Installation Management

Improved Programming Technologies — An Overview

IBM

GC20-1850-0

Improved Programming Technologies — An Overview

This document is intended to briefly describe to the reader six recently formalized techniques designed to improve the program development process: structured programming, top-down program development, chief programmer teams, development support libraries, HIPO (Hierarchy plus Input-Process-Output), and structured walk-throughs. These techniques are still evolving; initial use in a data processing activity should be subject to management review, to determine the form in which the techniques may best fit into each environment.

First Edition (October 1974)

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

Address comments concerning the contents of this publication to IBM Corporation, Technical Publications/Systems, Dept. 824, 1133 Westchester Avenue, White Plains, New York 10604.

Contents

Introduction

The last decade has been characterized by significant improvements in hardware speed and capacity, configuration flexibility, and programming system capability. There have also been many improvements in the capabilities of programming languages, but, in general, improvements in the techniques used in the program development process have lagged behind those in other areas. This period has also been characterized by the increasing complexity of application systems and by their importance to the organization. And, in the same period, application development, maintenance, and modification activities have comprised an increasing portion of the data processing budget. Data processing management, therefore, has been searching for ways to improve the program development process, with the objective of producing application systems that meet the needs of their users, are more error-free, require less maintenance, are easier to modify, and are developed on schedule with improved productivity.

This document describes six evolving techniques which have been implemented in various ways in some program development efforts within IBM and which may be of assistance in achieving management objectives. These techniques, some of which have elements that have been advocated or used in the past, are structured programming, top-down program development, chief programmer teams, development support libraries, HIPO (Hierarchy plus Input-Process-Output), and structured walk-throughs. The first four have frequently been used together in IBM's Federal Systems Division. HIPO and structured walk-throughs were developed separately and seem to logically complement structured programming, top-down programming, chief programmer teams, and development support libraries.

These techniques can be used individually or together. Since they are still evolving, their initial use in a data processing activity should be subject to management review, to determine the form in which they may best fit into its environment.

Chapter 1: Structured Programming

Traditionally, each programmer has applied his own set of rules to the construction of the logic of his program. He starts with this logic structure and, as he encounters additional combinations of conditions to be met, he adds them as afterthoughts rather than revising the logic of the program. The resultant control code might look like that shown in the left (Unstructured) column of Figure 1. This code contains a large number of GO TO statements and labels and its logic is not easy to follow. During subsequent unit and integration testing, disintegration of the programmer's original structure occurs as new constraints and conditions are imposed upon it—leading to more GO TO statements, more labels, and a final program whose original logic may be completely obscured. Reading, understanding, and testing such programs is difficult. The degree of confidence in their quality or correctness tends to be low. In addition, such programs tend to be difficult to maintain and modify.

UNSTRUCTURED	STRUCTURED
IF p GOTO label q IF w GOTO label m L function GOTO label k label m M function GOTO label k label q IF q GOTO label t A function B function C function label r IF NOT r GOTO label s D function GOTO label r label s IF s GOTO label f E function label v IF NOT v GOTO label k J function label k K function END function lable f F function GOTO label v label t IF t GOTO label a A function B function GOTO label w label a A function B function G function label u IF NOT u GOTO label w H function GOTO label u label w IF NOT t GOTO label y I function label y IF NOT v GOTO label k J function GOTO label k	①IF p THEN A function B function ②IF q THEN ③IF t THEN G function ④DOWHILE u H function ④ENDDO I function ③(ELSE) ③ENDIF ②ELSE C function ③DOWHILE r D function ③ENDDO ③IF s THEN F function ③ELSE E function ③ENDIF ②ENDIF ②IF v THEN J function ②(ELSE) ②ENDIF ①ELSE ②IF w THEN M function ②ELSE L function ②ENDIF ①ENDIF K function END function

Figure 1. A comparison of structured and unstructured code

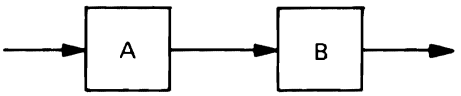
Research by computer scientists and mathematicians indicates that an alternative method of programming known as structured programming can help solve these problems. This technique involves coding programs using a limited number of control logic structures to form highly structured units of code that are more readable, and therefore more easily tested, maintained and modified.

Structured Programming Theory

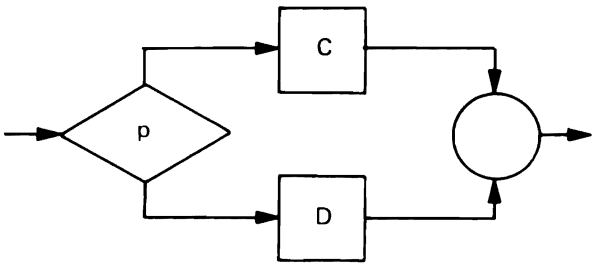
Structured programming is based on a mathematically proven structure theorem[1] which states that any program can be written using only the three control logic structures illustrated in Figure 2:

- Sequence of two or more operations (MOVE,ADD,...)
- Conditional branch to one of two operations and return (the IF p THEN C ELSE D of Figure 2)
- Repetition of an operation while a condition is true (the DO E WHILE q of Figure 2)

Sequence of two operations



IFTHENELSE: Conditional branch to one of two operations and return



DOWHILE: Operation repeated while a condition is true

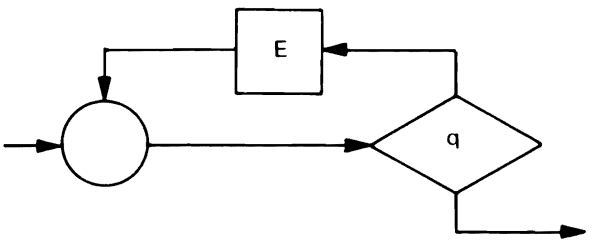


Figure 2. The three elemental logic structures of structured programming

Any program may be developed by the appropriate iteration and nesting of these three basic structures. Each of the three structures has only one entry and one exit. A program consisting solely of these structures is a proper program, a program with one entry and one exit. As illustrated in the structured code (right column) of Figure 1, it always proceeds from the beginning to the end without arbitrary branching. In PL/I, for instance, no GO TO statements are necessary. Proving the logical correctness of structured code is more feasible. The logic is easier to follow, permitting functions to be isolated, understood, and tested.

The use of the three control logic structures in structured programming is analogous to the hardware design practice of forming complex logic circuits from AND, OR, and NOT gates. This practice is based on a theorem in Boolean algebra which states that arbitrarily complex logic functions can be expressed in terms of basic AND, OR, and NOT operations. The use of three control logic structures in structured programming is similarly based on a solid theoretical foundation.

Extensions to the three basic logic structures are permitted as long as they retain the one-entry, one-exit property. An example of such an extension is the DOUNTIL structure (Figure 3), which provides for the execution of the function F until a condition is true.

DOUNTIL: Operation repeated until a condition is true

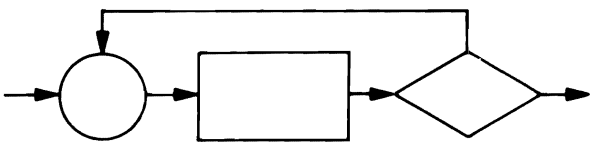


Figure 3. The DOUNTIL structure

Structured Programming Practices

Certain practices are followed to support the objective of producing readable, understandable structured programs—programs in which the writers can have a high degree of confidence.

Indenting within control structure blocks to reflect the logic of the program unit is one of these practices, as shown in the example of structured code in Figure 1. As illustrated by the number to the left of the statements, each logic structure nested within another is indented within it. All parts of a logic structure carry the same indentation level, and functions performed within a logic structure are indented within that structure. This practice highlights the

logic of the unit for the writer and the reader and thus contributes to the goal of more readable programs.

Limiting a unit of source code to a specified size—often one listed page, or fifty lines, permits the programmer to read and understand an entire logical expression or function without referring to multiple pages or relying on his memory. Should the complete logical expression require more than 50 lines of source code, the programmer can segment the code through the use of such statements as %INCLUDE (in PL/I) or COPY (in COBOL) to specify the inclusion of another unit of code (see Figure 4).

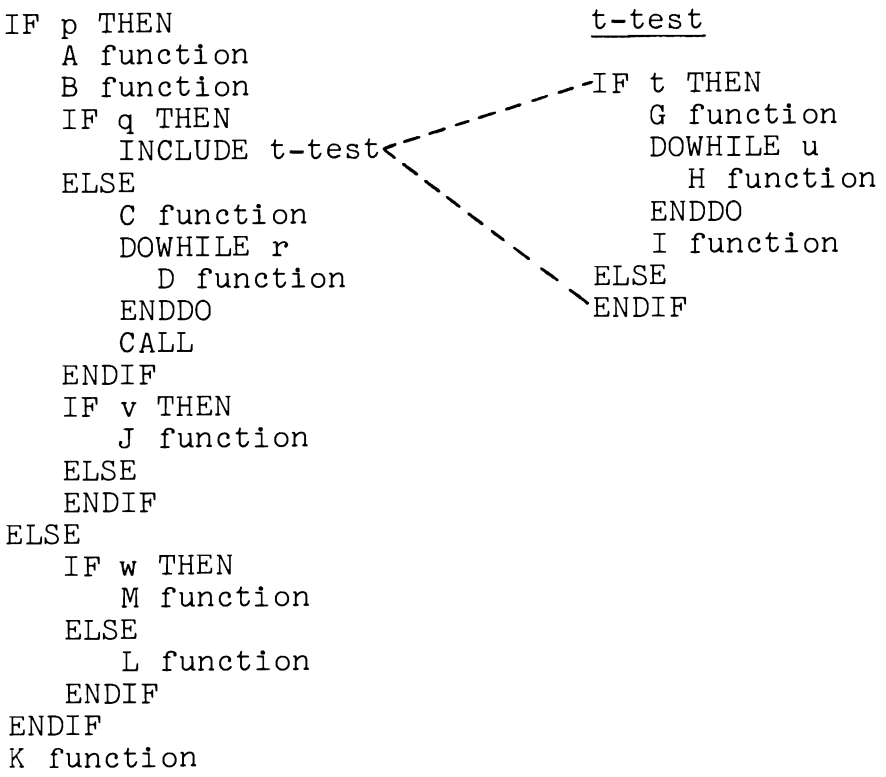


Figure 4. Segmentation

A graphic example of a program constructed of such units is shown in Figure 5. At the top are the job control and linkage editor statements that define the environment and major functions of the program. Subordinate to them is the hierarchy, or calling sequence, of the supporting units of code. Each unit specifies the invocation of the units immediately subordinate to it. Thus unit A would invoke units B and J, using COPY, CALL, PERFORM or %INCLUDE statements; unit B would invoke C and F; unit C would invoke D and E; etc. The next technique to be described, top-down program development, assumes such a hierarchical structure.

[1] Bohm, C., and Jacopini, G., "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules." *Communications of the ACM 9,* No. 3 (May 1966), 366-371.

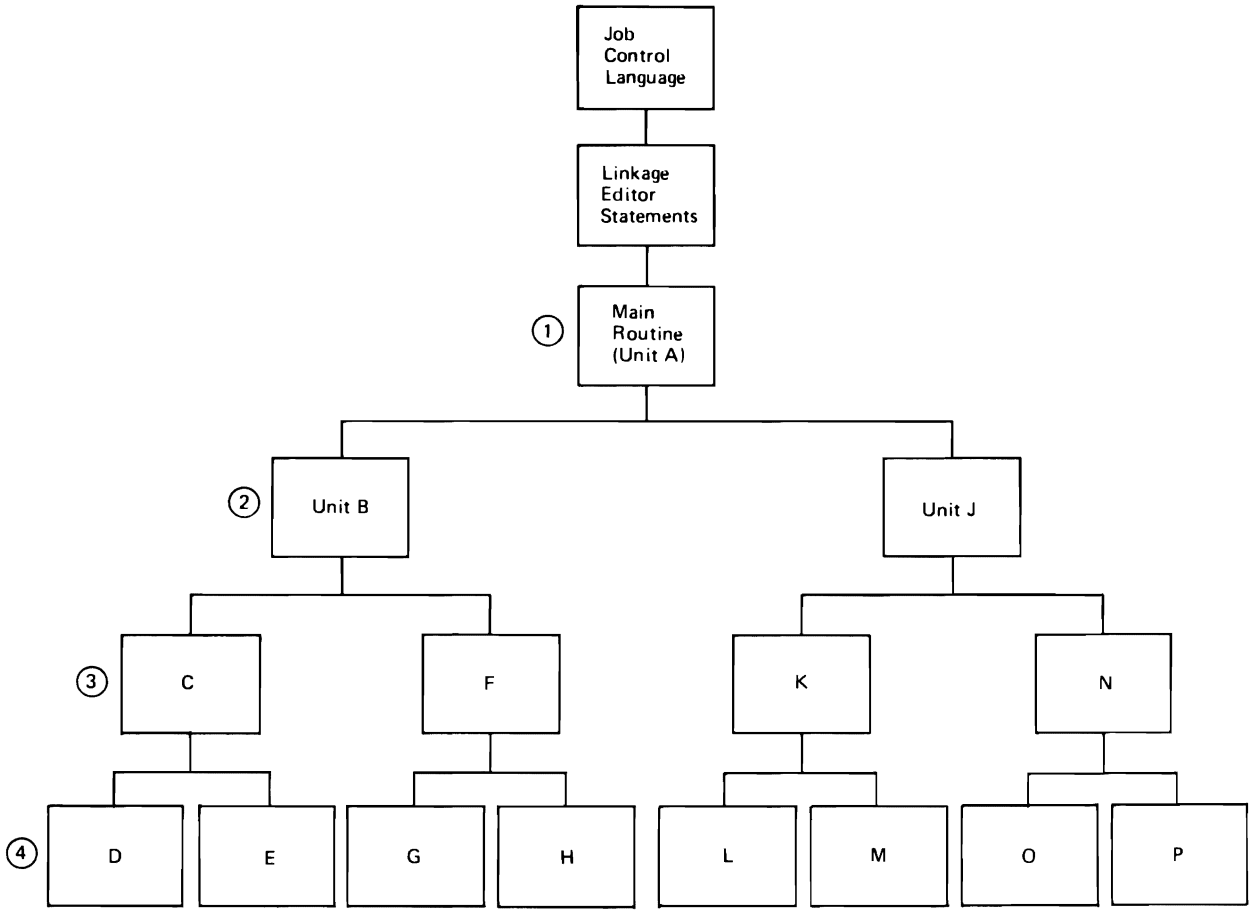


Figure 5. An hierarchical program structure

Chapter 2: Top-Down Program Development

Traditional software development has often been approached as a bottom-up procedure where the lowest level units are coded first, unit-tested, and made ready for integration (see Figure 6). Data definitions and interfaces between units tend to be simultaneously defined by each of the programmers, including those working on the lowest levels of code, and are often inconsistent. During integration, definitions and interface problems are recognized.Integration is delayed while the data definitions and interfaces are correctly defined and the units are reworked and unit-tested to accommodate the changes. It is often difficult to isolate a problem during the traditional integration cycle because of the difficulty in identifying which of the many units combined during integration is the source of the problem. The resultant program, because of last minute redesign, coding, and testing, is often lacking in quality. Superfluous code in the form of driver programs is needed to perform the unit testing and lower levels of integration testing. Management control is often ineffective during much of the traditional development cycle because there may be no coherent, visible product until final integration.

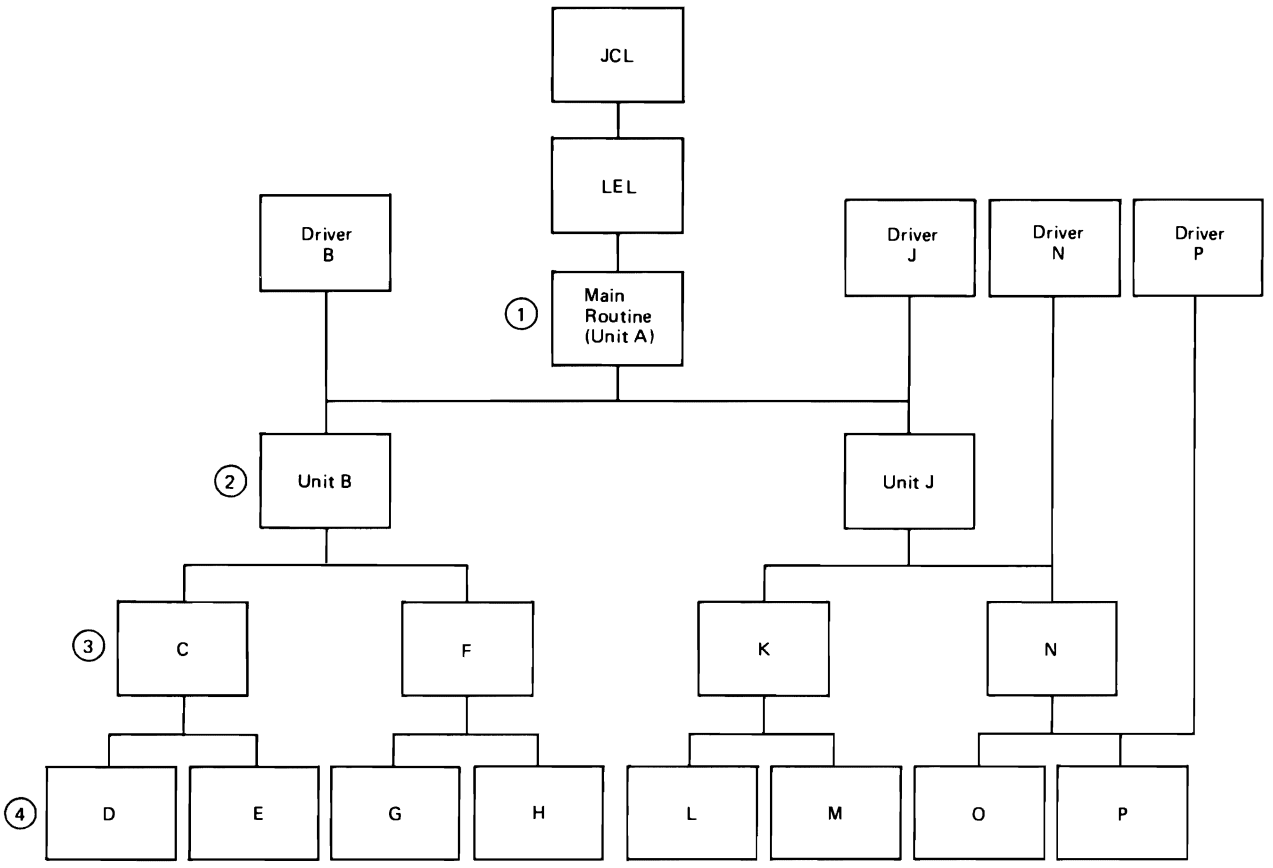


Figure 6. Traditional bottom-up development

Top-down program development is designed to reduce these problems by reordering the sequence in which units of code are written. A program unit is coded only after the unit that invokes it has been coded and tested. Therefore, top-down program development both assumes and is patterned after a program structure of hierarchical form as illustrated in Figure 5.

Figure 7 illustrates how the top-down approach is begun. Following program design, the job control

language (JCL), link-edit statements (LEL), and mainline routine (first level unit) are written (Figure 7A). Top-down programming then proceeds by writing the second level units (Figure 7B). While this is taking place, the logic of the first level unit can be tested by substituting dummy units (program stubs) for the second level modules. The program stubs do not normally perform any meaningful computations, but often produce a message to indicate to the tester that they have in fact been executed, thus testing the logic of the next higher level unit. The lower level units are built and integrated in the same manner, substituting actual program units for the program stubs until the entire program has been integrated and tested. The program is continually being integrated—with the higher level units, often the most critical, being the most frequently tested.

Using this method to implement a program design reduces the problem of hypothetical interfaces. Each interface is defined in code. In programming terms, this means not only that units of code are written in calling sequence, but that data base definition statements are written and data records generated before the code requiring those records is written.

Top-down program development permits test data to be generated in an incremental manner. For instance, when the mainline routine is tested, only the test data needed to test the system up to that point need be in readiness. As each subsidiary unit is tested and integrated, the test data needed to test those functions can be added.

The single starting point of top-down program development does not imply that the implementation must proceed down the hierarchy in parallel. Some branches intentionally will be developed earlier than others. For example, branches directly affecting user operations might be developed early in the cycle to permit early user training.

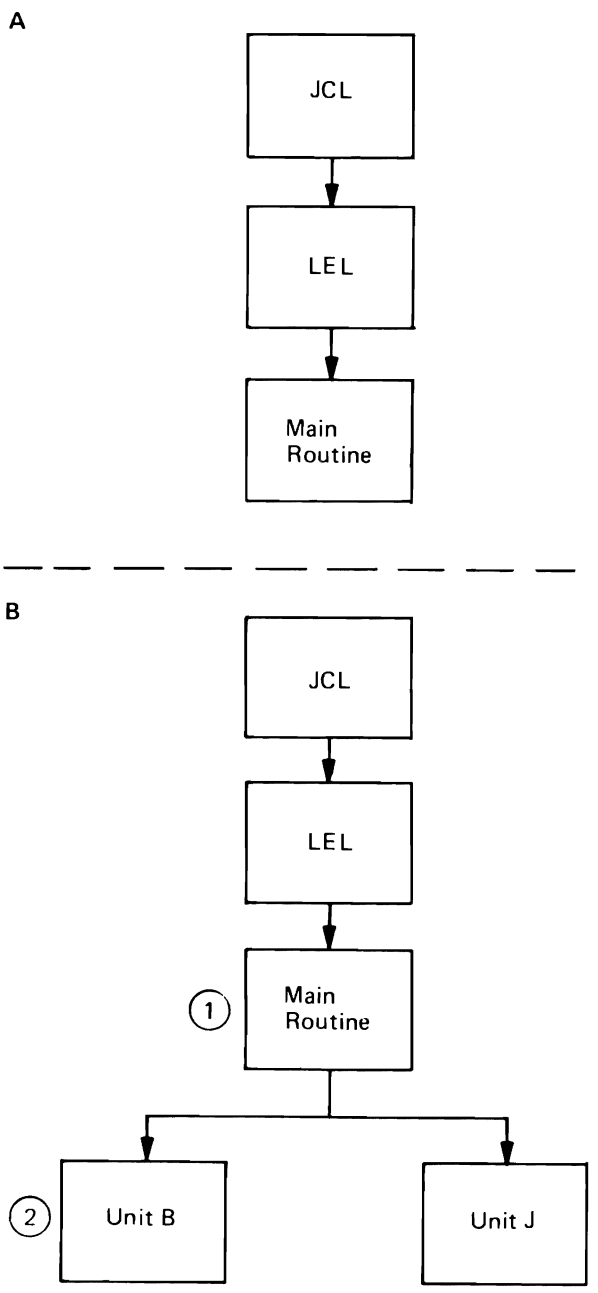


Figure 7. Beginning top-down program development

With top-down program development, the parts of a program and system are continually being integrated. A separate integration period does not exist (see Figure 8). Although it does appear to be theoretically possible that a project whose hierarchy diagram is narrow and long could experience an extended development cycle, it is expected that the cycle for other systems developed in a top-down manner should be no longer than for those developed in the traditional manner. In fact, projects using top-down development, structured programming, chief programmer teams, and development support libraries have described distinct productivity improvements along with improved program quality.

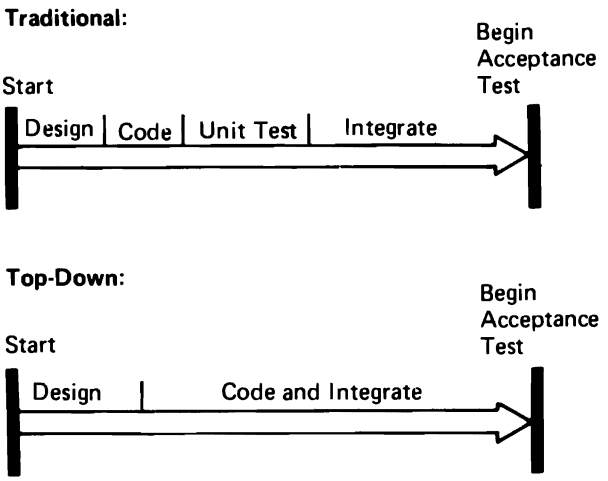


Figure 8. Effect on the development cycle of the traditional and top-down approaches

Chapter 3: Chief Programmer Teams

The increasing complexity of applications and major advances in hardware and software demand many advanced skills during the program development process. With increasing frequency, development managers find that applications and programs cannot be properly developed without a team effort. The chief programmer team is an organizational technique that complements the structured programming and top-down programming techniques, and is designed to coordinate the efforts of programming specialists while retaining the responsiveness and integrity of design expected of a skilled individual.

A chief programmer team is a small group of personnel, under the leadership of a senior level professional programmer called the chief programmer. It normally consists of three to five programmers, a librarian, and other specialists as appropriate. A chief programmer team represents an opportunity to improve both the manageability and the productivity of programming by moving the program development process from private art to public practice through an organizational technique that includes: restructuring the work of program development into specialized jobs that recognize the need for technical expertise in the leadership of the team effort and in the training and career development of its personnel; defining relationships among specialists; and using disciplines to help team members communicate effectively with one another and work effectively with a developing, always visible, project.

The Team Members

Chief Programmer

The chief programmer is responsible for program design of the system, and is vested with complete technical responsibility for the project. He writes the mainline routines, the critical code, and the operating system interfaces (job control language and linkage editor statements). He defines modules to be coded by other team members and is responsible for specifying the interfaces between modules and for the data definitions. He reviews code written by other team members and oversees the testing and integration of all code. He informs management of project status and arranges for additional team members, when necessary.

Since the chief programmer is the principal designer of the program, his duties begin early in the development cycle—while the program functional specifications are being formalized.

Backup Programmer

The backup programmer is a senior level programmer who works closely enough with the chief programmer on the tasks described above to be able to assume the chief's duties if necessary. He may be called upon to explore alternative design approaches, perform test and integration planning, or execute other special tasks. He is an active participant in technical design, internal supervision, and external management functions.

Librarian

The librarian is a dedicated team member who has the administrative skills necessary to handle the machine and office procedures involved in the coding and testing effort, as described in Chapter 4, "Development Support Libraries". The librarian is responsible for maintenance of project management statistics, and arranges for entry, compilation, and tests of programs as requested by team members. These responsibilities amount to a full-time job.

Other

Additional team members are scheduled into the team as required, as the development cycle progresses. They bring to the team such abilities as specialized application, hardware, or software knowledge, coding speed, or unique coding techniques.

Why Change to Teams?

The chief programmer team organization recognizes that program design is especially important in today's complex application environment and that it is best performed by a senior level professional programmer who also has responsibility for execution of that design.

Reintroducing senior people such as the chief and backup programmers into detailed program coding recognizes another set of circumstances in today's operating system environment. The job control language, data management access methods, utility facilities, and high level source languages are so powerful that there is both a need and an opportunity for using senior level personnel at this detailed, but critical coding level. The need is to make the best possible use of an extensive set of facilities. The functions of the operating systems are extensive and they are called into play by language forms that require a good deal of study and experience to utilize in the most effective manner.

The very definition of responsibilities in a chief programmer team forces a high degree of public

practice. For example, the librarian is responsible for picking up all computer output, good or bad, and filing it in the notebooks of the development support library, where it becomes part of the public record. Identification of all program data and computer runs as public assets, not private property, is a key principle of chief programmer team operations.

Chief programmer teams can provide the opportunity for professional growth and technical excellence in programming. Since functions involved in maintaining program data are the responsibility of the librarian, more time and energy can be devoted to developing key technical skills and to building the programs. Moreover, the close association with senior level programming personnel who review all code, its testing and integration, provides good training for less experienced programmers and can help prepare them for leadership in future teams.

Chief Programmer Teams in Large Projects

Large projects may require for their execution a number of chief programmer teams, with each reporting to a higher level team, and the top level team reporting to the project manager. The responsibility of each chief programmer team is defined by the structure of the program. Beginning at the top level, each team designs and codes a functional capability down to a set of program stubs. These program stubs become the assignments of the teams at the next level. Each next level team continues the design and coding, possibly to a new set of program stubs, until all the coding is completed. Each chief programmer is directly responsible for the members of his own team and for the chief programmers of the teams under him.

Over the life of a project, the upper level chief programmer teams go through definite phases of responsibility, e.g., design, code, test, and certify. These phases are nested between levels, in that the program stubs produced by a design at one level trigger the next level process, and the certify phases include verification that the program stubs have been carried out satisfactorily by that next level. The top level team will complete its design, code, and test phases early and will spend the remainder of the project certifying the contributions of lower level teams to the system. Each succeeding level starts a little later and has less certifying to do, until the lowest level teams simply design, code, and test their own programs. Note that the team structure mirrors the program structure. In this way, the integrity of the program structure can be preserved during the detailed coding process.

Chapter 4: Development Support Libraries

The development support library (DSL) function supports the environment created by structured programming, top-down programming and chief programmer team organization. It can also be used apart from these techniques. The technique consists of office and machine procedures used by a librarian to maintain units of structured code being tested and integrated. It is designed to promote efficiency and continuous product visibility during the program development cycle.

Basic Elements and Method of Use

A development support library function (outlined in Figure 9) consists of four elements: a machine-readable internal library, a human-readable external library, machine procedures, and office procedures.

The internal library contains all current project programming data, including program modules, linkage-editing statements, job control statements, and test information. The status of the internal library is reflected in the human-readable *external library* binders which contain current listings of all library members and archives consisting of recently superseded listings. The *machine procedures* consist of standardized JCL and utility control statements to perform such basic procedures as the following:

- Creating and updating libraries
- Retrieving modules for compilations and storing results
- Linkage editing jobs and initiating test runs
- 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 by programmers in the external library
- Using machine procedures
- Filing updated status listings in the external library
- Filing and replacing pages in the archives.

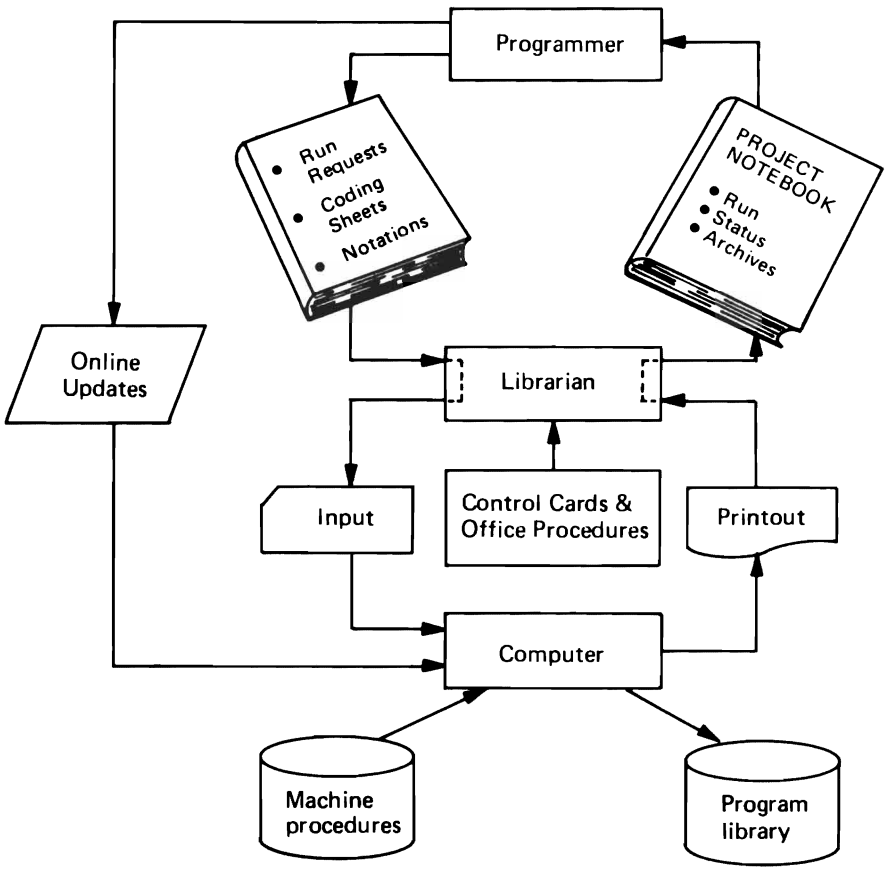


Figure 9. Flow of operations with a system development library

As shown in Figure 9, a programmer using a DSL prepares coding sheets and run requests. He submits them to the librarian, who arranges for the library create or update run. This generates the current version of the program in machine readable and printed form. The librarian places the printed version into the program's external library binder.Later, the programmer receives these updated binders, which reflect the new status of the internal library. If interactive program development is used, the external library update may be generated at log-off time. Programmer-requested printouts plus copies of all code changes are sent to the librarian, who files them in the external library.

The programmers are freed from such tasks as handling decks and interacting directly with operations, and thus can make more effective use of their time. In addition, a development support library function contributes to manageability, productivity, and program quality by making possible a project whose developing components are visible and available to all, including management. It permits programmers to be certain of the data definition and interface requirements, as well as the operational details of other program units by reading the actual code in the external library rather than by having to refer to a separate set of documents that may lag behind actual status.

Additional Library Facilities

Other facilities that users might consider for inclusion in their development support library function are:

- *Indentation listing.* Indents structured programming source statements to improve program unit readability.
- *Standards checking.* Checks source statements for adherence to installation standards.
- *Stub handling.* Provides the ability to support top-down programming by generating program stubs with a debug or trace capability and, if desired, routines to simulate time and/or storage to be used by the unit that will replace the stub.
- *Multiple project libraries.* Permits the existence of separate versions of a program. For instance, one set of libraries can contain program units under development while a separate set can contain an operable developing system with which tested program units will be integrated. Still other libraries may be used for operational systems.
- *Program hierarchy listing.* Shows the module calling sequence.
- *Management control listings.* Provides for the collections of statistics on program size, number of changes, compilations, tests, etc.

Chapter 5: Hierarchy plus Input-Process-Output (HIPO)

Application function documentation is often addressed towards the end of a project, and then described with prose, creating a twofold problem: (1) description of function is often imcomplete because of the difficulty in extracting the function of a system from the bit manipulation performed by the programs, and (2) prose descriptions of function are often voluminous while remaining ambiguous and without a systematic means of relating them to the program modules performing the function. HIPO helps solve these problems by providing the designer with a graphic technique designed for documenting function from the beginning, before programming starts and while it is clear in the designers' minds. It is also designed to reduce the ambiguity and the amount of prose required to document function, and to provide a systematic means of identifying all the functions to be performed and the modules that perform them.

In describing the functions to be performed, HIPO diagrams progress from a generalized functional description to greater levels of detail. The functions themselves are described in terms of the process that occurs, with its necessary inputs and resultant outputs. A HIPO package consists of a set of functionally oriented diagrams from generalized to more detailed descriptions of function. Specifically, a typical HIPO package consists of one or more overview diagrams, detail diagrams, and a visual table of contents.

The overview and detail diagrams describe function graphically, with each diagram consisting of three parts: (1) input - the inputs to the function (files, records, fields, control blocks, etc.), (2) process - the process steps that support the function being described, and (3) output - the outputs of the process (files, records, control blocks, etc.) (see Figure 10).

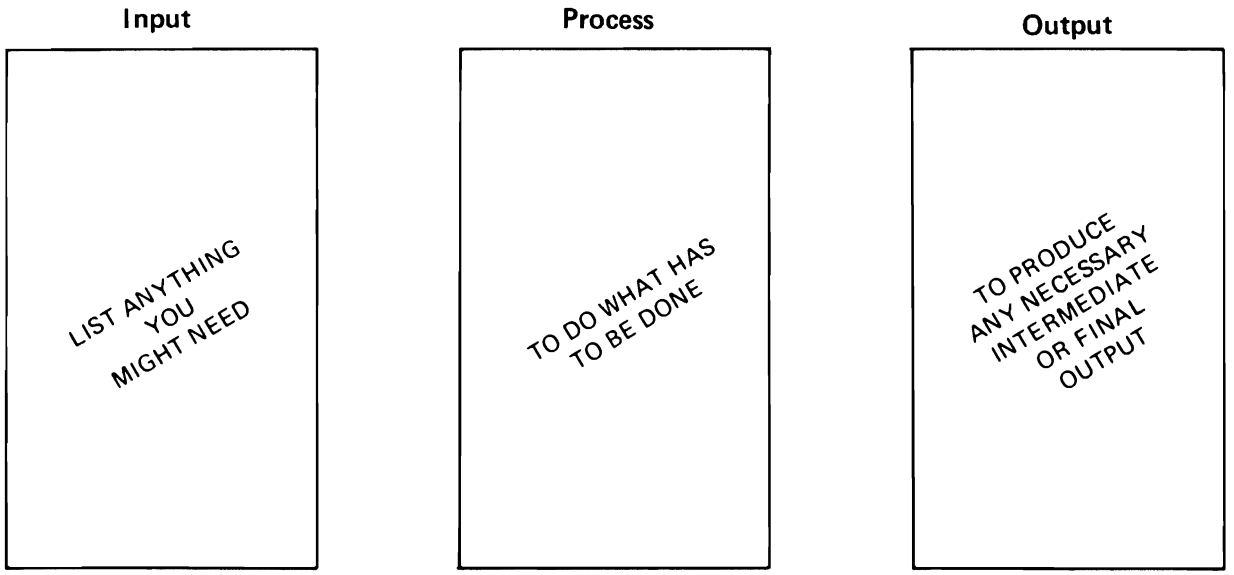


Figure 10. Input-Process-Output graphic relationships

An overview diagram describes, in general, one or more functions expanded by detail diagrams.Figures 11 and 12 illustrate an overview diagram and a detailed diagram, respectively.

In addition to the input, process, and output sections, each detail diagram includes an extended description section, keyed by numbers to the process section. In this section each numbered process may be described in more detail and can point to the program module or modules in which the process is implemented and to the module(s) calling the process. For an overview diagram, the extended description section can further describe each process and may also point to detail diagrams where the numbered processes are further expanded.

Diagram 2. Calculate Gross Pay

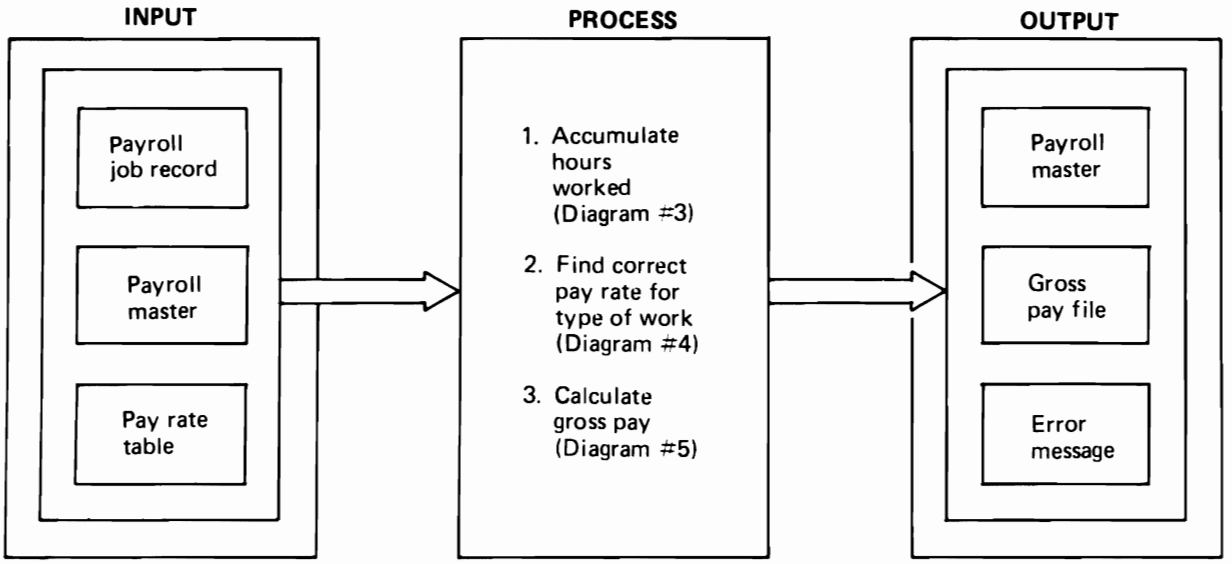
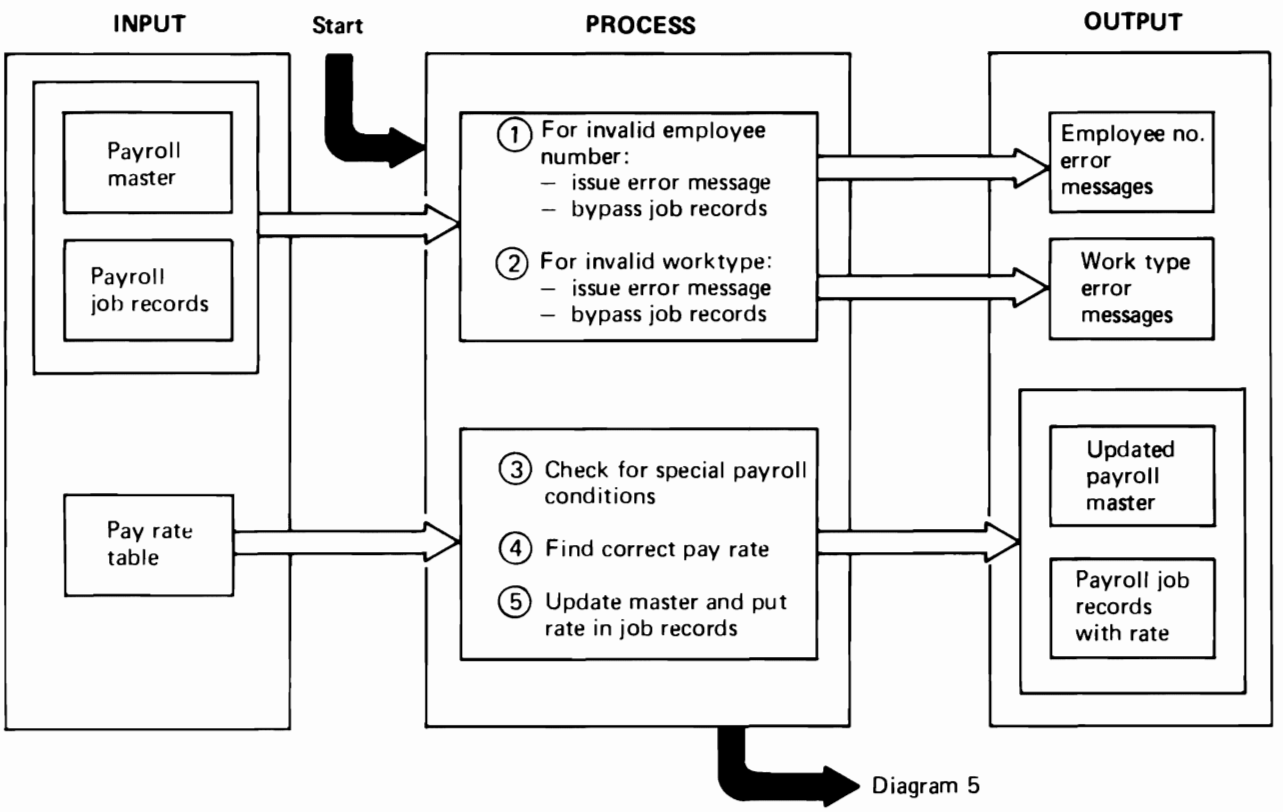


Figure 11. Overview diagram

Diagram 4. Determine Pay Rate



	Extended Description	Routine	Label
1.	The program checks for valid employee number. If valid, job records for that number are bypassed and an error message is printed.	IODNA	DETR
2.	A check is made for correct type of work. If invalid, bypass job records & print error message.		
3.	Special conditions such as overtime, shift pay, vacation pay, or holiday pay are checked to help determine correct rate.		
4.	The master record, job records, & pay rate table are all referenced to determine correct pay rate.		
5.	When all conditions are checked, payroll job records are rewritten with proper rate, payroll master updated.		

Figure 12. Detail diagram

The visual table of contents (see Figure 13) identifies all the overview and detail diagrams in the package, shows their hierarchical relationships, and permits the reader to quickly locate a particular level of information or a specific diagram.

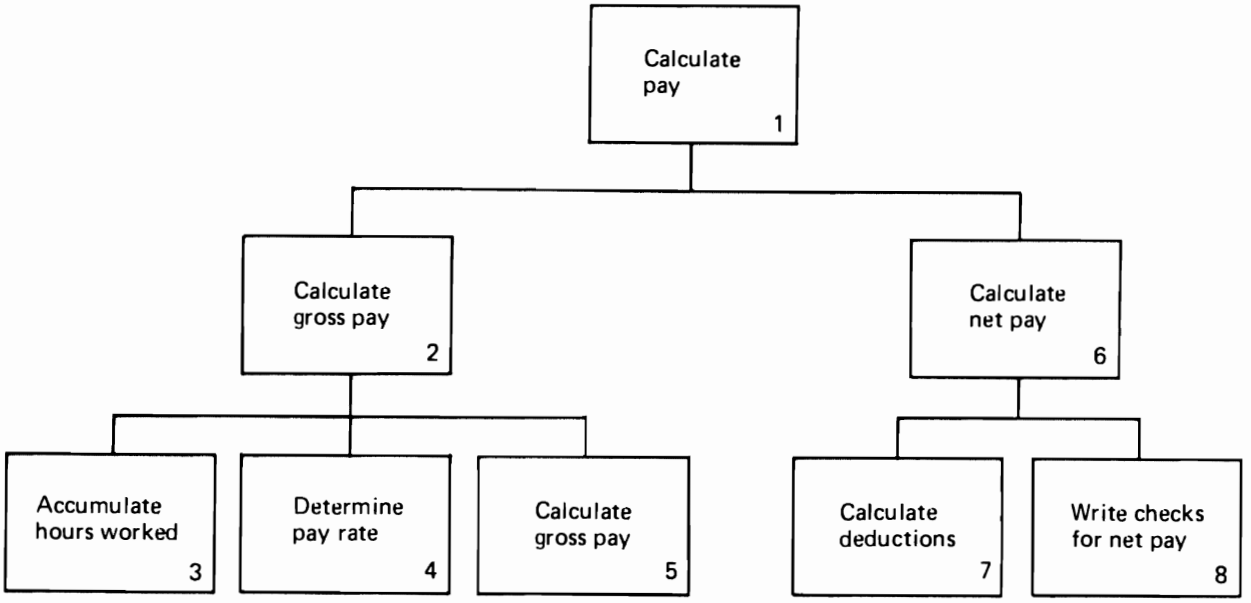


Figure 13. Visual table of contents

As shown in Figure 14, HIPO diagrams can be used throughout the development cycle and after its completion. HIPO documentation evolves throughout the development cycle from an initial design package, to a detail design package, and finally to a maintenance package. The initial design package, prepared by a design group at the start of a project, describes the overall functional design of the project and is used as a design aid. The detail design package is prepared by a development group. Using the initial design package as a base, analysts and programmers design in detail, add more levels of HIPO diagrams, and use the resulting package for implementation. The maintenance package, frequently identical to the detail design package, serves as the final documentation for the system.

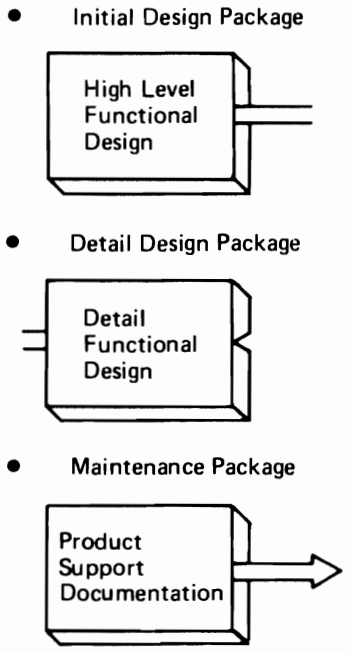


Figure 14. Types of HIPO packages

HIPO can help answer the requirements of the many types of people who rely on the documentation of a system. A development manager, for example, may want a system overview that is under-

standable to a user. An application programmer can use the documentation to determine the detailed programming requirements. A maintenance programmer requires documentation that quickly identifies functions to which changes must be made, and the modules that execute them.

HIPO may be used apart from or in conjunction with the other techniques described in this text. Because of its hierarchical depiction of function, it very effectively supports the hierarchical structures assumed in top-down program development and structured programming.

Chapter 6: Structured Walk-Throughs

Sometimes program errors result from the lack of experience of the designer or programmer (developer). Probably more often they result from the lack of perspective of the developer. He has been too close to this program for too long and finds it difficult to see any errors in it. And typically, programmers hesitate to ask either for guidance or for a check of their program's logic or completeness because they feel it to be an implied admission of incompetence, even though most programs, somewhere, contain a new challenge to their originators. Yet it is important to detect and remove errors as early in the cycle as possible when the cost of correcting them is lowest and their impact is smallest. The structured walk-through is designed to detect and remove errors as early as possible in the cycle in a problem-solving and non-fault-finding atmosphere in which everyone, and especially the developer, is eager to find any errors in the work product being reviewed.

A structured walk-through is a review of a developer's work (program design, code, documentation, etc.) by fellow project members invited by the developer. It is conducted by the developer and, in most instances, is not attended by his manager. These reviews help the developer find errors in his work earlier in the development cycle. In addition, they give reviewers an opportunity to learn new approaches and techniques. Structured walk-throughs also help the participants communicate the characteristics of their developing work to each other.

Structured walk-throughs can be used at various checkpoints in the development cycle to review each part of the system as it is developed in more and more detail. For instance, they can be used to review:

- Project plans and schedules
- System specifications
- Program functional specifications
- Program design (control structure)
- Detailed program design
  - Data specifications
  - Module interfaces
  - Documentation
- Coding (uncompiled source listings)
- Final documentation
  - User guides
  - Program maintenance manuals
- Finished product

Basic Characteristics

Structured walk-throughs, in various forms, are being used by some program development groups within IBM. The basic characteristics of one form are:

1. It is arranged and scheduled by the developer of the work product being reviewed.
2. Management does not ordinarily attend the walk-through and it is not used as a basis for employee evaluation.
3. The developer selects the list of reviewers but, in most cases, management reviews the list to ensure that developers of related work products will be invited. The walk-through is usually attended by four to six reviewers. Participants can include:
   - Developers of other parts of the system
   - Developers of other systems that interface with the one being reviewed
   - Testers responsible for component and system testing
   - Designers of the system to ensure compatibility and continuity of design
   - Individuals responsible for documenting the function being reviewed
4. Every walk-through should have a defined set of attainable objectives.
5. The reviewers are given the review materials four to six days prior to the walk-through and are expected to review them and come to the session with a list of questions.
6. The walk-through is structured, in the sense that all attendees know what is to be accomplished and what role they are to play.
7. A moderator, frequently a project team leader, is appointed or elected to chair the session. This individual insures that the walk-through stays on course. He compiles an action list consisting of all errors, discrepancies, exposures, and inconsistencies uncovered during the walk-through.
8. All issues are resolved offline. The walk-through provides problem *detection*, not problem resolution.

A typical walk-through is scheduled to last for a specified period of time, not longer than two hours. If the session's objectives have not been met at the end of the time period, or if a significantly large list of issues has been created, another walk-through is scheduled for the next convenient time.

Procedure

First, the reviewers are requested to comment on the completeness, accuracy, and general quality of the work product. Major concerns are expressed and identified as areas for potential follow-up. The developer then gives a brief tutorial overview of the work product. He next "walks" the reviewers through the work product in a step-by-step fashion, attempting to satisfy the major concerns expressed earlier in the meeting. Usually this includes examining the work product with test cases prepared by the developer. Thus the test cases as well as the work product are "walked through". New concerns may arise during this "manual execution" of the function.

Immediately after the meeting, the moderator distributes copies of the handwritten action list to all the attendees. It is the responsibility of the developers to ensure that the points of concern on the action list are successfully resolved, and that the reviewers are notified of the actions that have been taken and/or the corrections that have been made.

An essential ingredient for a successful walk-through is the proper attitude on the part of all participants. The reviewers should be concerned with error detection rather than error correction. The developer must have an open and nondefensive attitude to make it easier for the reviewers to find errors. He should welcome their feedback and encourage their frankness. It is difficult to have such an attitude if the developer feels that he is being evaluated by his manager on what occurs during the walk-through, and on the size of the action list. In this situation, he may tend to suppress criticism and become defensive and unreceptive to questions about his work product.

Relationship with Other Techniques

Structured walk-throughs can be used independently of the other techniques described in this text, or in an environment in which one or more of them are used. Structured walk-throughs seem to fit quite naturally with them. The visibility, the idea that code is meant to be read by others, the conventions, and the simplified program logic of structured programming make it easier for the reviewer to be "walked-through" code segments. Because in a top-down program development and chief programmer team environment, the chief and backup programmers design and code the top of the system first, their initial walk-throughs can, for the other team members, serve as an introduction to the system and a means of learning their senior programmers' design and coding techniques. Finally, HIPO's graphic representation of application function lends itself to walk-throughs both of function and of the code that fulfills the function.

GC20-1850-0

Installation Management Manual Improved Programming Technologies - An Overview Printed in U.S.A. GC20-1850-0

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)