EDP ANALYZER

MARCH, 1979
VOL. 17, NO. 3

PROGRAM DESIGN TECHNIQUES

For the past two months we have been discussing some relatively new software development techniques, some for improving the analysis of user needs and others for improving the design of software systems. This month we move along the software development life cycle a bit more by discussing some tools for improving the design of computer programs. We will concentrate on three of the more popular methods, ones conceived by Jean-Dominique Warnier, Larry Constantine, and Michael Jackson. And we will speculate on the future mechanization of the program design process.

The Office of the Administrator for the Courts for the State of Washington, in Olympia, Washington, was established in 1957 to study the operations of the state courts and to make recommendations for their improvement. The judicial system of the state consists of one supreme court at the top, three divisions of the court of appeals, 28 superior courts, 73 district courts and justices of the peace, and 238 municipal/police courts.

In 1975 the information systems division (ISD) of the Administrator for the Courts was formed. Its function is to develop judicial information systems for the entire state court system. The first system that ISD began work on was the Superior Court Management Information System (SCOMIS). This system will automate and integrate the indexing, docketing, case tracking, calendaring, accounting, jury management, warrant control, and reporting functions of the state's 28 superior courts.

The SCOMIS project began in June 1976, with development slated to continue well into the 1980s. A target date of February 15, 1977 was given for implementation of Phase 1 at one site. Phase 1 was the automation of the case-indexing sub-system for the superior courts. Up to this time, superior court filing indexes had been kept in huge books. Any changes, additions or answers to queries were made manually after a county clerk had located the pertinent name in one of a multitude of these books. Using SCOMIS the books are replaced with an on-line, interactive system accessed using CRT terminals.

Initial research for Phase 1 occurred during July 1976. During August, the six members of ISD on the SCOMIS project took a three-day course on ADABAS, the database management system to be used in the system, and a four-day course on the Warnier/Orr (W/O) structured systems method.

ISD decided to standardize on the W/O method for the design of computer systems, so all new projects are now developed using it. Jean-Dominique Warnier and his colleagues at

Honeywell Bull (now CII-Honeywell Bull) in France developed LCP (Logical Construction of Programs) and LCS (Logical Construction of Systems). We discussed LCP in the December 1974 issue, and we discussed LCS last month. Subsequently, Kenneth Orr, of Langston, Kitch and Associates, added to the LCP notation, making it applicable to designing systems, programs, data files, and manual procedures. We shall discuss the methodology later in the report.

Following the August 1976 training, the team began work on the sub-system requirements, which were completed and approved in September. From September through November the team created the external system design. Following the W/O methodology, the input and output requirements of the case indexing sub-system were developed. Some 36 screen formats were proposed, for use by the county clerks. These were presented to the clerks at the pilot site in simulated on-line sessions using a CRT. Several iterations of refining these screen formats followed. When these users were finally satisfied with the display formats, ISD felt it had the design of a system that would be *used*, not circumvented or ignored, by the clerks.

During December the internal design of the sub-system and the definition of the data elements took place. The team approach was continued and structured walk-throughs were emphasized. (A structured walk-through is a design or code review meeting in which the author of the work leads the attendees through the design or program in search of errors.) At the end of the month, seven months after the project had begun, the system design was completed. Coding began in January, with the programmers extending the W/O diagrams to the point where they could code directly from them. (For subroutines with complex logic, they went to the program statement level.) Some of the programmers used pseudo code rather than extend the diagrams totally. And all entered their code into the computer on CRT terminals using a text editing system. They also tested the 34 programs in the sub-system online.

By mid February, 1977 Phase 1 had been completed, with the case indexing sub-system installed at the test site. *Only two programming errors have been found since the installation.* And no adaptive maintenance has been requested by the users.

As is quite obvious from the above timetable, the SCOMIS case-indexing sub-system project followed a dramatically different schedule from the typical software project (at least the ones we are familiar with). They spent seven months on requirements analysis and system design, and only one and a half months on programming and system testing. Most development efforts devote much less time to design and much more time to coding.

The people at ISD told us that they never could have met their deadline without the use of both a good design methodology and organizational structure. The W/O diagrams are surprisingly easy to learn to read and clearly show the hierarchy of the system, we were told. And this immensely enhanced communication among the various people involved—managers, users, programmers and designer/analysts. During design walk-throughs, the hierarchical nature of the W/O charts made logic errors easy to spot. If, for example, a function had been placed on the wrong level in the hierarchy, this point showed up, because the chart would lose its symmetry later on. The diagrams allowed the designers to see a complete design and to see when a design was complete.

The ease of extending the system design charts for use during program design was a key to the speed in coding. The notation naturally grouped functions into sub-routines, so structured programs resulted. Without this type of a program design methodology, coding would have taken much longer, they told us.

ISD also stressed to us that their use of the life cycle development approach (including walk-throughs and teams) complimented the W/O method and contibuted equally to the success of the project.

Since February 1977, ISD has installed SCOMIS at five additional locations and has developed the docketing and case-tracking subsystems. Both of these new sub-systems were developed using the W/O methodology, both followed the same development timetable we described for the case indexing system, and

both have been virtually error-free since implementation.

All in all, ISD is very pleased with their selection of the Warnier/Orr method for designing systems, programs and data bases. In August 1978 they began using STRUCTURE(S), an automated documentation package from Langston, Kitch and Associates. It produces W/O diagrams, cross reference tables, and data element indexes from input statements. ISD sees their use of STRUCTURE(S) as a first step toward the use of more automated software design tools.

Exxon Corporation

Exxon Corporation, with headquarters in New York City, is an international integrated petroleum company. Its revenues of over $54 billion per year rank it as number two on the 1978 *Fortune* directory of the 500 largest U.S. indusrial corporations.

In the early 1970s, Exxon had become increasingly concerned about the rising cost of application system development and maintenance, particularly the manpower component of these costs. In addition, software reliability was becoming more critical to them; there was real concern about the possible effect that software errors could have on Exxon and its business operations.

So, in 1973, a project team was organized to focus on ways to improve the software development process. The main objectives were to reduce the cost of developing and maintaining systems, and at the same time to increase the quality and reliability of the software products.

The project team decided that the key to effective software development, enhancement, and future support is the *structure* of the computer programs themselves. Reliable program structures would be a first step toward greater software reliability.

After a thorough investigation, the project team recommended that the methodology developed by Michael Jackson of England be adopted for program design. This methodology, with several minor modifications, has been coupled with structured walk-throughs and top-down testing to form what Exxon calls 'program systems technology' (PST).

The Jackson program design methodology is based upon the concept that a program's control structure should reflect the structure of the data it will process. Jackson's outlook thus has points of similarity with that of Warnier.

At this point, Exxon had to address the question: "How do we teach this methodology to our programmer/analysts in Exxon installations around the world?" In 1973, there were some 1,200 people in Exxon operations worldwide working on system development; since then, this number has grown to about 1,700 people.

As a first step, Michael Jackson himself was brought in to teach the first three training courses on program design at headquarters. During this same period, the project team developed a four and one-half day workshop-type course covering all of the PST methodology. In the five years since this course was developed, it has been used to train over 1,300 programmer/analysts within Exxon, including many in other countries who do not speak English.

In order for the PST methodology to be successfully adopted at an installation, Exxon has found that several steps are necesssary. First, the corporate project team must gain the management support for the methodology at that installation. Then the initial training in PST is presented by the corporate project team. Then an on-going training program must be set up at the installation, for training the remaining programmer/analysts. And, just as important, consulting services must be provided, to help the new users during their first uses of PST.

For the on-going training, key programmer/analysts at the various sites, who were interested in teaching future PST classes, were assigned to assist with the courses. Then they became primary instructors on a part-of-their-time basis. This has resulted in a network of part-time instructors at the Exxon locations around the world.

Exxon has found that follow-on assistance is an essential part of the training process. To provided the needed assistance, Exxon has called upon these in-house course instructors to provide consulting services, supported by the corporate project team. Experience has shown, though, that the users actually need less follow-on help than they think they need. Once

they have been reassured that they are using PST correctly, their requests for help drop off quickly.

Exxon has evaluated five projects that used the PST methodology and has found important and encouraging results. These projects ranged in size from one-half workyear to over 25 workyears of effort, and included batch, interactive, data processing, and simulation systems. As compared with industry averages of 2,000 to 4,000 lines of code produced per workyear, these five projects averaged over 7,000 lines of code—and, on the largest of the five, the productivity was some 8,500 lines per workyear.

But increased productivity has not been the only benefit; program maintenance time has also been reduced. Exxon sees several reasons for this improvement. The original code has fewer errors in it; for instance, in one of the five systems evaluated, less than one error per program was found during the first year of use. The design of a program is now less complex than it was under former methods, so the program is easier to modify. And the code is more readable, so a change is more likely to be made correctly.

In fact, Exxon has found enough benefits from the use of PST that it has become the catalyst for formalizing other methodologies. One of these is for designing the logical structure of a database. Another is for better defining user needs for application systems. And still another is PSTAIDS.

PSTAIDS is a prototype graphic software package that has been developed at one Exxon affiliate and enhanced by the corporate staff. The intent of this package is to automate part of the PST process. In use, the programmer/analyst first inputs the Jackson data structure and program structure hierarchy diagrams, for the program under development. The programmer/analyst creates these structures interactively, at a graphics terminal. Then PSTAIDS validates the hierarchy diagrams, allowing only sequence, selection, and iteration (per Jackson). Hard copy output of the diagrams is available within seconds. And, if the programmer/analyst has used PL/1 conventions, compiled PL/1 code can be generated.

Exxon has found that standardizing on one program design method has indeed improved software production and reliability. They view this as a foundation for formalizing other portions of the system development cycle.

Wells Fargo Bank

Wells Fargo Bank is an international banking company with headquarters in San Francisco, California. It has 366 offices around the world and provides a full range of banking services to corporations and consumers. It is ranked twelfth on the 1978 *Fortune* listing of the top commercial banking companies.

In 1975 the vice president of systems development became interested in the structured methodologies, in the hope of reducing the number of systems personnel then doing software maintenance. He calculated that 75% of the 200 people within the department were involved in maintenance work. (We gather that this is a very typical percentage in systems departments). With a three-year flat budget cycle and several new development projects on the horizon, he felt they needed to institute a more effective methodology for developing software systems. He wanted maintenance to drop to 50% of the systems work in the long run.

In early 1976 the people at Wells Fargo heard about the Constantine/Yourdon (C/Y) structured design method. After some investigation, they decided to standardize on it for the development of new software and for the maintenance of current systems, where possible.

The original concepts of the Constantine/Yourdon method were developed by Larry A. Constantine in the mid-1960s. Drawing on that work, Glenford J. Myers of IBM wrote about composite design in 1973. More recently, Edward Yourdon has refined the more abstract ideas in Constantine's work, translating them for more practical use. The method is also known as structured design.

Wells Fargo assigned a group of four people to (1) co-ordinate the training of the entire department, (2) assist in the use of the new technique, (3) develop new standards, (4) monitor the initial projects for adherence to the new standards, and (5) pass along practical suggestions learned from project team to project team.

During 1976 all 200 members of systems development were given a one-week in-house training course on the C/Y design method. It concentrated on structured design for three days and on structured programming for two days. All managers within the department took the class along with the programmers and analysts. Wells Fargo believes that this approach to management training has given the managers an appreciation for the difficulty of introducing the new technique, as well as an understanding of the new concepts.

One of the first projects to be developed using the C/Y method was a very large, technically complex, and highly visible system called CYCLESORT. It contains 25 COBOL programs, some of which are very large, containing over 5000 lines of code each.

The CYCLESORT project began in early 1976, with the project team estimating a completion date of February 1, 1977. For two months, two analysts performed the analysis of user requirements. Due to the high visibility of the project, and their unfamiliarity with the new development techniques, the designers went so far as to write many of the specifications in detailed pseudo code.

During the next five months, four analysts performed the system design, using the C/Y conventions. First, data flow diagrams were created and reviewed during design walkthroughs. From these, structure charts and a data dictionary were created.

Finally, seven months into the twelve month project, programming began. As could be expected, being over half way through the elapsed time for the project, and having no code yet written, people became very nervous—analysts and managers, as well as the users. But project leaders and management held firm to seeing the new method through by completing the design before beginning coding. It even took some 'wrist slapping' to accomplish this, we were told.

Six programmers spent the next five months coding the programs in COBOL from the structure charts and pseudo code created by the analysts. Structured walk-throughs were held to review *all* coded programs. These walkthroughs provided excellent cross-training, Wells Fargo found. Good coding techniques were recognized by team members and were imitated, so many of the programs had the same style.

Testing by the programmers consisted only of on-line testing for abnormal terminations. System testing went rapidly, with the system ready for implementation on schedule. Since then, few errors have been found in the system. Wells Fargo has performed a lot of tuning on it, such as re-coding high use modules to achieve more efficient run times. But the original integrity of the design has not been changed.

Management and project staff at Wells Fargo were very impressed with the results of the CYCLESORT project. They were particularly pleased at making the deadline for two reasons, one being that this was their first use of the new technique. Secondly, the users had requested many enhancements throughout the system design phase, adding to both project complexity and system size. Even with these problems, they were pleased to find that no huge gaps in the design became apparent during programming. Major flaws in design had been caught earlier in the development cycle, and the requested enhancements had been incorporated properly.

After two years of using the C/Y technique, Wells Fargo system management is committed to this approach. They feel that taking a firm stand, as they did in 1976 by training all of their people, is the correct approach. "Do not let the practice spread by word of mouth, or conduct several experiments with various techniques," they told us. "Getting systems people to change their habits takes a lot of work." Now, after a number of successful projects, they feel that their efforts have been worth it. Their percentage of maintenance work has not yet dropped to the desired 50% level, but it is dropping. And they expect the drop to continue, as new structured systems replace their older unstructured ones.

The three design techniques

To illustrate how these techniques are used to design programs, we shall simply give the gist of each method, rather than state the step-by-step procedures presented by the developers. And we have taken the liberty of using our

own terms rather than theirs. For illustration, we use a simple payroll program that has two inputs—timecards and a payroll master file—and yields four outputs—a payroll register and an updated master file for each company division and a paycheck and a deduction slip for each employee.

Although we are only talking about program design in this issue, both the W/O and C/Y methods can be used for *system* design also.

Programs are made up of two types of statements—control statements and action statements. Structured programs are limited to three kinds of control statements—sequences, iterations and selections—but they are not limited to certain kinds of action statements.

The basic differences among the methods we are discussing are: (1) the order in which the program logic is developed, either beginning with the control portion and moving to the action portion, or vice versa, and (2) the method then used to put the two together.

The Warnier/Orr technique

The Warnier/Orr (W/O) technique is based on the work of Jean-Dominique Warnier, in his book *Logical Construction of Programs* (Reference 1). Warnier originally developed this technique for designing programs, and he used flowcharts as an intermediate step between his design diagrams and coding. Kenneth Orr (Reference 2) added ideas for designing systems, data files and manual procedures. He eliminated the flowcharting step, and he gave special emphasis to data design. (Orr's ideas are not the same as Warnier's LCS discussed last month.)

The W/O technique starts with an output definition phase. The user, with the help of the systems person, sketches the desired outputs, including all of the necessary data fields. These may include paper reports, documents, screen displays, etc.—anything the user will work with—plus the updated master files.

From these outputs, the programmer uses the Warnier diagramming technique to decompose the outputs into individual input data items, either captured or computed. For example, a paycheck as output would require employee name, date, and dollar amount as input. When all outputs have been thus decomposed, a list of the necessary input data items is made, with all of the redundancies removed.

The data diagram consists of columns separated by brackets. It shows the hierarchy of the data. And it contains symbols for the three control types, showing how the various data items are related to each other. So the Warnier approach begins with the development of a data control structure from which the program control statements will be derived.

At this point the basic design of the program is pretty well established. The next few steps have been introduced by Orr to refine and verify the design.

For the next step, Orr introduces the study of time sequencing. The user-defined outputs are 'scheduled' into the normal processing cycles of the company. A Warnier-type diagram is also used in this analysis. Such normal processing cycles as year, quarter, month, week, and day are listed on the diagram; in our example, the payroll program outputs are associated with their proper cycles. The purpose of this analysis is to assure that all needed inputs, those obtained directly and those from other programs, will be available at the right time in each processing cycle.

The next phase is Orr's 'change analysis.' For every input item, the question is asked: "What real world event could cause this item to change?" For example, in our payroll program, we could ask, "What event would cause the organization codes on the payroll register to be changed?" One answer could be, "By a merger." In this case we might need new division codes, requiring changing the field lengths for certain organization codes. If this type of change can be accomodated on the existing Warnier diagram, then the design is sufficient. If not, then the new requirement (to accomodate the change) means that the entire analysis must be redone up to this point. Orr states that it is *changes* in data that tend to be forgotten in design, and these cause the bulk of design errors.

Also in this phase, changes in the processing cycles are considered. For example, what types of events could cause the payroll to be rescheduled? Some possibilities are holidays, disasters, and vacations. So perhaps a new 'midweek' cycle must be added.

Finally, program design is performed. Using the Warnier *data* diagram already developed, the control structure of that diagram is translated into an identical control structure for the program. And the data items on the diagram are translated into program action statements. One way to achieve this one-for-one translation is simply to write 'process' before each data item on the data diagram.

Using our payroll example, let's assume that we have D divisions in the company and each division has E employees. The top of the program hierarchy, on the left in the Warnier diagram, would be the *division level.* This level would include: start the payroll program, perform the division payroll (for each of the D divisions), and end the payroll program. The next lower level decomposes the division processing into *employee level* processing: start division processing, perform employee processing (for E employees), and end division processing. Division outputs, such as the payroll register, would be produced at this level. The third level decomposes the employee processing into its parts: start employee processing, print paycheck, print deduction slip, store line for payroll register, create updated master record, and end employee processing. This type of decomposition continues until all data elements and their sources (input or computation) are determined. The result is a program structure that matches the data structure.

The program's procedures are then pseudo coded from the program diagram. Each bracket on the diagram becomes a module; therefore, says Orr, the technique creates a structured program "without really trying." Users tell us that this pseudo coding step follows quite naturally from the diagram. The action statements come from the words on the diagram and the control statements come from the symbols. From the pseudo code, the program is coded in the appropriate language.

Orr's company offers a course on structured system development (Reference 3) in which they concentrate on the use of the W/O notation. But they also encourage the use of decision tables and design walk-throughs. In addition, the company offers an automated documentation tool, STRUCTURE(S), for creating W/O diagrams. And they are developing an on-line version to aid programmers in designing programs, systems, and data files, while working at graphic terminals.

The Jackson method

The Jackson method is based on the writings of Michael Jackson in England (Reference 4). Jackson's premise is that a program's control structure should look like its data structure. His method is aimed onlyat program design and coding, after the inputs and outputs have been determined; it does not begin with an input or output determination phase.

The Jackson method is marketed by Infotech International in both North America and Europe (Reference 5).

The Jackson method begins by developing data structure diagrams, using pre-defined inputs and outputs. A separate hierarchical diagram is drawn for each input and each output. Jackson uses boxes and lines in his data structure diagrams, with the three control types (sequence, iteration, selection) designated by symbols within the boxes. As an example, a payroll register could have the following data hierarchy: register, report, line, field, and sub-field. All involve iterations, i.e. several reports form the register, several lines form a report, etc.

In the next phase a comparison of these diagrams is made. The diagram of an input is compared to the diagram of the output it will support. If the two diagrams correspond at every level, then one program can be written to process the input into the output. If, however, the two diagrams do not correspond, then, says Jackson, there is a structure clash. In order to resolve a structure clash the input is written into an intermediate file from which it can be processed into the output form. Thus, two programs need to be written, one to process the input and one to process the output. One of these programs is then made a sub-routine to the other, without changing its design structure. If creating an intermediate file reduces run time efficiency, the subroutine probably can be tuned.

After the data structure diagrams have been developed, then comes the program structure. Jackson uses the same box, line and control notation for the program structure diagrams. In the case of an input/output match, each level

of the corresponding data diagrams is translated into a program level. For example, an updated master file would have the same data hierarchy as the original masterfile, so the top level on the program diagram would be 'process master file giving updated master file.' Lower levels would be similarly translated from the data diagrams.

So, at this point we have the control structure of the program. Next we need to add the action statements to that structure. To do this the programmer lists all of the operations that he thinks need to be performed. Infotech has a checklist of the various types of action statements to help the programmer completely determine all of those needed. These are then numbered, and each number is written on the program diagram at all appropriate places, to determine that the operation will be performed the correct number of times and at the correct processing time. For example, 'calculate division totals' would be placed at 'process division ending,' where it would be performed once per division following all other calculations. It would not be appropriate to place this action statement number at 'process division heading' or at 'process employee body' because either the processing placement or the number of times executed would be wrong.

If all of the numbers (representing the action statements) can be placed on the program diagram, then the design is complete. If not, then it is deficient and must be redone.

The resulting diagram shows the sequence in which the operations will be performed. The control statements and action statements are then jointly pseudo coded. Users tell us that this step follows quite naturally. Finally the program is coded in the appropriate language from the pseudo code.

Using the Jackson method, the people at Infotech estimate that the program development cycle typically consists of: 40% of the time spent on developing the data structure diagrams, 35% spend on translating these into program structure diagrams, 15% on pseudo coding, 5% on coding, and 5% on testing.

Infotech International Limited in England (Reference 5) recently announced a pre-processor for converting pseudo code into PL/1 and COBOL. They say this is the first phase of their development of an automated 'diagrammer.'

The Constantine/Yourdon method

The concepts of the Constantine/Yourdon method (C/Y) first appeared in a 1965 article by Larry A. Constantine. A 1974 article by Stevens, Constantine and Myers (Reference 6) gives the best overview of the method. Later, Edward Yourdon and Constantine wrote a book, *Structured Design* (Reference 7), which has become the reference text for the Yourdon courses on structured design. More recently, a number of other consulting firms have begun teaching courses using these same concepts. We originally based our discussion on the Yourdon/Constantine book, as we understood it. The several reviewers of our writeup gave us other interpretations, from which we have selected the following description.

The C/Y method is quite different from the other two methods discussed, because it first develops action modules and then places them into a control structure. It begins with pre-defined inputs and outputs—that is, one or more given inputs that have to be processed to produce one or more given outputs. The first step is to discover all of the changes that must occur in the data to create the output from the input. These changes represent the actions to be taken on the data. The diagramming technique that the C/Y method uses for this step is the 'data flow diagram' or 'bubble chart.' It contains circles (the changes to be made) and lines (the changed data). It does not contain symbols for the three control types.

For our payroll example, we can list the following changes that turn timecard data into paycheck data: the timecard data must first be validated and then matched to a payroll master record. From that combination we calculate the pay and update the year-to-date fields. With that data in hand, we format the paychecks and then print them. This represents a single stream of data in a very simple problem. In most situations there are numerous inputs, outputs, circles (bubbles) and lines. Ideally the data flow diagram is constructed beginning with a few 'high level' bubbles. Then these bubbles are expanded into their component parts on separate sheets of paper.

The next step is to translate the lower level bubble charts into a hierarchical chart of functional modules, called a structure chart. This represents putting the action items into a control structure. The structure chart contains boxes, lines, control symbols, and data references. The top level of the chart contains the controlling module(s); we would call it 'payroll' for our example.

The most difficult part of this translation from data flow diagram to structure chart is determining which modules belong at the top levels, we are told. If these are not chosen correctly, the design will be difficult to maintain. One method for choosing the top modules, called transform analysis, involves tracking the input and output data streams inward on the data flow diagram. The inner-most bubbles discovered from this analysis form the top level modules on the structure chart.

For our payroll example, we can use the transform analysis technique to find four central bubbles—matched timecard, calculate pay, update year-to-date, and format paychecks. These four functions would form the second level on our structure chart; the top level would be 'payroll.'

Following this initial pass at creating the structure chart, the C/Y method recommends studying and revising it (1) to determine if the module hierarchy is suitable, and (2) to simplify the connections between modules. The connections are the parameters (either data or control flags) passed from one module to another. Bugs in programs, especially bugs caused by changes during maintenance, are transferred through, as well as caused by, these connections. So keeping interfaces simple will do much to isolate errors and ease maintenance. Proponents of the C/Y method say that this interface study is very important for good program design, and that it is totally missing in other methods.

The C/Y method offers a number of aids for this module study. These include studying module cohesion, coupling, span of control, and scope of effect/scope of control. We do not have space to describe these concepts here. Theoretically they aim to help the programmer determine if the interfaces are indeed simple, if the modules each perform only one function, and if the control hierarchy of the program is too complex. In practice, the concepts are rather difficult to grasp and then use, we gather. Programmers are not able to keep the connections or modules as simple as they would like. On the other hand, programmers who use the methodology feel the concepts do lead to good designs.

Next, the procedures to perform each module are pseudo coded from the structure chart. Some say this pseudo coding does not follow naturally from the structure chart. Others say that if the specifications for the lowest level bubbles in the data flow diagram have been expressed in structured English or decision tables, the pseudo coding step is not difficult. Yourdon told us that recent work at his company indicates that the W/O or Jackson method can be used at this point to design the inside of the modules.

Finally, the program is coded in the appropriate language from the pseudo coded modules.

From our talks with users, it appears that the C/Y method is the most challenging of the three techniques. While the steps are well defined, the procedures to use in each step appear to be difficult to grasp and then use. Despite these apparent difficulties, it is a popular design method.

We do not know of any available products for automating this methodology, but we suspect that such products are coming and will be of assistance in assuring the completeness and consistency of the designs.

Choosing a technique

In this report, plus the previous two reports, we have described a number of analysis and design techniques. What criteria should you use to choose the most appropriate ones for your company from among these?

Stevens (Reference 8) reports on a comparative analysis of several design techniques performed at the National Bank of Detroit. Their objective was to find a technique that would cause the least 'upheaval' within the bank. His brief summary of the study includes the following list of selection criteria. It is the most complete list we have seen in the literature.

Looking at the *user interface* of each technique, Stevens asks, "How good is this technique for communicating with users?" He points out that not all of the methods recommend user design reviews. Those that do include this phase are better, he feels. Also, the various diagrams differ in how easily they can be read and understood. Diagrams that include sequencing, hierarchy and are symmetrical are the most useful for talking to users, he says. And design errors tend to stand out more readily. Stevens emphasizes the user interface—and we agree that it is important, for an interesting reason.

We found in our discussions at companies that the users picked up the particular graphic technique of the design method and used it for totally unrelated purposes, such as for work assignment scheduling. So a good graphic technique can become more than a design and communication tool. It can evolve into a *de facto* standard method for decomposing complexity, which many people in a company can come to understand and use.

Related to the readability of the diagrams are the criteria of *controlling accuracy, completeness and quality* of the design. Stevens notes that control often depends on the number of people capable of reviewing the design. If users can understand and change the diagrams, the design is more likely to reflect their needs, and more likely to lead to high quality software.

Stevens also feels that it is best for a technique to be *consistent.* For one thing, does it use only one type of diagram or several? Secondly, does the technique tend to lead to similar good solutions when used by different people? Methods that concentrate on the sequence of events within the problem and the data structure tend to lead to consistent designs more than methods that rely on insight and inspiration, says Stevens.

Related to consistency, Stevens asks if the technique is a *good programming aid* as well as a design aid. All of the techniques are top-down and support successively refining a problem level by level. But not all lead naturally from designed modules to compilable code. Here again he asks, "Does the method leave a part of the process up to intuition and judgment?" Successive decomposition is best when it leads to a level very close to code or pseudo code.

Looking to the future, we suspect that those techniques that naturally lead to code will be the first to have automated code translators available, since the translators will be easier to develop. So we would add *suitability for automated aids* to Steven's list of criteria.

One question about any graphic technique is its *maintainability* during design. Is it easily modified, or does it need to be substantially redrawn when changes occur? The further along in the design process, the more likely a high level specification change will require numerous 'rippled' changes in a diagram. None of the techniques preclude this problem, but some make it easier to handle. Also, how easy is it to spot where these changes need to be made? A change in function or data may be easier to track on one type of diagram than on another.

Additionally there is the advantage of using these design diagrams to document the system for future maintenance purposes. For this use, Stevens again asks, "Are the diagrams easy to draft, comprehend, and update?"

With the advent of automated design tools, which we shall discuss further on, we see the system maintaining the diagrams. So the manual drawing procedure will disappear, but the need to track the rippling effect of a change will not.

Stevens next asks: "How *teachable* is the method and how readily will it be *accepted* by systems people?" He favors methods that are easily learned, all else being equal. These require shorter training periods and less follow-on consulting.

Finally, Stevens asks, "Is the technique *hardware and software independent*?" He found those that he studied to work with all types of applications and equipment.

A successful introduction

Once a method has been selected, how can it be introduced successfully? We define a successful use of a design method as one that is still in widespread use within a company after two years. You may ask, why two years? And

why not a more gradual introduction? The reason is that if the company does *not* make a concerted effort to turn the technique into a programming standard, then its use will die out. And its benefits will not have been realized. Programmers prefer to spend their time coding, not designing. It takes a real effort to reverse this natural desire. As we discussed above, that is what must be done. With these techniques, programmers will spend the bulk of their time designing, not coding.

The companies we talked with have met this two year criterion, so we would call them successful users. How did they achieve this success? Well, from our discussions with them, we see a pattern for successfully implementing a standard design methodology. It has the following stages.

Obtain management commitment. Getting programmers to alter the way they approach a problem requires getting them to use a new method to find out for themselves that it does improve program quality. They need to be 'converted.' This conversion cannot be taught nor proven to them; they must experience it. And very likely they will resist the new technique at first. Therefore, the first step is to gain management commitment. Management should be sold on the method—for committing the money for training and follow-on consulting and for the needed patience to see the first few projects through. Without management commitment, use of the methodology will be spotty. Management must continually say, "This is our programming standard now; you will use it and it only."

Initial training. Once management is convinced that the standard is worthwhile, then there needs to be training, and a lot of it. *Train everyone in sight* seemed to be the motto of the companies we visited. Train not only the analysts and programmers but also systems management and users. While users do not need to learn the technical aspects of the new method, they do need to know the new procedures. As we mentioned, the development life cycle is dramatically changed using any of the new methods. Over one-half of the development time will be spent creating only design diagrams, not code. This is not what users expect, so they need to be forewarned that this does not mean that things are going poorly. Actually it should mean that the project is going well, and the resulting system will be more to their liking.

So system development management needs to spend money to train its personnel, either by contracting for the training services or by sending people to outside classes. Live classes are necessary. Books and video taped courses are useful but not sufficient, we were warned; they just do not provide the interaction that is needed by the technicians.

Follow-on consulting. While one training course will give the basics of a new technique, it does not assure usage. Programmers will need some 'reassurance' help during their initial use of the technique. Someone very knowledgeable needs to be around to review design diagrams on a person-to-person basis and in structured walk-throughs. This can be a company employee or a consultant; the companies we talked with used both.

After the programmers have used the methodology on a project, they request little help. But answering the initial requests is essential, we were told. Using the C/Y method, where it is difficult to move from one stage to another, consulting help is particularly crucial.

So acquiring on-going help facilitates the correct use of the new techniques.

Expect mid-project panic. The people we talked with said that in some cases there was initial programmer resistance to the use of the new techniques. But the real panic occurred about half way through the project, when the team was still working on design diagrams. Users, management and even project members were used to seeing some code quite early in a typical project. This does not occur when using the new methods. And no matter how much warning has been given, when the length of the design phase actually does double, panic sets in. The fear, of course, is that the design will not be *that* much better to make the coding go *that* much faster than in the past. So the team members begin to think about cutting the design phase short. "Let's go with what we have,

it's pretty good" or "We could *at least* start coding these sections," they say.

It is at this point that management and project leaders need to remain firm about completing the design phase before beginning coding. This takes a lot of determination, we were told. But when the project is over, everyone will then say, "Yes, the design is a whole lot better than in the past, and yes, the coding did go a whole lot faster. So, yes, we were right to force the team to follow the methodology's life cycle."

Perform an audit. The first few projects using the new method probably will not go as smoothly as desired, and the teams may not actually be using the methodology quite properly. For these reasons, and to reassure the staff of management's commitment to the use of the new technique, we recommend an audit. It should be performed by outsiders who know the technique and can assess whether it is being used correctly.

Those then are the stages to successfully introducing a new standard program design method. Now we need to ask, "Is it worth it?"

Is standardization worth it?

Our discussion illustrates that standardizing on a program design methodology can be done, but it is difficult and expensive. Is it worth it? Well, the people we talked to think it is. The benefits they are receiving practically speak for themselves. Their development cycle is more predictable, their designs are more complete, their maintenance is greatly reduced, their software is more reliable, and their documentation is created during development, not as an after-thought.

In addition to these benefits, we see one further reason for standardizing. We expect the expanded use of these more popular methods to lead to the development of automated program design tools. Where there is a market, someone will surely create a product to sell. The automated documentation tools already mentioned are the first step. Let's see what future products might look like.

Indicative of what will be offered, we think, are two interactive development systems already on the market. The first of these is the Bell Labs' 'programmer's workbench' system, which runs on DEC PDP-11 computers under Bell Labs' UNIX time-sharing operating system (Reference 9). It is used to develop software not only for DEC equipment but also for other computers, such as IBM and Univac. Bell Labs has licensed some organizations (such as Interactive Systems Corporation of Santa Monica, California) to sell and maintain UNIX. The second is the Maestro Programming System, which is offered in the U.S. by Itel Corporation (Reference 10). It differs from the Bell Labs' system in that it is stand-alone, with its own processor. It can handle up to ten programmer work-stations, which are graphic CRT terminals.

We expect announcements of similar system development work-stations (and their enhancements) to become commonplace in the 1980s. So we plan to discuss this topic in more depth in a near-future issue.

All of the methods we have discussed use diagrams of one kind or another to help the programmer put some structure to the problem at hand. The drawing of these diagrams certainly can be (and has been) mechanized. We also see aids in future work-stations providing valuable help to programmers in four design areas: checking, translation, routine aspects of design, and documentation.

Checking. Some types of completeness of designs can be checked by a computer. The batch-run documentation tools now in use do have some checking capabilities; for example, they check to see that each selection operation has at least two output paths. We expect this checking function to become much more sophisticated in future products.

Translation. A design method that takes a programmer naturally into pseudo code can be automated to perform this diagram-to-pseudo code translation. It can also do the pseudo code-to-code translation. It is not too difficult to imagine a programmer performing the decision tasks in program design and letting the computer perform these code translations. This is analogous to our moving further and further away from machine language coding through the use of higher level languages.

Routine aspects of design. We see the automated tools significantly helping programmers perform the routine aspects of design. Interactive versions will allow programmers to design on-line, much as they program on-line today. Cheap micro work-stations will replace pencil and paper, and significantly speed up the design process.

The argument against on-line programming initially was that programmers could not *think* at terminals. Well, this argument has proven to be untrue; programmers do sit and think at terminals. And we expect the same to become true of on-line design. They will not worry about letting inexpensive terminals tied to micro-computers stand idle while they think.

We also expect the computer to store numerous design aids, such as lists of common operations used in the Jackson method, decision table routines, commonly used modules, and, of course, the ability to produce the graphics of the program design method.

Maintenance. Finally we see automated products maintaining the entire design process: updating diagrams, keeping project statistics, providing electronic work areas, linking the programmer work-stations to other systems, etc. The ease with which these facilities can be used, as well as the features themselves, will need to be considered in future selections.

So, to the question, "Is it worth standardizing on a program design method?" we answer, "Yes, because it will very likely provide benefits in software development today and lead to automated program design tools in the future." Automated products are in the offing, we feel, and instituting the use of one of the more popular methods will place a company in a better position to evaluate and take advantage of these new developments when they become available.

REFERENCES

1. Warnier, J-D. *Logical Construction of Programs*, Martinus Nijhoff, Social Sciences Division (for U.S.: 160 Old Derby St., Hingham, Mass. 02043; for Europe: Pieterskerkhof 38, Leiden, The Netherlands).
2. Orr, Kenneth T. *Structured Systems Development*, Yourdon Press (1133 Avenue of the Americas, New York, NY 10036), 1977; price $12.50.
3. For more information on the structured systems design course, contact Langston, Kitch and Associates, 715 East 8th St., Topeka, Kansas 66607.
4. Jackson, Michael A. *Principles of Program Design*, Academic Press (111 Fifth Avenue, New York, NY 10003), 1975; price $22.75.
5. For more information on the Infotech Programming Technology method, contact Infotech International: (1) in the United States at 234 East Colorado Blvd., Pasadena, California 91101, and (2) in Europe at Nicholson House, Maidenhead, Berkshire, England.
6. Stevens, W. P., G. J. Myers and L. A. Constantine, "Structured design," *IBM Systems Journal* (IBM, Armonk, New York, NY 10504), Vol. 13, No. 2 1974; pp. 115-139; price $1.75.
7. Yourdon, Edward and Larry A. Constantine. *Structured Design*, Yourdon Press (address above), 1975; price $25.00.
8. Stevens, Bruce M., "Structured techniques: Comparative analysis," available from Langston, Kitch and Associates (address above), 1976.
9. Ivie, E. L., "The programmer's workbench—A machine for software development," *Communications of the ACM* (1133 Avenue of the Americas, New York, N.Y. 10036), October 1977, p. 746-753; price $5 prepaid.
10. For more information on the Maestro Programming System, contact Itel Corp. (One Embarcadero Center, San Francisco, Calif. 94111).

Prepared by:
Barbara C. McNurlin
Associate Editor

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
3. Program Design Techniques

(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_____