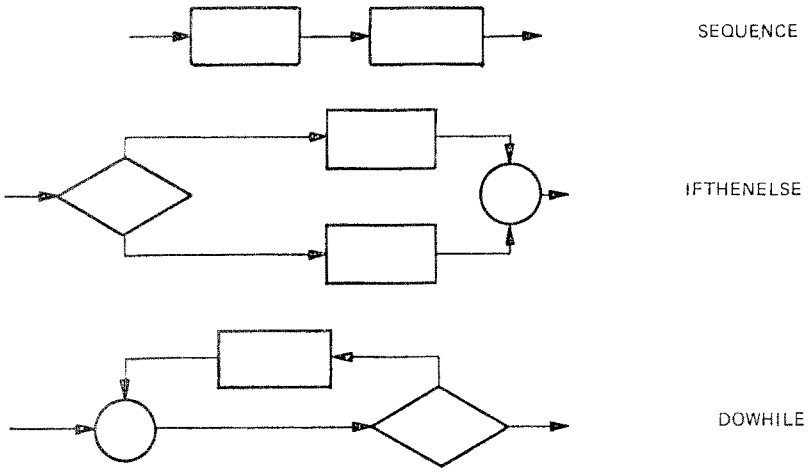
[17] G. M. Weinberg, The Psychology of Computer Programming, Van Nostrand Reinhold, New York, New York, USA, 1971



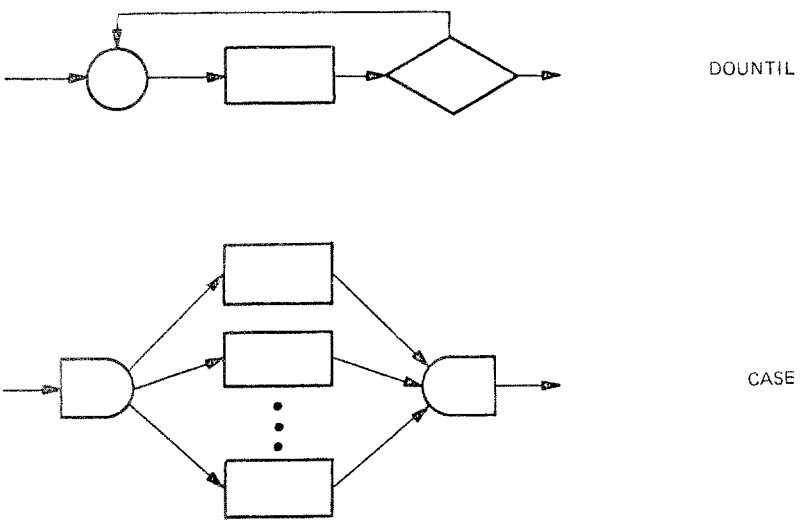
Hierarchy of Techniques

Figure 1

BASIC FIGURES



ADDITIONAL FIGURES



Control Structures

Figure 2

JCL	Job control language
LEL	Linkage editor language
SOURCE	Source (PL/1, Fortran, BAL, COBOL) language
TEST	Project test data
OBJECT	Compiler output
LOAD	Linkage editor output
SYSIN	PPL control data

PPL Internal Library Data Sets

Figure 3

A set of current status notebooks:

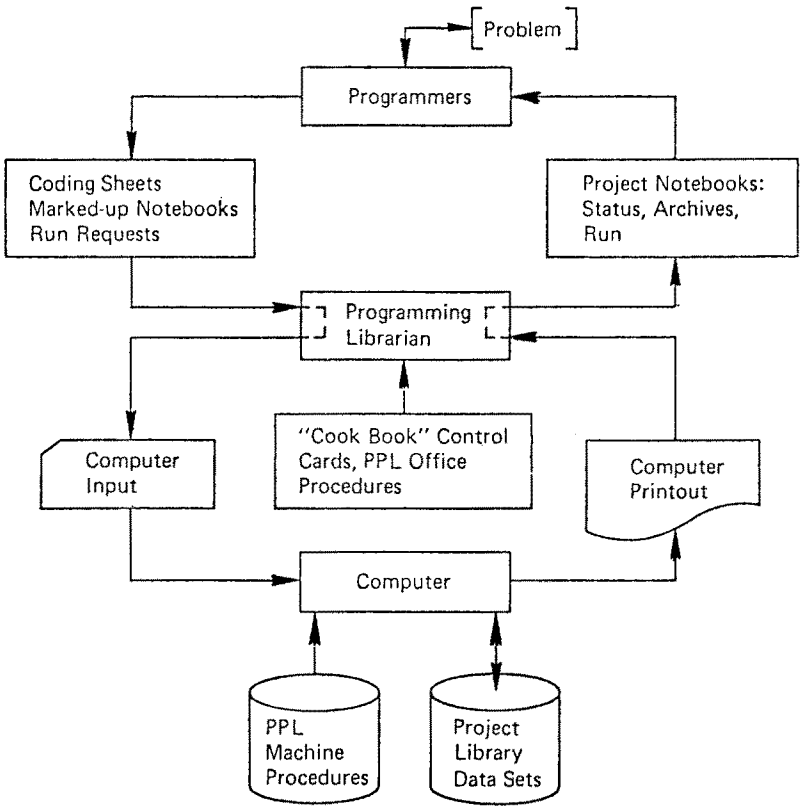
- JCL - Job control language
- LEL - Linkage editor language
- SOURCE - Source (PL/1, FORTRAN, BAL, COBOL) language
- TEST - Project test data
- OBJECT - Compiler output
- LOAD - Linkage editor output
- RUN - Execution output

A set of archive notebooks:

- For each of the above, plus
- General - PPL housekeeping output

PPL External Library

Figure 4



PPL Operations

Figure 5

Apollo

	Technologies Used	Bytes of New Code (Millions)	Total Effort to Delivery (Man-Months)	Productivity (Bytes per Man-Month)
Mission Operations Control	DSL	5.8	3748	1547
Ground Support Simulation	None	2.1	1809	1161

Skylab

Mission Operations Control	DSL	1.4	1665	841
Ground Support Simulation	DSL SP TDD	4.0	1065	3756

Productivity Comparison

Figure 6

Operation	Procedures	Functions
Initiating	PPLSTART*	Catalogs the project name and generates the SYSIN data set
	PPLSETUP*	Sets up space for a single PPL section on a specified disk pack
Updating	PPLENTER	Changes or adds members to any specified section other than OBJECT and LOAD
	PPLEDIT	Performs the same functions as PPLENTER and, in addition, provides for changing portions of statements and for shifting of statements either right or left
	PPLDELET	Removes one member from any section
	PPLINDEX	Provides a directory and VTOC listing for any specified section
	PPLJCL	Copies specified member from a project's JCL section to the installation common procedure library PPL.PROCLIB
	PPLJCLD	Deletes specified member from the project members in the installation common procedure library PPL.PROCLIB
	PPLMOVE	Transfers one or more members from any section, except LOAD, to a corresponding section of another project
	PPLCOPY	Creates a second copy of a member of any section, except LOAD, and gives the copy a new member name
	PPLPRINT	Prints out all members of a section
Processing	PPLBALSN	Invokes Assembler F to perform a syntax check on members of SOURCE written in System/360 Assembler Language (does not produce object code)
	PPLBAL	Invokes Assembler F to assembler members of SOURCE written in System/360 Assembler Language into members of OBJECT
	PPLBALLE	Linkage edits members of OBJECT derived from System/360 Assembler Language into members of LOAD
	PPLCBLSN	Invokes the ANSI COBOL compiler to perform a syntax check on members of SOURCE written in COBOL (does not produce object code)

* These procedures are normally used only by programmers.

PPL Machine Procedures

Table 1

Operation	Procedures	Functions
	PPLCBL	Invokes the ANSI COBOL compiler to compile members of SOURCE written in COBOL into members of OBJECT
	PPLCBLLE	Linkage edits members of OBJECT derived from COBOL into members of LOAD
	PPLFTNSN	Invokes the FORTRAN H compiler to perform a syntax check on members of SOURCE written in FORTRAN (does not produce object code)
	PPLFTN	Invokes the FORTRAN H compiler to compile members of SOURCE written in FORTRAN into members of OBJECT
	PPLFTNLE	Linkage edits members of OBJECT derived from FORTRAN into members of LOAD
	PPLPL1SN	Invokes the PL/I F compiler to perform a syntax check on members of SOURCE written in PL/I (does not produce object code)
	PPLPL1	Invokes the PL/I F compiler to compile members of SOURCE written in PL/I into members of OBJECT
	PPLPL1LE	Linkage edits members of OBJECT derived from PL/I into members of LOAD
Housekeeping	PPLCHKPT	Dumps all PPL sections of one project onto a specified tape
	PPLALLCT*	Closes up gaps between remaining members of any section to make room for additional members, or may be used to increase the space allocated to a section
	PPLSPACE	Closes up gaps between remaining members of any section to make room for additional members
	PPLRESTR*	Restores sections of a project from a checkpoint tape created by PPLCHKPT
Terminating	PPLCLEAN*	Deletes and uncatalogs any specified section of a project
	PPLENDUP*	Deletes a project's name from the system index

* These procedures are normally used only by programmers.

PPL Machine Procedures
Table 1 (continued)