EDP ANALYZER

FEBRUARY, 1979
VOL 17, NO. 2

THE PRODUCTION OF BETTER SOFTWARE

As we have indicated in recent issues, we think you will be hearing a lot more about improved system development methodologies in the months ahead. Numerous new methods are in field use, and their benefits and deficiencies are becoming apparent. In our review of what is emerging, we discussed (in last month's report) two methods for analyzing user needs. In this report, we discuss some other methodologies that support the analysis phase as well as system design. And next month, we will address program design. We believe that you should be familiar with the similarities and the differences among these leading methodologies. Here is what we see happening.

The goal in software development is to create software that performs reliably, meets user requirements, and does nothing that it is not supposed to do. But this goal is still not being realized in much of today's software, we gather. The problem centers around system complexity and the difficulties that arise because of this complexity.

We have addressed the question of how new methodologies attack this problem of system complexity—in our November 1977, February and March 1978, and in last month's reports. As we see it, the methodologies typically have the following components for handling complexity.

Disciplined approach. Complexity seems to be best handled by adopting a top-down approach to analysis, design, and construction, using successive decomposition. (Other names often used for this are functional decomposition and levels of abstraction.) Coupled with this top-down methodology is a set of preferred practices for conducting analysis, design, and construction. The term 'structured' is often applied to currently popular practices.

Recognition of tendency to err. This viewpoint recognizes that mistakes of both omission and commission are sure to occur at all stages of system building. One of the main tools used is inspections, performed at every stage of the project and at every level of system decomposition. The goal is to catch the errors as early as possible. The other part of this viewpoint is the realization that iteration will be required, to re-do parts of the system as the mistakes are uncovered. So the practices of design and construction that are preferred are those that allow for fairly easy correction.

More efficient use of resources. The methodologies recognize that individual differences exist among staff members. So specialization is often advocated, in one form or another, to

take advantage of these differences. Chief programmer teams are one form of specialization; Weinberg's unstructured teams are another. Then, too, mechanized aids are being advocated to help in the analysis, design, and construction stages. These include automated documentation tools, analysis tools, and interactive programming facilities.

As an example, in our November 1977 report we discussed the set of improved programming technolgies (IPTs) that IBM has developed and is marketing; see Reference 5. This is an assemblage of stand-alone methods. Users may choose almost any combination of the methods.

IBM's disciplined approach includes top-down design, HIPO charts, pseudo code, structured programming, and top-down programming. The recognition of the tendency to err is handled by structured walk-throughs. And the more efficient use of resources is accomplished by chief programmer teams, development support library, and an interactive debugging and testing facility using TSO. The development support library stores successive versions of programs and test data.

It will be helpful, we think, to keep these IPTs in mind when reading the description of the other methods we will discuss shortly. There are some significant differences between the IPTs and these other methods.

We will discuss Chase Manhattan Bank's use of data flow diagrams and PSL/PSA, Armco Inc.'s use of PRIDE, and Placoplatre's use of Warnier's LCS.

Chase Manhattan Bank

Chase Manhattan Bank, with headquarters in New York City, is the third largest U.S. bank, according to *Fortune* magazine. It has over $53 billion in assets and more than 30,000 employees.

In late 1977, Chase reorganized its headquarters operations functions into groups or divisions, structured along banking product lines. Together with this reorganization, the data processing responsibility was decentralized into the groups or divisions—including the application system development function. We talked to a data processing manager in the processing utility group, which handles check processing and payment systems.

In 1975, when he was a programmer, this manager was exposed to structured programming and, some months later, to structured program design and other development methodologies. He began using some of these methodologies in his own programming. Partially because of this, he was put in charge of the bank's programmer productivity program. He started to apply these techniques more widely. With the reorganization, he has adopted a set of methodologies as 'standard' within his group.

The methodologies used within his development staff include functional decomposition, data flow analysis, requirements specification language, structured design, pseudo code, development support library, interactive program development, structured coding, prototyping, inspections, and a data dictionary.

The more powerful of these methodologies were learned and applied almost in a bottom-up manner, he told us. That is, the first one learned was structured coding. Then came structured design, followed by structured analysis (data flow diagrams). Finally, he and his people learned how to perform functional decomposition. In practice, these techniques are used in the other order, starting with functional decomposition.

With functional decompostion, what one is really decomposing is the data, we were told. After the data is defined at each level, the processes for operating on the data are defined.

We visited Chase to learn about their use of two of the methodologies—data flow analysis and a problem statement language and analyzer.

Data flow analysis

In 1976, Chris Gane, then a vice president at Yourdon, Inc., presented a seminar at Chase on data flow analysis, which triggered the bank's use of this methodology. It has been derived from ideas originally developed by Larry Constantine, which we will discuss briefly later in this report and in more detail next month. Gane and Trish Sarson have since written a book about data flow analysis (Reference 1).

This is a very readable book upon which the following discussion is based.

Data flow analysis is a graphical method that uses four symbols—one for a source or destination of data, another for a flow of data, a third for a process which transforms flows of data, and one for a store of data. The mechanisms employed (computer, manual procedures, etc.) and the time or volume of data flow are not diagrammed.

Data flow analysis starts simply, by drawing the main data flows for the application under consideration. For instance, consider an order-shipping-billing application. The initial diagram might just show the flow of customer order data. Orders are received, credit is checked by reference to a customer file, and inventory is checked by reference to an inventory file. The diagram is then expanded to include the inventory replenishment function. As inventory falls below a specified point, a purchase order must be issued, which goes both to a supplier and to the open order file. Then the accounts payable and accounts receivable functions are added to the diagram. Eventually, a rather complex, convoluted data flow diagram of the whole application results; this is sometimes called a 'bubble chart.'

The experience at Chase indicates that such a high level diagram is very useful for describing an application system to user department management. The diagram is quite understandable by these people, we were told. No computer jargon is used on the diagrams. The managers can grasp what is, or is proposed to be, done and can indicate where they want changes to be made.

The next step is to start decomposing this overall diagram into departmental data flows, identifying organizational units that generate or receive data. The people at Chase who are using this method apply Constantine's ideas of cohesion and coupling for forming logical groupings of data flows. The next step is to identify blocks of data associated with jobs, tasks, or work stations. The blocks are then decomposed into logical records and data *structure* diagrams. The data structure of a report, for instance, might show a header, a body, and a summary, each with one or more levels of sub-division.

In a presentation given at the GUIDE 46 meeting in May of last year, three members of the processing utility group at Chase made the point that their functional decomposition deals mainly with data structures rather than with processes. Most analytic methods emphasize the hierarchical decomposition of processes. Hierarchy is not a meaningful feature of their decomposition, however. Instead, at Chase they use data flow diagrams, data structure diagrams, procedures written in structured English, and well defined attributes to describe the network of primitive functions which constitute the functional specifications of the system.

The next step is to start using pseudo code to describe the operations involved in each step, which operate on groups of data elements. The final decomposition step is to express the operations that are performed on elementary data items, in a manner most suited to the logic—pseudo code, decision tables, or decision trees.

As mentioned, the highest level data flow diagram is complex and convoluted. However, by the time it has been decomposed to the lowest level, the diagrams are fairly straight forward, they told us.

To support this decomposition process, Chase uses a problem statement language (PSL) and a problem statement analyzer (PSA).

PSL/PSA

These two tools were developed under the ISDOS project at the University of Michigan, led by Dr. Daniel Teichroew. Our first discussion of these technologies was in our November 1971 report. Since that time, PSL and PSA have graduated from the research and development stage and are now in everyday use at a number of large organizations such as Chase. For more information on these technologies, see Reference 6.

PSL is a language for describing systems; it is not a procedural programming language. It can name up to 20 types of objects (such as INPUT, OUTPUT, and PROCESS), can describe properties of those objects (such as synonyms and key words), and can describe relationships between objects.

PSA is a software package that is used to check the data as it is entered, store it, analyze

it, and produce up to 20 different reports. PSA is somewhat like a data dictionary in that it stores data definitions in human-readable format. But it goes much further; it also stores system and process definitions, and performs a variety of types of checks on these definitions. PSA runs on a number of large systems, including IBM, Univac, CDC, Honeywell, Amdahl, DEC, Siemens, and Fujitsu.

Chase begins using PSL when they decompose the departmental data flows into blocks of data. They enter the PSL statements into PSA, for validation, storage, and analysis. In subsequent decomposition steps, the lower level data flows and processes are defined in PSL. PSA can be used for drawing the data flow diagrams, making the updating of those diagrams much easier. Also, at each step, PSA produces reports that show inputs that have not been used, outputs for which no input exists, processes that have outputs but no inputs, and so on. Using these reports, the system analysts are able to spot many of the errors of omission and commission, well before programming begins.

The end result of using these methodologies, the people at Chase told us, has been better engineered software—software that is easier to maintain, easier to enhance, and with fewer aborts due to programming errors.

Armco, Inc.

Armco, Inc., with headquarters in Middletown, Ohio, is a major manufacturer of steel, metal drainage, and building products. Annual sales are in excess of $3 billion and the company employs more than 50,000 people. The corporate computer center uses twin IBM 370/168s, and a variety of other computing equipment is used at the divisions and subsidiary companies.

The metal products division in Middletown, with sales of about $350 million and about 5,000 employees, obtained the PRIDE system development methodology in 1976, to aid them with their system development. They selected PRIDE because it covers project management, data management, documentation, administrative procedures, and user involvement. Since acquiring PRIDE, the division has used it on all new development efforts. The division has a development staff of 15 people, and uses outside consultants as needed.

The division has used PRIDE in varying degrees on a wide variety of projects. For example, in connection with *purchased software*, they have used it to design the installation plan and to create all descriptive documentation. With this approach, they have obtained a better installation of purchased software than they had ever had before. Also, they have used PRIDE for *small projects*, such as enhancements to existing systems as well as for small, standalone systems. Perhaps the main use of PRIDE, though, is in connection with *major projects*, which can require up to three years to implement. These very large, complex, or very long projects remain a difficult problem to resolve. But PRIDE does provide them with a workable methodology (called 'chronological decomposition') for structuring the design of the new system.

PRIDE is a set of 'standard' practices for developing computer-based application systems, based on this structured method. It includes both the design and construction methods to be used plus the project management methods for controlling a project. PRIDE uses nine well-defined phases that relate to the structured components of the system. The first three cover the system study, system design, and subsystem design, the next three cover the design of the manual system, program design, and program test, and the final three cover system testing, system operation, and system audit. There are numerous check points and user sign-off points in these nine phases.

As the people at Armco say, a methodology does not *cause* successful system development projects—people do! But PRIDE does increase the probability of success almost regardless of who uses it. As compared with the methods they used previously, it provides more comprehensive documentation at earlier stages of a project, it provides better communication among the involved parties, it encourages greater user involvement at all levels and all stages of a project, and it leads to better definition and planning of requirements and resources.

They cited one interesting example of the use of PRIDE—their first major use of it, as a

matter of fact. The purpose of the project was to develop a manufacturing order entry system for a steel plant. The project was initiated using PRIDE, for determining the user needs and for designing the overall system and its sub-systems. In the course of the project, a minor crisis occurred—one that is not uncommon in data processing: a number of people associated with the project moved on to other responsibilities. Rather than hire and train new people, management decided to utilize contract programmers for that phase of the project.

So proposals were obtained from several software and consulting firms. Some of these firms proposed the use of their senior people (at consequent high daily rates) but Armco felt that the specifications were well-enough documented that the programming could be done by less experienced people. And, in fact, one firm agreed with this; it bid the services of two experienced programmers to work on Armco premises. The contract was given to that firm.

The results of that project still impress the people at Armco. In about three months, these two people wrote some 120 modules, 75 of which were in PL/1, that constituted the manufacturing order entry system. It was an outstanding example of productivity, they still feel. Looking back at this project, the systems department can see how they could have made the specifications for the system even more complete than they were. But even with this first major use of PRIDE, the specifications were quite satisfactory and provided the basis on which the programming was done.

PRIDE and Logik

PRIDE and PRIDE-Logik were developed and are marketed by M. Bryce and Associates, Inc., of Cincinnati, Ohio. We discussed PRIDE in our December 1974 issue, and Logik in our January 1978 issue.

As discussed above, PRIDE is the underlying project management and system development methodology, for developing computer-based systems. It includes a comprehensive 'data management' function that initially was handled manually during the development cycle.

Logik has mechanized that data management function, as the term is used by Bryce. That is, all system definitions, process definitions, and data definitions that fall within the scope of the system being built are captured, stored, and analyzed. As these definitions are developed during the various stages of PRIDE, they are entered into the Logik dictionary and analyses are performed on them. Processes that have no inputs or that produce no outputs are identified. Data fields in outputs that have not been inputted or computed are flagged. And so on.

Milt Bryce, the originator of PRIDE, believes that these two methodologies cover the engineering and building of complete application systems, including the management of the projects. Further, they also produce the documentation of the results as byproducts.

M. Bryce and Associates, Inc. are now marketing the combination of PRIDE and Logik under the name 'Automated System Design Methodology' (ASDM). This combination has been mechanized (in COBOL) to run on many existing maxi computers and is being made available on mini computers (any computer with ANS COBOL, 128K, and relative input-output, they tell us).

In using ASDM, analysts would retrieve from the Logik dictionary as many definitions (of both system and data components) as exist there that apply to the system under study. Such information helps the analysts determine user requirements.

The design phase use of ASDM begins with identifying the 'regular' outputs that are desired from the new overall system. The designer enters the definition of these outputs into ASDM, including any known procedures (such as 'gross pay = rate x hours'). The designer also enters the definition of inputs and, as required, of the files. Then the designer investigates a series of questions. Is the output data supported by data in the files? If not, is it supported by input? Is each output produced in time cycles or on request? And so on.

Then comes 'chronological decomposition.' The overall system is divided into sub-systems, in terms of time cycles, offsets within time cycles, response time requirements, etc. Those outputs which must be produced interactively are identified, as well as those which must be produced daily, weekly, monthly, and so on.

When the timing of the outputs has been established, the designers then determine when the inputs must be made available in order to support the outputs.

When this time cycle design has been completed, then the designers use ASDM to check all data flows in all sub-systems. Gradually, all inconsistencies and missing elements should be identified and corrected.

A point to make here is that ASDM incorporates the three elements we listed at the beginning of this report. There is a disciplined approach using a set of preferred practices. There is a recognition of the tendency to err, by providing inspection points in the several stages and by the checking that is done by Logik. And the methodology seeks to make more efficient use of resouces by mechanizing many of the routine functions that analysts and designers must perform.

For more information on ASDM, see Reference 7.

Société Placoplatre

Société Placoplatre, headquartered in Rueil, France, near Paris, is a manufacturer of plasterboard products for the construction industry. The company has four geographically dispersed plants in France plus seven district sales offices, and employs some 1200 people. For its data processing, Placoplatre uses a Honeywell H2050, plus a Datanet 2000 for serving terminals at the 11 remote sites. The data processing staff totals 25, of which seven are in system development.

In 1971, Placoplatre became one of the first users of Warnier's logic for constructing programs (LCP), developed by Jean-Dominique Warnier of CII-Honeywell Bull in France. We described Placoplatre's experiences with LCP in our December 1974 issue. They continue to use LCP and, for example, developed a teleprocessing monitor with it that worked so well they sold copies to other users.

In 1974, Placoplatre began using Warnier's logic for constructing systems (LCS); again, they were one of the first users of this technique. As with any new method, the company had some problems with it at first, but they stayed with it. Placoplatre is now using LCS (together with LCP) for *all* new application system development.

What did Placoplatre seek when it first considered LCS? They sought a methodology for developing application systems that would support flexible, reliable system development and that would meet the needs of their company. They wanted to be able to identify all of the data and data relationships that the company could foresee for its future data processing, so that they would not continue to be surprised by 'new' data requirements. They felt—and they still feel—that LCS meets these needs.

Let us consider briefly what LCS is. It is a fairly complex system and we cannot do justice to it in a brief write up. For more information, see Reference 8.

LCS

At the heart of LCS is Warnier's philosophy: "Do not try to copy your present system," he says. "This just leads to undesirable redundancy of data and programs. Instead, concentrate on what data the users really need."

LCS is primarily concerned with finding all elementary data items that are needed by the organization. These data items are then grouped into logical files and eventually into physical files.

To help flush out all of the data that users will need, LCS imposes a good deal of structure on the analysis process. Here are the main elements of Warnier's structure.

An organizational entity—which can be a section, a department, or a whole organization—has relations with 'customers' and 'suppliers.' Further, these customers and suppliers can be both internal (to the organization) and external. For example, an employee might request that a book he needs for his work be purchased by the company; the company thus becomes the employee's (internal) supplier. The company, in turn, orders the book from a book store; the company is the customer and the book store is the (external) supplier.

The customer is the person or organizational entity that initiates a transaction.

A base defines the relationship between a customer (or a supplier) and a 'product.' A product, in turn, can include services, such as the

services that an employee performs for the employer. The data incorporated in a base includes (a) general data about that type of relationship, such as pertaining to all employees, (b) specific customer or supplier data, such as name and address, (c) data about the specific product or service, such as the type of employment agreement for that employee, and (d) transaction data such as pertaining to the specific pay period.

The organization's bases are the aggregate of these individual bases. This aggregate is organized in terms of customer-internal, customer-external, supplier-internal, and supplier-external.

For practicality, it is not necessary to identify the complete set of bases at the outset. Instead, identify the ones of interest at the moment and lump the rest under 'other,' to be sub-divided later.

A transaction is the record of an action. Further, Warnier sees an action (whether involving internal or external entities) as consisting of four parts (although all four need not be present in every case). These are: (a) an order from the customer to the supplier for a specific product, (b) the delivery of that order, (c) an invoice for the delivery of the order, and (d) a payment for the delivery.

The needed data consists of two main types, each of which sub-divides into two parts. *Primary data* must be inputted and stored. In turn, it is either used for output or for computations. *Computed data*, as its name implies, is derived from primary data. In turn, it is either stored or is re-computed whenever needed.

"If you will look at the organization's data from this point of view—customers, suppliers, relationships, transactions, primary data, and so on," says Warnier, "you will be better able to identify what data users really need. You will minimize redundancy in your data files and you will be less likely to discover important gaps in your data, as new applications are developed."

Using the methodology

As Placoplatre has used LCS, the first step has been to look at the overall organization and identify the bases—the customer/product and supplier/product relationships. The company feels that it has identified all such relationships.

Then, for the application area under consideration, they start collecting the 'output' data of the present system. They identify the primary data and the computed data, along with the formulas of computation.

Next, they sub-divide the bases into logical base files—logical groupings of customer/product and supplier/product relationships. Placoplatre has identified some 110 logical base files.

The next step is to define the physical files, looking for data items that serve more than one function. Placoplatre has, so far, defined 22 physical files.

Then they identify 'logical programs,' considering the sources and destinations of the data. Following this, they identify the 'physical programs.' At this point, the use of LCP can be started, for constructing the physical programs.

LCS includes a variety of cross-reference reports, for analyzing these definitions for consistency, gaps, and redundancies. These reports include primary data versus outputs, primary data versus logical base files, and so on. Placoplatre has found that the use of these LCS reports have greatly reduced the 'errors' that used to occur in the data definitions. It is now known just where each data item is used, for instance. Maintenance is easier because the programmer knows just where the change is to be made.

With LCS, there are no big data files; rather, there are many small ones. Some files are only tables. New applications generally can make use of one or more existing files because the data is so fundamental. Redundancy has been reduced and new applications integrate quite easily with existing LCS applications. (It will take Placoplatre some time, however, to integrate their pre-LCS applications with the LCS data structures.)

LCS allows all development people to know the whole LCS application area. The impact of a new system can be seen, as can a change to an existing system. Also, programming cannot begin until the computer operations people can see that a new application integrates well with the existing applications. The people at

Placoplatre are pleased with the benefits that they are getting from LCS.

Two other sources

There are two other sources of relevant information that we would like to mention.

Structured analysis and system specification, by Tom De Marco (Reference 2), is a quite comprehensive discussion of the system development methodology offered by Yourdon, Inc. Much of the original thinking of this methodology is credited to Larry Constantine, but Edward Yourdon and his colleagues (including Tom De Marco) have done a lot to extend and refine the concepts for field use.

Constantine's ideas originally were applied mainly to the design of modular programs. But the people at Yourdon, Inc. have extended them into the areas of system analysis and system design.

As P. J. Plauger says in the foreward of this book, "What I like most about this book is how well it teaches the construction and evaluation of Data Flow Diagrams. Larry Constantine encouraged the use of such graphic aids over a dozen years ago as a way to analyze a restricted class of systems known as transform centers. Transform Analysis, however, seemed to get lost among the myriad innovations of Structured Design. It is only with 20-20 hindsight that we can see that the transform center is but the simplest non-trivial Data Flow Diagram, and that an understanding of data flow is vital to the success of *any* system design."

The heart of the book might be considered to be the structured analysis of system requirements through the development of data flow diagrams. But the important role of the data dictionary is also emphasized. And eventually the system developers get to the processes, which are handled (at this level) by structured English, decision tables, or decision trees.

The book is very readable and abounds with examples.

User experiences with new software methods, a technical session at the 1978 National Computer Conference (Reference 3), was filled with practical user experiences with some of the methods we have discussed this month and last month.

The conference proceedings contain rather brief position papers by the panel members. But, in addition, a cassette recording of the actual session can be obtained; it includes information not found in the position papers.

We would encourage readers who are interested in the methodologies that we are discussing in this series of reports to obtain the several source materials listed in the references.

An emerging pattern

As the above discussion has indicated, it appears that a number of common characteristics are emerging for a system development methodology.

One might ask, of course, whether it is reasonable to expect *one* methodology to win out. Will one methodology adequately serve both large and small organizations, for systems that range from simple to complex, and for applications that range from business to scientific to air traffic control? We suspect that, while different methodologies might be used in these several environments, they will have many points in common. It is those points in common that we are talking about.

Following are the common characteristics that are emerging, in our opinion.

A disciplined approach

As mentioned, a disciplined approach is one of the common characteristics of the methodologies. It, in turn, has several attributes.

An integrated methodology. It seems to us that, for any particular organization, the development staff should be concerned with only one methodology, consistently applied from project initiation to operation. We do not see a series of techniques, which the development staff may or may not use as they choose, as the right answer. True, the one methodology may consist of a number of elements for handling analysis, design, construction, and testing. But these should all be part of the whole, not stand-alone techniques.

Of the methodologies discussed in this report, probably ASDM comes closest to meeting this goal.

Handle complexity by successive decomposition, starting with the analysis of requirements.

This same concept is used for design and construction, and should lead naturally from analysis to design to construction.

Of the methods discussed, ASDM, LCS/LCP, and data flow diagrams seem to come closest to meeting this ideal. Some might argue that IBM's HIPO charts also should be included here. While they do provide for successive decomposition, users have reported to us that HIPO does not lead naturally to program code.

More emphasis on the data. Conventional system development tends to look first at the processes (the programs), and eventually gets around to considering the data definitions. It seems to us that this sequence should be reversed. Start with the data definitions; as they begin to crystallize, consider the processes. Consider the processes as 'black boxes' at first, and decide (as Warnier suggests) what outputs they must provide to meet user needs. And as noted at Chase Manhattan Bank, successive decomposition is really a matter of decomposing the data.

Constrain the effect of errors. Another characteristic of the ideal methodology is that it tends to constrain the effects of errors on the part of the development staff. Two methodologies stand out in this regard.

One is Constantine's ideas on modular design, incorporated in IBM's top-down methods and also discussed in References 1 and 2; we will have more to say about these methods next month. Constantine seeks minimum coupling between modules so that each module is as free standing as possible. He also seeks maximum functional cohesion within a module, where the elements of the module perform only one function. With such a design, the impact of errors is constrained.

The other method is Warnier's LCS, which modularizes data into many small files. These 'logical base files' are logically developed, not defined haphazardly. So an error in data definition is likely to be constrained to one such file.

These four characteristics—an integrated methodology, the ability to handle complexity by successive decomposition, giving more emphasis to the data, and constraining the impact of errors by modularization—should apply to a set of preferred practices.

Set of preferred practices covering analysis, design, construction, and testing that all development staff members use.

Here is the crux of the matter, it seems to us. The overall methodology, consisting of this set of preferred practices, *must* be so good that data processing management can mandate its use. The development staff generally will resist 'standards' being imposed, because such standards probably mean that the people must change their thinking habits and job methods. If those standards can be shown to be deficient by the staff members, management will have a hard time demanding their use. So the methodology must be good enough to stand up under such attacks.

The methodology must provide a set of usable, effective, appropriate practices that are part of an integrated whole and that are used for analysis, design, construction, and testing. It seems to us that the state of the art probably has not reached this point yet, but it is getting close.

Recognition of tendency to err

The disciplined approach, while it may seek to help the system designers and builders 'do the job right,' must recognize that it must also help them to 'do the job over' when errors are detected. A fundamental proposition is that errors will be injected into the work products at all stages of a project. So the methodology must help detect those errors and then facilitate the changes to correct for the errors.

Inspections. As we discussed last February, a good amount of research is going on in 'proof of correctness' methods for insuring the correctness of programs as they are being developed. In a sense, inspections occur continually as a program is being designed and written, so that inspections by someone other than the author may not be needed.

But the proof of correctness method is not yet a part the technology that most system builders can use. For them, inspections of their

work products by qualified people are essential. The inspection procedure should be inherent in the system building methodology that is used.

Of the methodologies that we have discussed, in both this report and the one last month, we were most impressed by the inspection approach used with SADT. In this case, each diagram document is studied by a person with the most knowledge of the area under consideration. This 'commentor' notes questions, errors, etc. right on the diagram. The author must respond to all such notations, either making the changes or indicating why they are not being made. The inspection process is clearly 'built in' for SADT.

The other methodologies provide for inspections, but not to the same degree, in our opinion. Also, in the cases of ASDM, PSA, and LCS, the mechanized processing results in diagnostic (inspection) messages. The IPTs include structured walk-throughs which are inspections, but their use is not 'built in' or mandatory.

Iterations. The inspection procedures should flush out errors at all stages of the project—analysis, design, construction, and test. Once the errors have been detected, the methodology should make the correction of the errors as easy as possible. (We are including changes in design, due to any of a variety of reasons, with the correction of errors.)

The graphical methods—SADT, IA, and data flow diagrams—can be corrected fairly easily, we gather from talking to users. It is no big task to redraw a diagram to incorporate changes. We understand that the wordiness of HIPO diagrams make the task more cumbersome, however.

The mechanized documentation methods—such as ASDM and PSA—provide for the easy updating of the documents. Changes are entered into the computer and the new documentation can then be printed out.

None of the methodologies have included, as an integral part of their approaches, methods for easily changing programs. Nothing like the incremental development approach of Basili and Turner (to be described below) has been included, to our knowledge. This is one of the major missing elements of the methodologies, we think.

Also, none of them have incorporated on-line program development as an integral part of the methodology. IBM's TSO might be considered part of the IPTs, but its use is optional. On the same basis, any on-line programming system could be used with any of the methodologies. In addition to on-line services, though, this approach also requires a variety of software development tools.

For changes in data definitions, PRIDE-Logik and PSA provide data dictionary functions. However, we understand that neither provides a direct interface with a DBMS, for feeding the data definitions and changes to those definitions to the DBMS, for productive use.

Efficient use of resources

As mentioned, the more efficient use of development resources can come about by staff specialization and by the use of mechanized aids.

Staff specialization. Only IBM's IPTs emphasize this point, via the chief programmer team concept. However, this concept has received its share of criticism, because the demands made on the chief programmer are such that many installations may have no one that qualifies as a chief programmer.

Perhaps more could be done along the lines of Gerald Weinberg's unstructured teams, where the team member with the greatest capability for the current phase becomes the team leader for that phase.

Not all development staff people can cope with top-down development; it demands a new way of thinking, as contrasted with the more conventional bottom-up approaches. If a methodology of the type we are discussing is adopted—one where top-down development is fundamental—then these staff members will have to transfer to other jobs.

Mechanized aids. We see five main types of mechanized aids being part of the overall methodology.

One is a development 'data' dictionary. Actually, the dictionary should be able to store much more than just data definitions. It should store process definitions, for both manual and

computer processes. It should store system definitions, including data volumes, timing requirements, and so on. We discussed the development dictionary function in our January 1978 issue.

Another is an analysis program, to be used with the dictionary. It can be used to check for consistency, for instance, to make sure that all inputs are used, that all outputs are supported either by inputs or computations, that all processes have both inputs and outputs, and so on.

ASDM and PSA both provide development dictionary and analysis capabilities.

A third aid is a program and test data library. It should be able to store both the current and the immediate past version of each program under development; further, the printouts should provide an audit trail of all changes to all programs. Versions of the test data and test results also should be stored. IBM's development support library provides these functions.

A fourth aid is an interactive programming facility, mentioned above.

Finally, we think that an incremental development facility, designed to make it easier to modify and enhance programs and data definitions, should be a basic component of the methodologies.

The ideas of Basili and Turner (Reference 4), which we discussed in our February and March issues last year, are of interest here. Their approach starts with the construction of a simple skeletal subset of the system under development. It includes a sampling of the key aspects of the system and ones that will deliver useful outputs to the users. This skeletal solution is only an initial guess at the structure of the final solution.

Build this skeletal solution and give the outputs to the users, they say. Find out how the design must be changed, and then change it. Since only a part of the total system has been constructed, the complexity should not be too great and the changes should not pose much of a problem.

Then add more aspects of the overall system to the skeletal solution and repeat the process. As the system evolves, analyze it for structure, modularity, reliability, and so on. As the need for changes becomes apparent, make them. Most of the significant changes will occur early in the process, say the authors, when it is not too difficult to make them.

The system designers can, and should, take steps to make the initial design reasonably good. But this approach recognizes that user requirements errors and system design errors are sure to creep in. When the errors are detected, work has to be done over. Mechanized aids are needed to make this process as efficient as possible.

System development 'workbenches'

As we discussed in our November 1978 report, we expect the new computer systems of the early 1980s to emphasize system development modules. There are two main reasons for this belief. For one thing, the state of the art does, in fact, support many mechanized aids for system development. Secondly, we see these modules as strong sales points for the mainframe manufacturers, as they seek to counter the plug-compatible competition.

The term 'workbench' is being applied to these modules, in that they would provide software tools for use by software developers. The first use of the term, to our knowledge, was for the 'programmer's workbench,' developed at Bell Laboratories. This workbench was created to run on larger DEC PDP-11 systems, under Bell Labs' UNIX operating system. It has been used by Bell Labs for developing application programs for IBM 370, Univac 1108, and Xerox Sigma 5 computers.

As conceived by Bell Labs, the workbench concept applies to the programming function in its broadest sense—that is, the complete development and maintenance cycle. The workbench tools apply to the generation of system specifications (based on system requirements), and cover program development, test, monitoring, evaluation, maintenance, conversion of data files, and so on. It is our understanding that the tools included to date have applied mainly to the conventional programming function.

We suspect that what will be marketed, however, will be several workbenches—an analyst's workbench, a designer's workbench, a

data administrator's workbench, and a programmer's workbench. At first, these probably will be available only on maxi and mini computers. For instance, users can now purchase the Bell Labs' programmer's workbench by buying the UNIX operating system from companies that Bell Labs has licensed to sell and maintain it. In this environment, the workbench services would be provided on a time sharing basis, and each programmer might have his/her own terminal.

Not too far in the future, however, we foresee workbenches being made available on micro computers, on a hardware/software package basis. Further, the price of such packages might well be low enough to be economically attractive to a large number of companies.

Last month, we discussed some ideas related to the analyst's workbench. The analyst would draw system diagrams on a graphics terminal, and the workbench would make easier the revising of diagrams, checking for completeness, numbering them, retaining prior generations of diagrams, etc.

How about a designer's workbench? As we see it, this workbench should accept the results of the analyst's work—the graphic diagrams, data definitions, etc. It should help the designer perform at least the routine aspects of the design function: help create system diagrams and revise them as necessary, check inputs and outputs for consistency and completeness, and such. It *might* also help in the system decomposition function, although this is still largely a judgmental process. And it should have a library of standard application system software components from which the designer can select.

We also see a programmer's workbench, which we will discuss briefly next month, and a data administrator's workbench, which we will describe month after next. These several workbenches, among them, should provide the mechanized aids that we discussed above in this report.

Yes, the concepts of system development workbenches are well along toward realization. Many of the tools are in use today. But it would be very desirable if the workbenches could be based on a 'common' approach to system development.

The search for fundamentals

As we see it, none of today's methodologies contain all of the elements that we believe will become part of the common approach to system development. But it also appears that today's methodologies, among them, contain almost all of those elements. So it may not be long before a 'complete' methodology appears.

This common methodology, when it evolves, probably will incorporate basic ideas that have been contributed by a number of the original thinkers in our field.

For instance, Edsger Dijkstra has advocated the concept of successive decompostion ('levels of abstraction') for handling complexity. Further, he has urged that the use of GO TO be (essentially) eliminated in order to reduce program complexity. Harlan Mills has proposed the use of three basic control structures—sequence, iteration, and choice—to reduce program complexity. Larry Constantine has developed the ideas of coupling and cohesion for module design. Jean-Dominique Warnier has pointed out that data is the 'driving force' that should shape both system and program design. Michael Jackson concurs that the program structure should be based on the data structure, and seeks to find the data structure that fits a given problem most naturally.

It is not yet clear just what are the true fundamentals of system and program design and construction. Each of the above doctrines has it adherents. Each has had its share of successes in field use. Each is certainly non-trivial to learn to use. Each requires a change in 'conventional' thinking habits. And so far, each has its own terminology, rules of use, and viewpoint. As yet, they are not too compatible with each other.

Hopefully, as each is exposed to more field use and as researchers begin to make comparative studies of them, certain fundamental principles will begin to emerge from these doctrines.

What is the best next step?

In the meantime, what is the best approach to use, for producing better quality application software?

The group at Chase Manhattan Bank that we interviewed made a point worth repeating. Their 'learning sequence' was, first, structured coding, then structured design, then structured analysis, and finally successive decomposition. Their 'using sequence' is the opposite. It starts with successive decomposition and continues through analysis and design to coding.

It might be wise to think of installing the methodology in this manner. Start with the coding aspect and then work up toward successive decomposition. This is a controversial point, of course. Some methodologies, such as SADT (discussed last month), require starting with successive decomposition.

But which of the doctrines should one choose? Eventually (hopefully) there will be a common doctrine, or perhaps a family of doctrines. But at present, as mentioned, they are not compatible. You will have to assemble your own package. And that is going to take a bit of study.

We suggest that you study the methodologies and tools that we have discussed in these reports—SADT, IA, PSL/PSA, LCS/LCP, ASDM, and data flow diagrams, plus the IBM IPTs we discussed in our November 1977 report.

Then study the concepts of Dijkstra, Mills, Constantine, Warnier, and Jackson, plus others. Next month, we will continue our discussion of better development methods by describing user experiences with some program development methodologies. We suggest that you not make a selection of a program design and construction methodology too quickly. Study these different approaches and talk to some users of each.

Then decide what methods best fit your needs and your resources.

We wish we had a shortcut to suggest. A few years from now, maybe there will be a good, common approach for developing better quality software. It seems imminent, most of the technology has been developed, but it is not yet here. Until then, you will have to assemble your own approach, to fit your own situation.

REFERENCES

1. Gane, C. and T. Sarson, *Structured systems analysis: tools and techniques*, Improved System Technologies, Inc. (888 Seventh Avenue, New York, N.Y. 10019), 1977, 373 pages; price $33 ($30 if prepaid).
2. De Marco, T., *Structured analysis and system specification*, Yourdon Press (1133 Avenue of the Americas, New York, N.Y. 10036), 1978, 352 pages; price $25.
3. *Proceedings of 1978 National Computer Conference*, AFIPS Press (210 Summit Avenue, Montvale, N.J. 07645), price $60. The position papers referred to are on pages 629 to 639. In addition, a cassette recording of the session "1978 NCC, Th98, User experience with new software methods" can be obtained from On-the-Spot Duplicators, Inc., 7309 Fort Hunt Road, Alexandria, VA 22307; price $5.50 plus billing and shipping charges.
4. Basili, V.R. and A. J. Turner, "Iterative enhancement: a practical technique for software development", *Software Engineering* (IEEE Computer Society, 5855 Naples Plaza, Suite 301, Long Beach, Calif. 90803); December 1975, p. 390-396; price $10.
5. For more information about IBM's IPTs, contact your local IBM office. They are described in a number of general information manuals. See GE 19-5086 for an overview (price $2.70).
6. For more information about PSL/PSA, write ISDOS Project, Department of Industrial Engineering, 231 West Engineering Building, University of Michigan, Ann Arbor, Mich. 48109.
7. For more information about PRIDE and ASDM, write M. Bryce and Associates, Inc., 1248 Springfield Pike, Cincinnati, Ohio 45215.
8. For more information on J.-D. Warnier's LCP and LCS: in U.S., contact Van Nostrand Reinhold (450 West 33rd Street, New York, N.Y. 10001); as we go to press, the book on LCS is still being negotiated; in Europe, contact Martinius Nijhoff, Social Sciences Division, Pieterskerkhof 38, Leiden, The Netherlands.

SUBJECTS COVERED BY EDP ANALYZER IN PRIOR YEARS

1976 (Volume 14)

Number

1. Planning for Multi-national Data Processing
2. Staff Training on the Multi-national Scene
3. Professionalism: Coming or Not?
4. Integrity and Security of Personal Data
5. APL and Decision Support Systems
6. Distributed Data Systems
7. Network Structures for Distributed Systems
8. Bringing Women into Computing Management
9. Project Management Systems
10. Distributed Systems and the End User
11. Recovery in Data Base Systems
12. Toward the Better Management of Data

1977 (Volume 15)

Number

1. The Arrival of Common Systems
2. Word Processing: Part 1
3. Word Processing: Part 2
4. Computer Message Systems
5. Computer Services for Small Sites
6. The Importance of EDP Audit and Control
7. Getting the Requirements Right
8. Managing Staff Retention and Turnover
9. Making Use of Remote Computing Services
10. The Impact of Corporate EFT
11. Using Some New Programming Techniques
12. Progress in Project Management

1978 (Volume 16)

Number

1. Installing a Data Dictionary
2. Progress in Software Engineering: Part 1
3. Progress in Software Engineering: Part 2
4. The Debate on Trans-border Data Flows
5. Planning for DBMS Conversions
6. "Personal" Computers in Business
7. Planning to Use Public Packet Networks
8. The Challenges of Distributed Systems
9. The Automated Office: Part 1
10. The Automated Office: Part 2
11. Get Ready for Major Changes
12. Data Encryption: Is It for You?

1979 (Volume 17)

Number

1. The Analysis of User Needs
2. The Production of Better Software

(List of subjects prior to 1976 sent upon request)

PRICE SCHEDULE (all prices in U.S. dollars)

	U.S., Canada, Mexico (surface delivery)	Other countries (via air mail)
Subscriptions (see notes 1,2,4,5)		
1 year	$48	$60
2 years	88	112
3 years	120	156
Back issues (see notes 1,2,3)		
First copy	$6	$7
Additional copies	5	6
Binders, each (see notes 2,5,6)	$6.25	$9.75
(in California	6.63, including tax)	

NOTES

1. Reduced prices are in effect for multiple copy subscriptions and for larger quantities of a back issue. Write for details.
2. Subscription agency orders are limited to single copy subscriptions for one-, two-, and three-years only.
3. Because of the continuing demand for back issues, all previous reports are available. All back issues, at above prices, are sent air mail.
4. Optional air mail delivery is available for Canada and Mexico.
5. We strongly recommend AIR MAIL delivery to "other countries" of the world, and have included the added cost in these prices.
6. The attractive binders, for holding 12 issues of EDP ANALYZER, require no punching or special equipment.

Send your order and check to:
EDP ANALYZER
Subscription Office
925 Anza Avenue
Vista, California 92083
Phone: (714) 724-3233

Send editorial correspondence to:
EDP ANALYZER
Editorial Office
925 Anza Avenue
Vista, California 92083
Phone: (714) 724-5900

Name_____

Company _____

Address _____

City, State, ZIP Code_____