EDP ANALYZER
©1977 by Canning Publications, Inc.

NOVEMBER, 1977
VOL. 15, NO. 11

USING SOME NEW PROGRAMMING TECHNIQUES

In the early 1970s, IBM began to market a group of programming aids called IPTS—Improved Programming Technologies. These methodologies include top-down development, structured programming, chief programmer teams, HIPO (Hierarchy plus Input-Process-Output), pseudo-code, development support library, and structured walk-throughs. In addition, IBM offers several programming facilities on TSO (Time Sharing Option), including a COBOL interactive debugging facility. In this issue, we shall look at some user experiences with these techniques, and see how they have been adapted and modified to fit different needs.

We have chosen to review the offerings of IBM in this area of programmer support products because they represent a body of technology to which a large number of data processing installations have been exposed. A major push for the use of programming aids has come from IBM. IBM employees have published many articles and given numerous speeches, and the three IBM Systems Science Institutes in the U.S. give about 40 seminars a year on the use of these techniques.

As a group, IPTS complement each other and are intended to be used together, throughout the entire software development effort. Given all of the exposure that IPTS have received, we were interested in seeing if users have adopted these techniques as advertised, or whether they have only implemented certain ones. Thus, we shall describe some uses of IPTS as examples of how users might improve the software development process.

This is not to say that these IBM offerings are the only ones on the market. Software houses, education firms, consultants, and other mainframe manufacturers offer similar products and education. We have discussed a number of these other techniques in past issues. For example, see the list of past subjects on the last page of this report.

We categorize the use of IPTS in three groups: for designing, for building and for testing software. We expand the group of IPTS a bit to include interactive debugging and testing.

This is how we categorize these aids:

For designing software:
- Top-down design
- HIPO
- Pseudo code

For building software:
- Development support library
- Structured programming
- Top-down programming
- Chief programmer teams

For testing software:
- Structured walk-throughs
- Interactive debugging and testing.

We will briefly discuss each one of these techniques. Readers interested in more details should obtain the IBM general information manuals, listed under Reference 1. In addition, we will refer to selected other literature that gives recommendations on the use of these techniques.

For designing software

In design we are talking about two distinct types of design. One is functional design—designing *what* the system will do. This is for the users' benefits, so that they can verify that the system will do what they want it to do. The other is logic design—designing *how* the system will operate. This is for the programmers' benefits, so they can write the code.

Top-down design

The original IPTS did not include a top-down design discipline *per se.* However, IBM subsequently included "structured design," which is one form of top-down design, in its Systems Science Institute curriculum. Structured design is based on concepts expressed by Larry L. Constantine in 1964 and expanded in the paper by Stevens, Myers, and Constantine (Reference 2a).

Top-down design—sometimes called the levels of abstraction approach or the stepwise refinement approach—begins by defining the major functions of the system. Decisions pertaining to lower level sub-functions are delayed as long as possible. Each major function is decomposed into its constituent sub-functions.

Constantine's concepts include the system's executive functions at the top of a hierarchy of subordinate modules. The system is designed for minimum coupling between modules, so that each module is as free standing as possible. Further, each module is designed for maximum functional strength, where all the elements of the module contribute to performing only one function.

HIPO

HIPO (Hierarchy plus Input-Process-Output) is a design and documentation technique from IBM. It uses two types of diagrams. The H chart is a hierarchy chart of the functions of a software system. These functions state *what* is to be done, not *how* it is to be done. The functions in the chart go from general (at the top) to specific (at the bottom). The connecting lines between the function boxes do not show flow of control but rather show the decomposition of the functions into subfunctions.

The second element is the IPO chart. It describes each function by its inputs, processes and outputs. A function box on the H chart is represented by one IPO chart, with the various inputs, the processes on those inputs, and the resulting outputs listed for that function. This IPO information is then used to create the next lower, more detailed level on the H chart. So the lowest levels of the H chart are the most detailed, most decomposed subfunctions.

Pseudo code

Pseudo code, metacode, and program design language are synonyms for a structured, natural language notation used during software design. It is an informal method of expressing logic, using an indented format to show control structure. It uses terms permitted in structured programming, such as DO WHILE, DO UNTIL, ELSE, IF, and CASE. Thus, it can be readily translated into compilable code. It is often described as being useful for detailing structure and IPO charts.

The authors who discuss structured design, HIPO and pseudo code claim that these techniques complement each other and can be used for both functional design and logic design. Thus, they can be used to aid communication with the user as well as make coding better correspond to design.

For building software

F. Terry Baker (Reference 3) has described his experiences in implementing IPTS in a production programming environment. We will give some of his views in connection with building software.

Development support library

A development support library (DSL) is the collection of all information pertinent to a software development project. It includes current and backup versions of the programs (source, object, and load), the operating system instructions, test data, results of tests, documentation, history of the code, project performance data, and library procedures. A major purpose of the DSL is to keep all project records at one place.

The DSL is maintained by one or more librarians (who may also perform secretarial or other functions) and perhaps by an automated library package. For effective use, the DSL must be up to date and its contents must be readily available to the analyst and programming staff. Analysts and programmers may access the DSL by terminals, in some environments.

Structured programming

Structured programming embodies two concepts, according to Baker. One is structured coding, i.e. developing and testing individual modules of code. The objectives are that each module be given an intelligent name, be about one page of source code in length, and contain functions that logically fit together. Each module should have one entry and one exit, and coding should be restricted to three control structures—sequence, alternation and repetition. Some organizations do allow very limited use of GO TO. Indentation conventions should also be used to show the structure of the program on the source listing. The purpose of structured coding is to make programs easier to read and comprehend as well as easier to debug and maintain.

The second aspect of structured programming, according to Baker, is that team members can and should read each other's code, and that all exceptions to the structured coding standards be thoroughly documented and have management approval.

Top-down programming

In top-down programming, modules are coded, tested and integrated basically in execution order, beginning with the top level. The control functions are normally called the top level; the basic utility routines are normally on the bottom level. Once the top level has been developed and tested, the next level can be worked on. Stubs are written, for testing purposes, to represent unwritten modules in the lower levels.

The purpose of top-down programming is to ease the system integration problem, and distribute testing, integration and computer usage more evenly throughout the development cycle. And the necessity for writing drivers to test low level modules, as required in bottom-up construction, is mostly eliminated.

Chief programmer teams

A chief programmer team may be the hardest IPT to implement, states Baker, and thus, it should be implemented last. A chief programmer team is a software project structure that consists of a chief programmer, a backup programmer, a number of subordinate programmers, and a project librarian. The chief programmer has technical responsibility for the project, backed up by the backup programmer. Together these two people design the system and then code the key portions, usually the top control level and any other particularly difficult modules. They also assign the work to the other team programmers, and review the code of these members. The librarian, who is an integral part of the team, maintains the project DSL and provides secretarial services for the project.

For testing software

The two IPTs that pertain to testing software are structured walk-throughs and interactive debugging and testing.

Structured walk-throughs

Structured walk-throughs are technical reviews of design, coding, test plans, and so on, performed by peer groups. Each walk-through has its own specific objective. A walk-through is called and led by the designer or programmer whose work is to be reviewed; copies of this work are distributed to team members before the walk-through. Walk-throughs are normally attended by no more than six people.

The times in the development cycle when walk-throughs are normally performed are: 1) after preliminary design, to study the layout of the module hierarchy; 2) after detailed design, to study the logic in the modules and the file organization; and 3) after coding, but before compilation.

The emphasis of the reviews is on error detection, *not* error correction. Following a walk-through, a listing of the errors detected is distributed (by the librarian) to each project member. The designer or programmer is then expected to make the needed changes to correct these errors.

Interactive debugging and testing

Interactive debugging and testing differs from interactive (or on-line) programming in that the programmer uses the terminal only *after* his code has been entered into the system. He does not initially enter code at the terminal himself. It may be entered by a data entry clerk or via keypunch. Once the code is in the system and has been compiled, the programmer uses the on-line terminal to test and debug it.

These then are the techniques we are discussing. To describe how they are being used, we begin with the experiences of TRW DSSG.

TRW DSSG

TRW Defense and Space Systems Group (DSSG) is a division of TRW, Inc. DSSG provides software for governmental agencies and private industry. Their headquarters are in Redondo Beach, California, a suburb of Los Angeles, where the staff includes over 1000 system analysts and programmers. We talked with people in management systems who provide business data processing services to DSSG on two IBM 370/158s.

Early in 1975 the people within the training function of management systems spent six months evaluating the newer programmer productivity techniques that we have just described. During this time they ran a small pilot project, visited companies that were using these techniques, attended relevant seminars, and read the trade literature.

The most disturbing problem they encountered in their study was that no one was able to show them a project that had modular traceability from start to finish. Modules in the final programs were not the same modules created in the design phase. The loss of traceability during development caused them to wonder if these techniques really would help them better control the development process.

From the study, the team recommended that structured design, structured programming, pseudo code and HIPO be used in the department. Strict top-down design was not recommended, because it did not address the problem of ordering data structures, especially in a multi-application environment. Also, it did not allow a careful study of lower level interfaces that were highly technical. So a parallel bottom-up and top-down design approach was recommended.

Chief programmer teams were not recommended because the study team found them to be the least utilized and conceptually weakest of the techniques. And they recommended further study on the DSL before making a decision on whether to use it or not. The team's recommendations noted that following training, a learning curve of two to three months would be required to fully utilize the recommended techniques.

After management accepted these recommendations, in-house training courses for both project managers and technical staff members were designed. These were given to project teams just before they began a new project where these techniques would be used. Project managers were given a one-day course first; technical staff members were then given a four-day course tailored to the management systems environment. Two days were devoted to top-down, structured design, and two days were spent on structured programming. Audio visual courses were also made available to supplement the training, technical guidelines were issued, and a technical library was started.

The training people found that most of the staff members were curious enough about the new techniques to take the training with an open mind. Members of four major projects have been trained on these techniques so far. Management systems uses the new techniques as follows.

In the design phase, once the requirements and specifications have been approved by the user, structured design is performed, using structure charts and data path analyses. At critical points during the design phase, structured walk-throughs are held. These last anywhere from one-half hour to three hours and involve from two to four team members. At the initial walk-through, the structure, definition and interaction of the modules are studied. At subsequent walk-throughs, data flow analyses and the packaging of functions into modules are studied. Limiting structure charts to one page helps participants easily grasp problem solutions, we were told. For any set of functions or modules, they have found that three walk-throughs are needed during the design phase.

Once the design is complete, the designer translates each module into low level pseudo code. Logic that was previously described verbally is now written down. The pages of pseudo code are cross referenced to the module boxes on the structure chart. The people in management systems are very pleased with this use of pseudo code. Using it in conjunction with a structure chart and a data flow analysis provides them enough documentation that they no longer use HIPO charts. Their HIPO charts had been large and cumbersome, and they did not greatly increase the communication with users.

The conventions of structured programming that are emphasized within management systems are: 1) modularization of source statements into blocks of about 70 lines of code, 2) logical indentation of source code to show structure, and 3) limiting statement design to constructs with one entry and one exit point.

All of the programmers who have used structured programming in management systems are enthusiastic about it, and the results of its use are being seen. During unit testing, performed by another programmer, many fewer errors are being found. The same is true in acceptance testing. And when an error is found, locating the place to institute the change is easier, due to the constructing conventions.

In 1976, TSO became available to the programmers. They now can either code on coding sheets, and then have the code keyed into the system by a data entry clerk, or they can key the code into the system themselves. One problem with structured programming, that the use of TSO has eased, is fitting modifications into the indented format. With TSO the program is displayed on a CRT, so changes can be fit easily into the existing structure.

One benefit of these new techniques has been the ease of bringing new people into a project team. On two occasions team members have left, once during the design phase on one project, and once during final debugging, just before implementation, on another project. In both cases, the new members were able to quickly comprehend the project and begin their work.

Documentation for the project is kept on a module basis, with a unit development folder created for each module. The folder contains (1) the design structure chart, showing where the module fits into the project hierarchy; (2) the data flow analysis, showing the input/output requirements of the module; (3) the pseudo code; (4) the test plan; (5) a log of changes; and (6) the structured code. Standard numbering and naming conventions are used in all documentation and programs. Thus, management systems is able to show traceability of modules, from design through coding.

All in all, management systems at TRW DSSG is well pleased with their use of the newer programming techniques, and they plan to use these techniques on all new projects. With a major emphasis on the design phase and design tools, they are finding that the projects are progressing more steadily than in the past.

Columbus Mutual Life

Columbus Mutual Life Insurance Company, with headquarters in Columbus, Ohio, is a multiline company with about $3 billion in force. The company provides both individual life and health as well as group life and health insurance. For its data processing, the company uses an IBM 370/145, together with IBM's ALIS (Advanced Life Information System). Programming is done mostly in COBOL, although ALC, FORTRAN, and PL/1 are also used.

We visited Columbus Mutual to learn about their use of the development support library. The DSL played a very key role in their conversion to ALIS, we were told, and it continues to play an important role.

The company began considering the use of ALIS in 1970. In late 1971, it was decided to go ahead with the system. During 1973, the company declared a moratorium on most new system development, as well as on changes to existing systems, in order to adapt ALIS to its needs. Most of the company's staff of nine programmers were assigned to the project. In addition, Columbus Mutual obtained contract programming services from IBM and a local service bureau.

At the time, Columbus Mutual was using the Librarian package from Applied Data Research. Librarian proved to be very helpful for controlling the creation of the ALIS support programs, plus changes and extensions to the ALIS package. One problem, however, was that three staff members were spending a good amount of time carrying card decks to computer operations and printouts back to the programmers, delaying turnaround. These three staff members were secretary/librarians for the programming staff. About this time, ADR announced its ROSCOE remote job entry system. Columbus Mutual obtained ROSCOE and tied it in with Librarian. This step eliminated the need for the secretary/librarians to carry the materials to and from computer operations, substantially reducing turnaround time.

At the peak of the effort of tailoring ALIS to its needs and developing the support systems, Columbus Mutual had 30 programmers working on the project. With the combination of Librarian and ROSCOE, and the three project secretary/librarians, it was possible to keep up with the changes and development of the program. As production got underway in January 1975, in only three cases the first year did production cycles fail to run to completion, where it was necessary to go back to the beginning and rerun the cycle.

All source code is stored under Librarian. For maintenance, production source code is copied to create a test version. All changes to the test version are "temporary" until the program is compiled, tested and approved by the user. Then Librarian is used to make the changes permanent. Further, Librarian provides a configuration history for each program, showing the changes that have been made previously. The source code listings are the most recent approved generation of the source code, along with the printouts of prior changes.

Columbus Mutual feels that it is getting the following benefits from its DSL. The source code is secure. There is no worry about computer operations erroneously using an outdated version of the program. Nor is there worry about the only copy of the source code being destroyed. Code is accessible for review and there is an audit trail of all changes made to each program.

The data processing manager at Columbus Mutual summed up his feelings as follows: "We wouldn't do our program development and maintenance any other way than by using the support library function, now that we have used it so successfully."

Mellon Bank

Mellon Bank, with headquarters in Pittsburgh, Pennsylvania, has assets of over $9 billion. The bank is rated sixteenth in size among the largest U.S. banks, according to *Fortune* magazine. For its data processing, the bank uses three IBM 168s each with 4 megabytes of memory and operating under MVS, plus two 135s and a number of minicomputers. The development staff numbers 154 people.

Mellon Bank is one of the most automated banks in the country. The development of new application systems is given high priority. New techniques that improve the efficiency of building and maintaining application systems are tested out regularly.

When the series of IPTS was announced by IBM, Mellon Bank studied the techniques thoroughly. The results of this analysis were interesting. The bank had an extensive set of installation standards that were being practiced. They found that they were already doing something very close in concept to the chief programmer team idea. The program design methods they were using were reasonably close to structured programming. And so it went with the other techniques analyzed. Mellon Bank has tried essentially all of the IPTS and finds that they like some and are not too happy with others.

The best point at which to begin the discussion of Mellon Bank's experience with IPTS is with their in-house training program—the Advanced Technology Seminars.

Advanced technology seminars. Prior to 1974, conventional staff training methods were used. These included the use of video tapes, attending computer manufacturer training courses and outside seminars, and so on. Management began to see a need for continuing education of more up-to-date or more advanced knowledge.

In 1974, Mellon Bank began a series of weekly advanced technology seminars. These seminars are held on Thursday mornings and are about three hours in length. Each seminar is limited to 25 people; if the subject is very popular or important, the seminars are repeated. Notices are sent out to the six groups within the development staff as to the subject of the forthcoming seminar. People sign up based on their interest in learning about the subject—or their desire to criticize it. Project leaders may encourage some of their people to attend, but attendance is not forced. Actual attendance is then allocated among the six groups. Seminars are either taught by members of the staff or are panel discussions involving staff members.

These seminars have proved to be a very effective way to expose staff members to new technology. Essentially all IPT methodologies have been taught and discussed in this manner.

Chief programmer team. The bank had been using a project team approach that was very similar to IBM's chief programmer team approach, except that the bank did not make use of project librarians on the teams. To support the programming staff, the bank had two experienced data entry people who could not only enter programs but could also correct some input errors and could request compiles. The bank does not separate systems people from programming people, nor does it separate development work from maintenance work. A project leader is a top programmer and is essentially equivalent to a chief programmer. The bank has had quite good results from the use of this approach.

Top-down design. Members of the staff have tried top-down design with varying degrees of success. Results seem to depend on the interest of the individual project leaders.

HIPO. The results from using HIPO are somewhat the same as with top-down design, although HIPO is not used as much. The staff has modified HIPO charts to incorporate a visual table of contents on each chart. Some staff members consider HIPO to be weak as a programming logic tool, and awkward to use in going from the chart to code.

Structured programming. Each new programmer joining Mellon Bank goes through a training program in the use of the bank's installation standards and programming practices. Each programmer must write eight COBOL programs as part of this training. A programmer is encouraged (but not required) to use structured programming in writing all but one of these programs. Two self-study texts are used to provide the needed training. So each new programmer is familiar with structured programming practices from the outset.

Structured walk-throughs. This technique has proved to be very useful. It is used for reviewing system specifications, system design, program design, and coding. The appropriate group of people is selected for each of these reviews. Mellon Bank finds that structured walk-throughs (1) provide better communication with users, (2) give better communication with computer operations, (3) provide project leaders with a clearer picture of how actual accomplishments agree with plans, and (4) show project leaders what still must be done. Structured walk-throughs are not yet used by all development groups, but the use is expanding.

Development support library. At the time the IPTS were announced, Mellon Bank was already using an automated library system for controlling programs under development and in production. The bank has continued to use this technique.

Interactive debugging and testing. Mellon Bank has 32 CRT terminals to support 90 programmers—about one terminal for every three programmers. Interactive debugging and testing are done via TSO. Programs are coded on coding sheets by programmers and then entered by the data entry specialists. Programmers put testing aids into the programs, for monitoring progress through programs and for detecting the cause of abnormal terminations.

The bank was not too happy with TSO under the previous operating system that was used. But with MVS, results have been much better. A programmer can now get three to five test shots per day, as compared with less than two test shots per day previously. By comparing the time required to test new systems (where TSO can be utilized fully) with the time needed to test old systems (where TSO may not be a good choice as a testing strategy), management can see somewhere between 3 to 1 and 5 to 1 improvement in the man-hours needed to do a maintenance job.

No attempt is made to closely control the use of these terminals. The rule is: "When you think you need to use a terminal, use it. When you are finished, get off." Management finds that the programmers do an effective job of self-policing in the use of the terminals.

The best performers. Among the IPTS that they have tried, the people at Mellon Bank see the best performers for them as being the chief programmer team, structured walk-throughs, and interactive debugging. The other techniques are used to some extent. In addition, the bank continues to look for new ways to improve staff productivity.

Training and implementation

A major problem with using IPTS is implementing them. Getting people to change their way of doing things is not easy. Barry Boehm (Reference 4) gives three recommendations for implementation: (1) carefully plan the introduction of the new techniques, (2) only tackle a couple of the techniques at one time, and (3) train everyone involved well, *before* the use begins. Substantial education, management, practice, and encouragement are required for staff members to unlearn their old habits, he points out.

The most typically successful approach for implementing IPTS that we found is the pilot project.

One, or a few, senior programmers and a manager take a course and then use the newly-learned techniques on a pilot project. Following this, standards for the department are created, and then more staff members are trained. If only junior programmers are trained initially, probably nothing will come of the effort; the use will not spread.

The role of IPTS is most often discussed in terms of developing new software. But these techniques also have a role in system and program maintenance—although opinions differ on the scope of this role. Many say that the use of IPTS for maintenance usually means starting over again, rather than fixing up existing code. At the IBM Systems Science Institutes, they recommend developing a new modular structure for a program being maintained and then creating structured code. However, Parikh (Reference 8) feels that some IPTS can be used effectively for modifying or enhancing a system without restructuring it—as well as for restructuring a system to improve its maintainability.

Charles Holmes (Reference 4) gives a good case study on the dos and don'ts of implementing IPTS. He describes two attempts at McDonnell-Douglas Automation Company to implement structured programming, chief programmer teams, an on-line production library via TSO, top-down programming and structured walk-throughs. On the first attempt, all of these methodologies were tried at once. First, three chief programmers spent five months defining project standards. Then 20 programmers were given a two week training course on structured programming and chief programmer teams as well as a one week course on TSO. Two computer operators were given a one week course on using TSO to maintain a DSL. The new techniques were then to be used on four program modification efforts and one small development project. The implementation failed, says Holmes, "because we were too ambitious and because we lacked coaching and the visibility of the experiences of other companies."

On their second implementation attempt, formal training was given in phases. First, a two week course on structured programming, top-down programming, pseudo code and structured walk-throughs was given. Several months later, after the staff had a chance to use these techniques, courses on chief programmer teams and the use of the library were given. TSO training was postponed to a later date.

The people at McDonnell-Douglas Automation Company are pleased with this second approach. Implementation statistics show that projects are being completed more quickly, and they feel the quality of their software has improved.

What we found seems to coincide with what Boehm recommends. He says that the typical pitfalls are going too fast and expecting too much. He recommends tempering enthusiasm with careful preparation and adequate training.

Experiences with the use of IPTs

What, in general, have been the user experiences with IPTS? Here is a summary of what we found from our interviews and our search of the literature. In the area of design, our references to the literature draw upon Boehm, Katkus, and Gordon, all in Reference 4, and upon McCoy in Reference 5.

For designing software

The major impression that we received from talking to IPT users was that the design phase is the most crucial stage in the development process. The biggest payoffs will come from improving the quality and depth of designs—leading to an easier and faster programming effort, less testing, fewer integration problems, and decreased future maintenance.

Top-down design. Structured design, which is one type of top-down design, appeared to be the IPT that was of the most current interest. It is relatively new and is not yet widely used. Those who are using it are pleased with it. The aspect of structured design that people find the most useful is the ability to iterate the design to find the best solution. Requiring people to do a lot of thinking about the design of the system is an important step in itself for improving software quality.

Boehm points out three pitfalls that can occur during top-down design. One is using strict top-down design on a problem where high risk, low level functions may exist. In this case, he recommends doing a risk analysis of such modules to see which are the riskiest and should be designed in more detail before proceeding. A second pitfall of pure top-down design is the inability to identify and combine bottom level common routines or utilities. The third pitfall is that the top-down control structure commits a project to a specific hardware system early, before the hardware implications of various design decisions are completely understood. Boehm points out that all three of these problems can be resolved by full top-to-bottom design reviews. A similar recommendation comes from Stevens, Myers and Constantine, as well as the people we talked to at TRW. They recommend keeping the design of a program on one sheet of paper. By doing this, common modules, incompatibilities and the probable effect of changes can be more easily recognized.

HIPO. In our study we found that the use of HIPO, as supplied by IBM, was mixed. In cases where it has been tried and discarded, the same information is still being used, but in a different format. It is claimed that HIPO can be of use as a design tool, to improve communication with users, as documentation, and as an aid for maintenance. We shall briefly discuss each of these uses.

Stephen McCoy states that HIPO charts are fine for defining major program functions; however, they give a limited and disjointed overview of what the program as a whole is doing. HIPO ignores the sequential nature of programming and thus it leaves inexperienced programmers floundering. He also says that the condensed format of HIPO makes it difficult to estimate the degree of complexity and the amount of coding involved. Several people we talked with do find HIPO useful for design, but say it is awkward to use when going to coding. Thus, for detailed design, they rely on flow charts. The people who do use HIPO for detailed design use pseudo code in the IPO charts, to identify what is being done and what subroutines are called.

On using HIPO to improve communications with users, the people at TRW found that it helped them little. Possibly, they said, this was because they presented the users with too much detail or used too many currently popular "buzz" words. Gene Katkus, on the other hand, found that HIPO did improve communications between programmers and engineers. He said that HIPO made modular design easy to review and correct.

Those who are using HIPO during design plan to save the diagrams as documentation and to aid in future maintenance. Combined with pseudo code and structure charts, and based on their development experience, they feel that these documents will be much better, for maintenance purposes, than what they had used in the past.

Pseudo code. Interestingly, pseudo code is one of the techniques most highly recommended by the people we talked with. They recommend using it in conjunction with HIPO and/or structure charts. They state that its use makes the design modules more detailed. And this precision and readability allows a more concentrated design effort. It is this ability to get into the details of the modules with pseudo code that is so highly praised.

Gordon reports that after numerous design iterations using pseudo code and top-down design, when his group finally began coding, they felt they were rewriting an existing program. They had confidence in the code and so they coded large chunks of it before any testing was done. So they even avoided stub writing. And, although he reports that the number of lines of code written per day did not increase much, he feels that they solved the problem with many fewer lines of code—reducing machine time during development and writing better quality code.

The other major contribution of pseudo code that was pointed out to us is that it allows a smooth transition from design to coding. The modules are so well defined that coding is done much more quickly.

For building software

For our discussion of building software, we draw upon Boehm, Katkus, and Romanos, all in Reference 4, as well as on Stay (Reference 2b), Baker (Reference 3), and Holton (Reference 6).

Development support library. We found the DSL, as defined by Baker, to be one of the least used IPTS; but those who use it would not be without it. The comments on DSL vary from "We really don't need it" to "It would have been impossible to implement the system as quickly or as well without a library." We found that companies often use part of the library concept, but do not put everything together at one location, as recommended by Baker. For example, in several cases, the program data entry task has been turned over to clerks for on-line entry. These people are often able to catch programming syntax errors as they enter the code. And they often request code compilations. Automated library packages are also used to track projects.

Holton did a survey of 23 larger companies in the Los Angeles area. He found that 12 of those companies use computerized library packages for software development. But none have full-blown DSLs, nor do any of them have librarians on software project teams.

One benefit of the DSL that the people at Columbus Mutual pointed out to us is that it prevents programmers from possibly making modifications to obsolete versions. Source code is out of the hands of the programmers, and this is good from management's viewpoint.

On Columbus Mutual's large project, which had 30 programmers at one point in time, they had three secretary/librarians on the project. With this combination, they were able to provide librarian backup. For smaller projects, providing backup may be a problem.

Another problem often mentioned is: "Where can we get a librarian?" The one answer most often heard is: "Don't use a programmer." A librarian who begins to code will cause conflicts and reduce the team's efficiency. Columbus Mutual is happy with their choice of a former clerk and two former keypunch operators.

The conclusion we come to is that companies who have what they feel is an adequate development support system, be it manual or automated, do not feel the need to change to a DSL. But we noticed that on large projects especially, it might be a worthwhile technique to investigate.

Structured programming. In his survey, Holton found that 14 of the 23 companies surveyed were using some form of structured programming. The consensus among these users was that it probably increases processing costs, but it does produce better quality programs, makes maintenance easier, and makes more efficient debugging and testing. We found essentially the same reactions to structured programming.

One of the benefits that Baker points out is that structured programming is useful in the virtual systems (VS) environment. 4K byte modules are desired because they fit onto one page in a virtual system. Also, structured programming encourages locality of reference, keeping frequently used code together and keeping the main line together. These all make more efficient use of virtual systems.

Two problems with structured programming were noted to us. One is the training problem, and the other is the increased use of computer resources in running structured code.

The companies we talked with were aware that the introduction of structured programming would initially increase programming costs—creation of new programming standards, staff training, and initial lower productivity. However, they did mention that the learning curve was short, when proper training was given. Boehm noted that the problem of getting team members to review each other's code did not seem a problem when the structured walk-through approach was used. However, we did not hear of any project managers who review their team's code.

On the increased use of computer resources, the overhead of modules is the increased execution time and memory space needed to effect the module call. James Romanos reports that on one project that was measured, structured code took 6-10% more memory and run time was increased by a small percentage. But he notes that these increases can be reduced by monitoring the programs to identify the most active or most inefficient pages. Stay states that the most active modules can be rewritten for efficiency. And the most active pages can be fixed in main storage. Or modules can be combined into logically related pages. However, before this tuning can occur, the program will use more memory. This should be taken into account during development.

Top-down programming. In our visits we found people attempting at least some top-down programming. In his survey of 23 Los Angeles firms, Holton found that 14 were doing top-down development (design and programming). He noted that these firms saw more of a payoff from these techniques than from the use of structured programming.

Both the people at Hughes Aircraft Co., as reported by Katkus, and the people at TRW did concurrent top-down, bottom-up programming. At Hughes, the bottom modules that were coded first were the complex ones. Katkus reported that the incremental demonstration of the system was very effective, because of the top-down programming. As each lower level was added, the entire system was tested, to verify that these modules did not adversely affect the operation of the completed modules.

Katkus reports that top-down development puts a lot of pressure on the programmers doing the top two levels of modules. He states that this is good, because it is better than putting pressure on lower level modules. He found that they did have to write some drivers to test bottom level modules, when some second level modules were delayed.

Chief programmer teams. We saw a number of variations of the chief programmer team concept. In only one case did the company seem to follow closely the concepts for these teams, as proposed by IBM. Even in this instance, the "chief" of the team need not be the most skilled programmer on the team but rather is the person with the most combined management and programming capabilities. Further, the concepts are not used for all projects.

The adaptions of the technique that we saw differ from the pure chief programmer team in two ways. First, the teams do not use a project librarian. Second, the chief is a chief designer rather than a chief programmer. This reflects an emphasis on the design phase rather than the coding phase. The chief manages the design to maintain its integrity. We found that these chief designers do varying amounts of actual design work. Most often they produce the design specifications, and often they do the functional design. Sometimes they also do the detailed design; if not, then they assign, co-ordinate and review detailed designs, interfaces and data definitions done by others.

The major pitfall, of course, is picking an inappropriate chief. The answer to this that we heard most often was to use the concept only when the personnel mix on the project seemed well suited to the technique.

For testing software

Our discussion of testing will draw upon the papers by Katkus (Reference 4), McCoy (Reference 5), Holton (Reference 6), and Freeman (Reference 7).

Structured walk-throughs. In our interviews with people, we found structured walk-throughs to be very well received. In fact, they were the most highly touted technique, and were used by the largest number of people. In Holton's survey of 23 companies, he found 14 used structured walk-throughs. These firms reported that walk-throughs tended to improve staff morale and produce more team cohesion. They also generally felt that the walk-throughs contributed to faster implementation, better quality programs, and produced clearer and more useful programming documentation.

In the literature, walk-throughs are described as a programming review technique. We found, however, that many people think they are more effective as a *design* review technique. Whereas they might need one walk-through during the coding phase, they would need three during the design phase. In fact, Katkus noted that, in his group, code walk-throughs were not even required, because the systems had been so thoroughly reviewed during the design walk-throughs. He found that all designs required at least two walk-throughs. Some required more, not because they were poor designs, but rather because of the complexity of the system or the misunderstanding of the requirements. Those designs that required more walk-throughs eventually produced better code, in his opinion.

Peter Freeman notes that walk-throughs also perform a forcing function—they force people to get work done by a specific time. And the education of other staff members is a by-product. They may be likened to (or used as) a tutorial on a particular design aspect.

One point often made about structured walk-throughs is that management should not attend them, because they may use walk-throughs for employee performance appraisal purposes. We found several users who agreed with this position. But Katkus noted that, in his experience, programmers liked the project manager to attend, because they felt that the manager obtained a better understanding of the software and of the details of their work.

So, all in all, structured walk-throughs were found to be a very useful technique. McCoy noted, however, that they can become less effective if too many are held or more than 3 to 5 people attend any one. He found that they are useful, up to a point.

Interactive debugging and testing. The people we talked to who used interactive debugging and testing were very pleased with its easy implementation. They found its use was quickly accepted and led to immediate productivity gains. They also found it to be useful in managing projects. The system used by these people is IBM's TSO, used in conjunction with the COBOL debugging facility and the structured programming facility.

We were told that the newer versions of these products have greatly enhanced productivity of the programmers. They display information by scrolling, they are quick, and they provide helpful utilities. IBM also offers HIPODRAW, a facility for drawing HIPO charts, and SCOBOL, which supports structured programming in COBOL.

Conclusion

As we mentioned, IPTS have been widely marketed and discussed. Their use now appears to be growing, with this use not being restricted to only IBM shops. The people at the IBM Systems Science Institute say that their five-day programmer productivity techniques course is open to everyone, and non-IBM users do attend. The techniques taught there, and discussed in this issue, are not proprietary to IBM (except TSO). So their use need not be limited to IBM users.

A major benefit of the use of these techniques, we are told, is that they make the system development staff think more—during design, building and testing. And these techniques encourage team thinking and reasoning, which companies are finding to be very beneficial in problem solving. Each of the IPTS is being put to good use somewhere. They are not equally widely used, and no one that we contacted is using all of them, yet. But companies seem to be gradually experimenting with them, one by one, keeping some, modifying others, and discarding the rest. We think that the approach taken by management systems at TRW DSSG is a good approach. They studied the techniques and first implemented those that seemed to fit in with their environment. They stressed to us that trying to implement too much at once can result in chaos. Further, strong management support and understanding are necessary.

People we talked with were genuinely impressed (if not somewhat surprised) with the gains they had made in their software development process through the use of certain IPTS. But the choice of which techniques to use and how to implement them played a key role in their successful use. And they expect that the best is yet to come, with decreased future maintenance on these structured systems.

Prepared by:
Barbara C. McNurlin
EDP Analyzer Staff

REFERENCES

General information on IPTs

1. IBM general information manuals; order by number from local IBM office:
   a. Code reading, structured walk-throughs and inspections, GE 19-5200, $2.70
   b. Improved programming technologies—an overview, GE 19-5086, $2.70
   c. Structured programming in COBOL, GC 20-1776, $2.00
   d. Structured programming in PL/1, GC 20-1777, $2.00
   e. Structured programming in FORTRAN, GC 20-1790, $1.90
   f. HIPO—A design and documentation technique, GC 20-1851, $3.20
   g. OS development support libraries, GC 20-1663, $1.10
   h. TSO:3270 structured programming facility (SPF), GH 20-1638, $1.80
   i. COBOL interactive debug facility, GC 28-6454, 65 cents
   j. HIPODRAW, SH 20-1728, $3.20
   k. SCOBOL, GK 10-6089, $1.40

Using IPTs

2. *IBM Systems Journal* (IBM, Armonk, New York 10504), price $1.75 each:
   a. Stevens, W. P., G. J. Myers, and L. L. Constantine, "Structured design," Vol. 13, No. 2 1974, pp. 115-139.
   b. Stay, J. F., "HIPO and integrated program design," Vol. 15, No. 2 1976, pp. 143-154.
3. Baker, F. Terry, "Structured programming in a production programming environment," *IEEE Transactions on Software Engineering* (IEEE Computer Society, 5855 Naples Plaza, Suite 301, Long Beach, California 90803), June 1975; pp. 241-252; price $10.00.
4. "Structured programming: A quantitative assessment," (6 papers), *Computer* (IEEE Computer Society, address above), June 1975, pp. 38-54; price $6.00.
5. McCoy, Stephen M., "Structured programming: Miracle or mirage?" *Journal of Systems Management* (Association for Systems Management, 24587 Bagley Road, Cleveland, Ohio 44138), August 1976, pp. 10-11.
6. Holton, John B., "Are the new programming techniques being used?" *Datamation* (1801 S. LaCienega Boulevard, Los Angeles, Calif. 90035), July 1977, pp. 97-103.
7. Freeman, Peter, "Toward the improved review of software designs," *Tutorial on Software Design Techniques*, (IEEE Computer Society, address above), price $12.
8. Parikh, G., "Improved maintenance techniques," Shetal Enterprises (1787 B West Touhy, Chicago, Illinois 60626); 1977; price $6.00.