EDP ANALYZER

Editor and Publisher: RICHARD G. CANNING

EDP ANALYZER

OCTOBER, 1981
VOL. 19, NO. 10

APPLICATION SYSTEM DESIGN AIDS

Automated design aids have been commercially available since the early 1970s. New ones are appearing and older ones are being enhanced and strengthened. This month we look at user experiences with *two design aids* that are being used successfully for the development of complex application systems. One supports design activities for both batch and on-line systems, while the other uses simulation for designing on-line systems. Also included is a brief discussion of *designing distributed systems.* And the new Commentary section presents the views of a consulting firm on the *prototyping process*, based on a recent survey they made.

The government of the Province of Ontario, Canada, with its headquarters in Toronto, has 22 ministries. Data processing is performed at three data centers, administered by the Ministry of Government Services, which employs multiple IBM 3033s, as well as at private service bureaus and on local mini-computers. Each of the ministries has its own development staff for computerized systems, and consultants are used extensively to augment this staff. For instance, the development staff in the Ministry of Consumer and Commercial Relations totals about 25 people.

In 1975, people in the Ministry of Consumer and Commercial Relations (MCCR) were looking for a better method of application system development. They read about PRIDE, developed by M. Bryce and Associates, Inc. of Cincinnati, Ohio, and decided to investigate it. At the time, none of their application systems were well documented and, because of this, maintenance was a problem. One of PRIDE's benefits was (and is) the extensive system documentation that is produced. MCCR investigated, liked what they saw, and acquired PRIDE.

PRIDE divides the system design and development process into nine structured phases, with each phase having well-defined 'deliverables.' These nine phases are based on a design process, not a project management process. As long as any of these deliverables have not been completed, the phase is not complete. We first discussed PRIDE in our December 1974 issue (which is where the MCCR people tell us they first learned about PRIDE).

ISSN 0012-7523. Multiple copy prices listed on last page.

At first, MCCR considered PRIDE primarily as a documentation tool. But in 1977, they began using it as a system development methodology. And in 1978, they acquired the systems and data dictionary/directory feature for PRIDE, which had been introduced in 1974, called 'Logik.' When they ordered Logik, they planned to use it on a trial basis for 60 days—but after two weeks of use, they decided that it was 'indispensible.' (We discussed Logik in our January 1978 and February 1979 issues). Finally, in late 1979, MCCR obtained the latest enhancement to PRIDE—the automated design facility (ADF). All three products are integrated into one product called PRIDE/ASDM, (for automated system design methodology).

The automated design facility. As indicated earlier, PRIDE/ASDM divides system development into nine phases, beginning with the initial system study and ending with the audit of the installed system, computerized or not. Logik (now called the 'information resource manager') provides an automated dictionary/directory function, for storing the definitions of systems, subsystems, organizational entities, data, procedures, programs, outputs, and so on. It provides system designers with design diagnostics and documentation. The design method is based on the concept of *chronological decomposition*—grouping outputs by the time cycles in which they must be produced.

In the automated design process, as an analyst performs the initial system study and begins to see what the users want from the new system, he/she enters the user's first ideas on what outputs—both scheduled and on-request—are desired. Each of these outputs is defined in terms of its cycle (such as daily, weekly, etc.), offset within a cycle, and response time requirements. The ADF analyzes all data flows back to their sources, plus other checking, that is usually very tedious to do manually. In addition, each output is defined in terms of the data fields that are expected to be included in it, grouped into logical records, and each data field can have up to 38 logical and 13 physical attributes.

As its first step ADF creates a *rough logical design*, or model, of the overall system. It uses the definitions just mentioned—data, cycle, offset, and response time—to group the outputs into compatible sub-systems. It attempts to use existing data files and sub-systems as much as possible. ADF can do a pretty good job of matching the new system data to existing data, we were told. Where a new logical record almost matches an existing one, ADF points this out and indicates what it thinks is needed in order to achieve a complete match.

This information—the grouping of logical records into logical files, and the indication of matches and almost-matches of logical files to physical files already in Logik—is printed out for study by the analyst. If changes or corrections are needed, the relevant outputs can be re-defined or some run-time parameters can be changed.

When the design looks satisfactory, the analyst proceeds to use ADF to perform the functions of phases 2 and 3—that is, dividing the overall system into sub-systems and then performing the more detailed design of each subsystem. Note that this is still *rough design*, based on the analyst's preliminary ideas of what the user wants. Each sub-system (that is, each logical grouping of outputs within a specified time frame) is defined in terms of its stored data and necessary input data.

For each sub-system, ADF determines the administrative (manual) procedures that will be needed for supplying input and the computer procedures that will be needed for producing the desired outputs. The outputs are not shown in report format, but rather are indicated by listing the data fields that each will have.

The point of all of this is to show the analyst the implications of the design decisions. For instance, ADF sets up a logical file based on how the data is used (not on how it is stored). A logical file is created unless ADF can find a matching file in Logik. But the analyst may see that there are just too many logical files, indicating not enough common use of data. Or it may be apparent that the administrative procedures for supplying some data item may be impractical at that point in time. In any case, it is a relatively simple matter to go back to phase 2 and re-define some of the outputs, to improve on the design.

In practice, the analyst may go through this process—phases 2 and 3 for all sub-systems—to see what the implications are. After correcting any glaring problems, the system design is discussed with the user, using ADF design printouts (input, output, and record definitions, sub-system flowcharts, etc.). The designer may show the user some alternative designs at this point, to see which best fits the user's needs.

Typically, at this time the user begins to see changes that are desired. As mentioned, it is no big matter for the analyst to go back to phase 2 and add, delete, or change outputs. By going back to phase 2, this means that ADF files will always be up-to-date with the latest changes.

After, say, two or three such iterations, generally the user's needs will appear to have been met and more of the details of each sub-system will have been entered into ADF.

And, as indicated earlier, the information resource manager (Logik) portion of the overall methodology performs system design diagnostics—showing any outputs that are not supported by inputs, or inputs that are never used, and so on. So, at this point in time, the documentation which the programmers will use has been developed. The users will have signed off on the system design (ADF produces the necessary sign-off sheets!) and the gaps and overlaps in design have been corrected. Program design can then begin. For more information on PRIDE/ASDM, see Reference 1.

Usefulness of the automated tools. With their use of PRIDE, Logik, and ADF, which together make up PRIDE/ASDM, MCCR feels that they are in a position to evaluate what these tools can do for them, in aiding productivity in system development. For systems developed under a rigorous adherence to PRIDE and Logik, actual development times and costs are nearly always within 10% of the estimates. Also significant is the fact that the subsequent corrective maintenance requirements for these systems are negligible.

On one of their systems, for instance, in which they used PRIDE and Logik, they found no bugs and had no maintenance needs during the first five months of its operation. They see the overall maintenance efforts of PRIDE/ASDM-developed systems as being only a small fraction of the effort required with conventionally developed systems.

As far as the use of ADF is concerned, the 'reporter' portion produces the design manual, user manual, operations manual, glossary of terms, and flow charts. They find this documentation to be excellent. Moreover, they maintain the documentation of current systems on the computer, including flow charts and linkage definitions among system components, as support for system maintenance.

The 'designer' portion of ADF has proved to be more of a challenge. It produces useful, satisfactory designs for batch systems (which still constitute the bulk of the new system development at many organizations) and MCCR is using it for the batch portions of their systems. For on-line portions (which occur in all of their systems), the "jury is still out," as they said to us. However, ADF is undergoing continuing improvement and MCCR hopes that the enhancements they have been receiving will result in demonstrable savings in the area of on-line system design.

Software design aids

Software design aids are tools for improving the quality, maintainability, and cost effectiveness of custom software. These tools are becoming more and more visible as they become more widely used by data processing departments. In past reports we have discussed a number of such software design aids. One of the products discussed this month, PRIDE, plus others such as data flow analysis, PSL/PSA, and LCS, were discussed in the February 1979 issue. In addition, SADT and IA, which deal more with the system analysis stage of application development, were discussed in the January 1979 issue. This month, we discuss the types of functions that newer design aids are able to perform.

The goal of software design, and hence of software design aids, is to produce systems which perform satisfactorily for users, and, at the same time, minimize a system's *total* life cycle costs. User satisfaction is affected by the degree to which the user can particpate in the design as well as comprehend it. It is also affected by the flexibility of the system during design and

after implementation. Flexibility includes the ease of responding to changes in user or system requirements. Life cycle costs are all costs from the time the system is conceived until the time it is retired (which might be decades later).

Software design can be thought of as having two components: clerical and logical. The clerical tasks in design include consistency checking, organizing, summarizing, and formatting. All manual and automated design methodologies address clerical functions, to one degree or another. The early automated aids were aimed only at taking over many of the clerical tasks. Particularly on large projects, removing some of the clerical load from designers has improved the efficiency and effectiveness of the design task.

The other aspect of design is application logic. In this context, logic refers to the system aspects which derive outputs from inputs. This part of design is obviously much harder to automate than the clerical functions. Even so, simple logic functions such as validation—matching inputs to outputs— have been performed by design aids for some time. We expect more logical functions to be automated in the future.

A very significant portion of the life cycle costs of software is consumed in doing things over. During development, uncovering and correcting design errors is common; the further along in development that they are detected, the more costly it is to correct them. After the system has been implemented, the cost of software maintenance can easily exceed the cost of the original development project (in fact, by several times), over the system's life-time. Therefore, it makes sense to expend more effort in the design phase, in order to reduce both rework and maintenance.

From a designer's point of view, software maintenance falls into three categories: errors, functionality, and user learning.

Correcting errors, often called 'corrective maintenance,' includes correcting a program that does things wrong, as well as meeting any unfulfilled requirements of the system, whether implicity or explicitly stated. Many errors are uncovered when nonsense outputs appear during production use. Or when the system is interfaced with other systems, errors commonly occur in the communication between the systems.

The second maintenance category—functionality (sometimes called adaptive maintenance)—refers to the degree to which the system actually supports the area of activity for which it was designed. There are three major reasons why a system may have functionality problems. First, the area of activity may have changed. Or, second, it may not have been well understood by the designers. And, third, the system, in its current form, may not be well accepted by the users. All changes in functionality will require system design modifications.

Changes based on user learning—usually called enhancement maintenance—constitute an area of maintenance which is often overlooked by designers. Users frequently change their needs once they begin to use an automated system. Last month we described how developing systems by prototyping tries to anticipate this tendency by giving users the opportunity to experiment with a prototype of the system early in the development process. Then changes can be made during design, rather than during development, or worse yet, after installation.

General problems of designers

There are a number of problems which all designers face, and which design tools can help to improve. These are:

Inadequate specifications. Since it has not been the responsibility of software designers to become experts in the business area under study, they must rely upon users to supply this expertise.

However, users (to their surprise) typically do not have a clear idea of their needs. Until actual cases arise that help the users clarify the application logic they need, designers may get an incorrect understanding of all of the system's components. Hence user requirements typically are not well specified, and users generally (always?) end up supplying inaccurate and incomplete requirements information—which leads to incomplete and inaccurate system specifications.

Changes in design. System design studies may motivate users to think more critically about

both the functions that they are performing and their manner of operation. So even projects intended to automate a current manual system end up performing significantly different work. Users' desires tend to change over time, and day-to-day activities often bring to light relevant procedures that were overlooked in the requirements study. For projects extending over many months, numerous desired changes come to light and these must somehow be incorporated into the design. So change is continually being introduced in the design, making it very difficult to freeze design requirements.

Consistency. System design can involve many people and long periods of time. In these cases, personal preferences, as well as the memories of the designers, can lead to the introduction and propagation of inconsistencies. For example, different identifiers can be given to functionally equivalent portions of the system. And just the opposite is also common—the same name can be assigned to different functions. Such problems are compounded when the system is changed. These types of clerical problems can consume a good deal of a designer's time, as well as add frustration to the design effort.

Documentation. The whole area of documentation, from design through program maintenance, has been an industry problem. Without good documentation, a system ranges from difficult to nearly impossible to understand, correct, and modify. The quality of documentation depends heavily upon the skills and interests of the designers. And generally, documentation is not viewed as a creative exercise, so designers have little enthusiasm for it. Also, the skills needed to write good documentation are different from those needed for design or programming.

Data entry. There are three common areas of concern associated with data entry: data entry errors, data entry routines, and data entry conventions.

Designers need to consider errors of omission and commission, by helping users detect input errors and provide the capabilities within the system for voiding and/or correcting incorrect entries. The system should prevent users from mistakenly changing or deleting data by allowing such actions to be reversible. In other words, the system should be 'forgiving.'

As users become familiar with a system, the nature and frequency of their mistakes will change. For example, users will need less guidance from the system after they once learn how to use it. The design should reflect this learning by the users. To be efficient, data entry and validation routines should change to reflect user needs.

Also, conventions for data codes and data names should be established to avoid confusion. Avoid codes that are 'cryptic' strings of characters; they are highly error-prone. Conventions should be established for operating with missing data, so that the input rejection rate is not unnecessarily high. Output from the input validation routines can be used to communicate back to the users, to indicate the nature of missing or incorrect data.

Breakdowns. Computer technology has improved, so hardware and software breakdowns generally are no longer common everyday occurrences. However, such breakdowns do occur. In addition, breakdowns may result from electrical problems, data communication interruptions, and other causes, such as fires and floods. Designing the system to store data in preparation for an unforeseen breakdown is a responsibility of designers. So they need to consider the effects of all types of breakdowns at every point in the operation. Often certain types of breakdowns are overlooked. Also, designers need to include procedures to bring the system back to the state it was in just before the failure. The design should minimize the amount of either automated or manual effort necessary for complete recovery.

Operation schedules. Designers also need to consider peak load periods as well as how the manual and automated procedures will 'mesh' with each other. To this end, the design effort should include documenting the order of activities that must occur around the system (both before and after) so that scheduling conflicts can be resolved *before* the new system is implemented.

User training. The manner and order with which users perform required functions is de-

fined in the user interface. Many designers neglect to consider carefully the discipline the new system will require of users. The most easily accepted user procedures are those that are consistent with company or department standards. User training must be designed into the system, and implemetation strategy should be approved by the using department before the design phase is complete. User training is often overlooked by designers until implementation appears imminent.

Preliminary user manuals. In order to enable users to fully understand a system, user manuals should be developed and updated as part of the design process. Companies that write user manuals during design have told us that this documentation helps users detect and correct design errors and gives them an appreciation for the amount of user interaction and preparation that will be required by the system. Unfortunately, creating user manuals during design is often overlooked.

Concern for maintenance. Organizing data into tables improves the readability of program code and simplifies the task of updating a system. Changing business conditions often require that systems be changed. Designers must assume that parameters, such as constants and rates, will change. Using tables to store parameters is one method used by designers who are concerned with system maintenance. Future maintainability is often overlooked during design.

The value of automated aids

Automated system development aids, such as PRIDE/ASDM (including ADF) can ease the problems just discussed. For example, the structure and detail required by such aids in themselves help to improve the adequacy of the system specifications. Also, the automated categorization and linking of design details allow designers to introduce changes easily and with considerably less human effort. The changes can be introduced at any time, as the need for them becomes apparent.

The ease of introducing and evaluating changes, in itself, helps reduce the problems that arise from incomplete and inaccurate requirements statements and specifications. So errors of omission are reduced.

Also, aids which sort data elements into logical groups and provide a data dictionary function can help designers spot inconsistencies more easily. Hence, errors of commission are also reduced.

Finally, the structure and detail required by the aids leave less discretion to the designers about documentation details. The aids may produce much of the needed documentation automatically.

SIMULATION AS A DESIGN AID

Application systems can be conceived and initiated either from within or from outside the using department. The place of system birth can be an important factor in determining whether or not the system will be a success. In the design and implementation phases, acceptance by and co-operation of the users are very important; the system can have only limited value without them.

Those application systems conceived by users have a headstart in gaining their co-operation. Since the users are seeking assistance in implementing their ideas, they are likely to receive personal satisfaction and company recognition for completing the system. The users are more likely to channel their energies toward supporting such systems, instead of working against the designers.

Also, in user-conceived systems, the users have formalized—to some extent, at least—ideas on desired characteristics of the system before they approach the designer. The designer's job becomes one of helping these users complete the system conception and document the system in a clear manner.

Application systems can also be conceived outside the using department—from, say, corporate executives, from employees interfacing with the using department, or from the data processing department. Systems initiated from the outside are often viewed as a threat or annoyance by the using department. Therefore, the designers have an additional job of selling the system concept to the users in order to gain their co-operation. Often selling is done simultaneously

with the early design stages of information gathering and analysis. Before users will co-operate with such a design effort, they want to find out what benefits they will receive from the system.

Integrated systems, which are commonly conceived from the outside, particularly need involvement and co-operation from all of the affected departments. Designers often encounter a lack of user enthusiasm for these systems. In the eyes of the users, the benefits may not be apparent, while the troubles and 'hassles' may appear imposing.

So gaining user support, for integrated systems especially, is an added burden for designers.

In addition, as mentioned earlier there is the problem of getting users to specify the application logic with anything approaching accuracy and completeness. It is difficult for any user, in a series of interviews, to specify the complete range of situations that will arise and how each should be handled. Often overlooked are infrequent or non-periodic procedures or events.

Designers must therefore interpret or extend the users' specifications in order to complete the design. These actions by designers can lead to errors in application logic, which may not surface until after the system is implemented and the cases are encountered in practice.

Exception cases—where generally established conventions do not apply— are often a major portion of a system. Users think of their systems and procedures more in terms of generally established conventions. However, exception cases often require a greater portion of the design effort than do the standard logic flow. Improper handling of non-standard cases can severely limit the usefulness of a system.

So how can system developers gain user support for the new systems, as well as obtain a better understanding of the application logic required?

One approach to getting this user co-operation and feedback early in design is by 'simulation' of the new system. This can stimulate discussion and bring to light some of the aforementioned overlooked details. The term 'simulation' may seem ambiguous, especially since we used the term 'prototyping' last month in much this same context.

What is the difference between simulation and prototyping, you may ask?

By simulation we mean using an automated design tool that lets users see how the system *would look* from their viewpoint. A simulated system is not a full-blown prototype because it generally does not 'work.' With prototypes, as we have used the term, a workable system is created, into which users enter data and from which they can obtain useful outputs. A simulated system, on the other hand, cannot be used by people to perform work, because it generally consists of program 'stubs,' which return pre-determined answers, rather than full-blown modules that produce useful outputs. The purpose of simulation is to show users *examples* of the required inputs, the flow of the system, and the desired outputs. So simulation can be viewed as a first step toward a prototype. In fact, that is just how one company, the Bank of Nova Scotia, has used a design simulation tool called ACT/1.

Bank of Nova Scotia

The Bank of Nova Scotia is a federally chartered Canadian bank with 1,000 branches across Canada and in other countries. It is one of the five largest banks in Canada with assets over $40 billion (Canadian dollars). With headquarters in Toronto, the bank provides services to both consumers and industry.

Data processing work for the bank is handled at two large centers in Toronto, where they have three IBM 3033s, a 3031, a 370/168, and an Amdahl V7. Ten regional centers perform mostly remote job entry, via the bank's data network, and some local processing, such as check processing.

In 1979 the data processing department decided that it needed to create an integrated code library system to keep track of the some 15,000 software modules, computer programs, IBM job control language procedures, and screen definitions they had accumulated at their computer centers. The library system was envisioned to perform seven functions: (1) track code changes and keep a history of the changes; (2) maintain code libraries, such as a test library, production library, and development library; (3) promote code from, say, created status to tested status, or

demote code from, say, from production status when creating a new version; (4) release code, which requires linking all current modules with new modules and giving the entire system a new release number; (5) provide administration functions, such as creating reports of activity for project leaders; (6) maintain descriptive documentation of all items in the library system; and (7) provide backup and security for the library system.

The bank decided to use three outside packages to perform some of the functions, rather than write these programs in-house. One package is Librarian, from Applied Data Research. It would be used to maintain the source code in the library system. The second package is Datamanager, from MSP. This data dictionary product would be used to define and maintain the documentation of the modules, programs, etc., as well as the relationships among them. And the third package is ACF 2, from Cambridge Systems. It would be used to restrict access to sensitive items in the library system, where desired, as well as to provide data security functions.

The system was to operate in an on-line mode under IBM's time-sharing option (TSO). The question was, how could the bank most effectively design this large library system to best meet user needs and to incorporate the three packages? They wanted to be able to test the design as early in the development process as possible to be sure that users' needs would be satisfied and that the packages would indeed work together. In mid-1979, about the time that the requirements for the system had been completed, the bank learned about ACT/1, from Art Benjamin and Associates, of Willowdale, Ontario.

ACT/1 is an on-line development system for designing and running on-line applications. It has two parts—a design aid and a production system.

The design aid allows a designer to sit down at an IBM 3270 CRT terminal (or its equivalent) and: (1) create input and output screen formats ('screens'), (2) create user dialogs and menus, and (3) specify the logic flow among these components. Some 'screens' can be used to represent the output of subroutines that will be written in the final system. ACT/1 keeps track of all components and flags discrepancies, such as screens or routines that have been identified but not developed.

By entering some sample data, the designer can move through the application and simulate its operation on-line. The people at Art Benjamin and Associates call each simulation a 'scenario.' During a simulation, logic flow and screen formats can be changed. Also, comments made by users or other designers can be attached to specific screens of the application, using an on-line NOTE facility. Notes can be used for several purposes, such as to identify and describe needed audit and control features, planned enhancements, programming specifications for needed subroutines, and documentation in general. Users can operate the scenarios themselves for verification or training purposes.

The production system allows an application that is created using the design aid function to be run in a production mode, once the necessary subroutines have been coded.

ACT/1 performs the mainline logic—generating the screen formats that have been defined, interpreting user responses, invoking appropriate routines, editing and reformatting data, controlling the application logic, and passing data between the various components. Coded subroutines only need to be written to access the database (or files) and perform calculations. These routines may be coded in COBOL or PL/1.

ACT/1 requires a System/370 instruction set and IBM 3270 display terminals (or equivalent). It interfaces with TSO, CICS, or INTERCOMM, operating under OS/VS2, MVS, CICS/VS, (under DOS/VS(E)), or VM/CMS and is compatible with the leading database management systems that run on IBM equipment. For more information on ACT/1, see Reference 2.

The bank's system development cycle. Use of ACT/1's production system (which includes the design aid) allowed the Bank of Nova Scotia to have a highly unusual, yet very effective, application development cycle for their integrated library system.

Based on the system requirements that had been already drawn up, the bank used ACT/1 to create four successively more powerful versions

of the library system, within an eighteen month time period. The process began in late August 1979 when the team of five programmers and analysts created their first 'scenario' of a 'version zero' system. As described above, the scenario was a simulation of how the system would operate; it was created entirely on-line using the ACT/1 design aid function. The purpose of this scenario of the version zero library system was to show the eventual users—programmers and operations people within the bank's program change control groups—how the system would operate from their point of view. As it turned out, the scenario was a very important design tool, because these users were able to visualize how they would use the system—and before significant time and money were spent. They detected a number of missing functions that the version zero system should perform. If these functions had not been included in the version zero system, it would not have been accepted for production purposes by the users.

After the simulation (or scenario) had been refined and was satisfactory to the users, the development team added the necessary subroutines to make version zero a working prototype of the final system. The subroutines they wrote linked the various purchased packages to the system and performed the functions not provided by the packages.

For coding the subroutines, the bank used MetaCOBOL, from Applied Data Research, in order to increase their COBOL coding productivity. Both MetaCOBOL's structured programming facilities and a bank-written MetaCOBOL interface to ACT/1 were used.

The purposes of the version zero effort were to study the internal design of the system—to create a design where all of the parts would fit together—and to demonstrate the utility of the library system for the intended users. Four months after they starting using ACT/1, in December, the version zero 'prototype' had been tested by some users and was ready for use as a temporary production system.

Up to this point, requirements definition had taken two work years of effort, and building the prototype had taken one work year, as compared with the original estimates of eighteen work years to develop and implement the system. The prototype was not considered to be the 'final' system, but it solved the users' basic needs and could be used for the time being in a production mode for a few development projects. Two new development projects, plus a maintenance project and the library development project itself, were chosen to use the prototype to maintain the code on their projects.

During December, the department was given approval to create a full production version of the system. Again using ACT/1, and "now knowing how we should have done it the first time," the team followed the same development cycle to create the production version. The team developed a scenario and then gradually added the subroutines to create the production version. From their experience in building the first prototype, the team was able to develop the version one production system within five months time, by May 1980. This version also ran under the ACT/1 production system, with the code that was not generated by ACT/1 or supplied by the other packages again being hand-coded in MetaCobol, using structured programming conventions. The version one system had about 20,000 lines of code.

Within the first five months of use of the version one production system, only four coding errors were found and no design errors were uncovered. Gradually, all users of the prototype were converted to the first production version and the remainder of the implementation effort—to get all 15,000 modules, programs, JCL procedures, etc. under the system—began in earnest.

But even this production version was not the final one; it did not perform all of the functions originally planned. During the remainder of 1980 and first quarter of this year, the system was revised two more times. Each time the designers used ACT/1 to go through the whole development cycle—from scenario to production version—and built upon their past experiences. Each time the resulting system performed more of the desired functions. The second production version was implemented in November 1980, and the third and final version was completed in March of this year. The final system now per-

forms all of the functions envisioned at the outset of the project.

The project took a total of 3,700 work days to complete, which was very close to the original estimate of eighteen work years. In all, the bank is pleased that ACT/1 has allowed them to create this large (now 60,000 lines of code) system within schedule, within budget, with very few errors, and that meets users' needs.

How automated aids can help

Design aids can be useful in most application system development efforts, and certainly for the larger, more complex systems. Batch, on-line, and distributed systems are categories of applications that have been addressed by developers of design aids. For each of these general types of systems, the design aids provide specific assistance for the designer.

Some design problems, which were discussed earlier in this report, are either reduced or solved by the use of design aids. The current state of the art in application system engineering enables these aids to provide a great deal of assistance, particularly in the clerical functions of design.

So far, however, only limited logical functions have been addressed by these aids. For instance, simulation helps to audit and refine the logical flow of control in an application. But it is the user of the simulation package, not the package itself, who is evaluating the correctness of the logic.

The main reasons for using design aids are to build better systems (which meet user needs better) and to reduce system life cycle costs. Not only should development costs be reduced but, perhaps even more importantly, maintenance costs should be greatly reduced. These maintenance costs include corrective, adaptive, and enhancement maintenance.

As the science of information system engineering advances, the quality of the design aids will continue to improve. The aids will become even more user-friendly and will require less effort on the part of the designer and the end user. Also, the aids probably will be enhanced in their ability to assist in the logical aspects of the design.

Application design is still an art. Several important aspects of design cannot be automated effectively. One of these aspects is that of selling the system to the users; another is helping the users in the transition from the old system to the new. Also, automated design aids are very limited in their ability to make good decisions as to the optimized use of resources.

Then, too, the responsibility of marrying the technology and the automated tools to the human aspects of a user department cannot be automated. Success of a new system is almost always measured in terms of its acceptance by its users, and its ability to perform required and desired functions for the users. The new system must be integrated into the operations of the user department, and it is hard to visualize this function being automated.

Even so, automated design techniques can greatly improve the technical soundness of an installation. They provide capabilities not available to designers using manual methods. These capabilities help to reduce the life cycle costs of the system. They allow the designer to give users some previews of how the system will operate, at various stages of development. They often allow the user to interact with a prototype of the system, to provide feedback to the designer on how the system should operate. And they allow the designer to make changes to the system design with a great deal less effort than if the changes had to be made manually. Automated aids help the designer to catch errors of a careless nature, as well as oversights and consistency violations. Finally, they produce up-to-date documentation rapidly, including versions of users manuals that are available during the development process.

The discipline and the structure required by design aids have been successful in helping designers to create more complete and maintainable systems. But their chief value may well be in the area of reducing life cycle costs, and particularly the maintenance costs, of a system. We can safely say that, if a well designed aid is properly used on a project of appropriate size, it will help achieve a reduction in system life cycle costs—and probably a substantial reduction, at that.

DESIGNING DISTRIBUTED SYSTEMS

For the design of a distributed system, most of the good design principles are the same as they are for a centralized system. But there are some things that *are* different, which a designer must keep in mind when laying out the structure of a distributed system.

This point comes through clearly in a new book by Robert L. Patrick, *Application Design Handbook for Distributed Systems* (Reference 3). Patrick gives a good coverage of the design principles that can be used for both centralized and distributed systems.

We will point out some of the principles that Patrick presents that seem most relevant for distributed systems.

A distributed system, in a broad sense, can be defined as a network of two or more computers which communicate and share resources. Three types of distributed systems are emerging: (1) hierarchical systems, (2) networks of co-operating processors or work-stations, and (3) stand-alone departmental systems which can communicate with each other and/or with central processors.

At the top of a hierarchical system are one or more computers with complete, overall responsibility. They often receive communications on transactions from all computers in the system; sometimes, however, only summaries of transactions are received. At the bottom are computers with very limited responsibilities. In between are computers with varying degrees of responsibility.

In a network of co-operating processors or work-stations, the processors are all at the same level; there is no higher or lower level. These processors can draw on each other for data or processes, in the performance of their work. Each processor has some unique data or process modules.

Stand-alone departmental systems are not a throw-back to the days of the 1960s, when things got out of control with a multitude of small computer centers. Instead, today's departmental systems must meet some corporate standards and must be able to communicate with each other and with the company's central computers.

We have selected five design principles from Patrick's book which illustrate, we think, some considerations that apply particularly to distributed systems. Specifically, these considerations apply to those cases where portions of the application may exist on more than one computer, and where co-ordination among the computers is necessary.

Portable processing modules. In order to achieve flexibility and back-up within a distributed system, the processing modules should be portable, says Patrick. The design of each module should consider hardware and software characteristics of the entire system, so that the modules can be used at multiple nodes.

Should a node's hardware fail or be destroyed, that node's workload can be taken over by other nodes. Also, as the total workload increases, new processors can be added to take on some of the work. So designers need to consider company-wide requirements when designing an application to run on a distributed system. And one of the most critical components of portability, that must be considered, is uniform and consistent data definitions.

Output distribution analysis. In designing a distributed system, Patrick says, a careful analysis of the output requirements is needed. The reason it is needed is because some outputs may be required at locations other than where the data is processed. So both the points of origin and the points of destination for output data must be identified.

This analysis should include all sites that are expected to use the outputs individually, as well as the composite need for outputs for the system as a whole. It may turn out, for example, that the same inputs, processing, and/or reports are required at multiple locations. The volume of activity may point out the most appropriate node(s) where the data should be stored and the processing performed.

Global processing controls. For transaction processing in distributed systems, not only might transaction sequence numbers and item counts be required but also the point of origin of the

transactions needs to be recorded. The need for such controls may not be apparent, because the source documents themselves do not move from department to department, as they do in manual systems. While this lack of movement reduces the chance of loss of documents, it does not reduce the chance of error.

Also, when the same data and/or processing modules exist at more than one node, controls are needed to ensure consistency, priorities, and the ability to track errors and changes to data and programs.

Security controls. As access to a system increases, as is the case with distributed systems, security measures are needed to protect the integrity and privacy of data and programs, says Patrick. Controls and audit trails provide the information needed to detect unauthorized accesses. All users should be required to pass through one security module, in order to use programs and data.

Controls such as these cost money. These costs must be weighed against the loss that might result without the controls.

Internal program structure. Distributed systems should be designed for maintainability, because they generally are more tightly coupled to a population of users than are many centralized systems. The effects of computer down-time and program errors can be felt by these users very quickly. So programs should be designed in a straight-forward manner and the number of inter-module connections should be minimized.

The reader will find a wealth of practical design principles such as these in Patrick's book.

REFERENCES

1. For more information on PRIDE/ASDM, and ADF, write M. Bryce and Associates, Inc., 1248 Springfield Pike, Cincinnati, Ohio 45215.
2. For more information on ACT/1, write ABA Software Industries, 250 Consumer Road, Willowdale, Ontario M2J 4V6, Canada.
3. Patrick, Robert L., *Application Design Handbook for Distributed Systems*, CBI Publishing Company (51 Sleeper Street, Boston, Mass. 02210), 1980, price $18.95.

Local computer networks have been receiving increasing attention in both large and small firms (some with only two dozen employees). But most firms will soon be finding that their new within-building communication needs will be much broader than are generally envisaged today. Next month, we will discuss a new integrated approach to within-building communications that will meet the needs of data processing, voice telephone, security, life safety, and many other new needs. If you are considering a local computer network, or planning to rewire your building, get acquainted with this approach.

Then, in December, we will discuss a question associated with the rapidly growing use of minis, 'personal' micros, and work-stations in business. The question relates to the state-of-the-art of software portability for these machines. We'll discuss both horizontal (between different brands of machines) and vertical (to more powerful machines) portability. If small computers are spreading in your organization, there are some things you can do to promote software portability.

COMMENTARY

PROTOTYPING—A METHOD NOT TO BE MISSED

By E. K. Somogyi, Butler Cox and Partners, London, U.K.

(Because of the clearly growing importance of prototyping, which we discussed last month, we asked Butler Cox and Partners if they would prepare this Commentary. Ms. Somogyi has been surveying users of alternative system development methods in the U.K., Western Europe, and U.S., for a near-future report of the Butler Cox Foundation. This Commentary draws upon some findings of that survey.—Editor)

In recent years, system development methods and approaches have received much attention. Among the various new methods, one in particular stands out—a method that is quick, iterative and in many ways the opposite of what has been practiced ever since methods were invented to develop systems. It is called 'prototyping.'

Several advanced facilities available on large and small machines make it possible to perform prototyping and to develop systems faster than before—provided the basic requirements of the new system are understood reasonably well. These facilities include databases and their management systems, data dictionaries, report generators, high-level non-procedural languages, screen formatters, and the like. With them, system modifications are also easy to make, so that iterative enhancement of the system is possible.

DP prototypes exhibit some essential features of completed systems. A prototype is not a static image; it can be used and exercised. It can give an early visualization of the system, and users can experiment with it. Using a prototype in this mode helps to clarify requirements and to finalise the user interface of the system. The likely future effects of a new or modified system are also easy to detect. Designers may gain a better understanding of the future system, so prototyping with the user helps to create better systems. Also, a prototype occasionally becomes the final system.

In short, prototyping as a method offers possibilities that no other method provides: speed, easy modification, and quick delivery of new systems.

Before adopting the prototyping process, however, it is wise to consider some of the shortcomings and problem areas that are associated with the process. Because prototypes often are not 'designed,' but rather are 'put together,' their internal technical arrangements may be haphazard and inefficient. They may not be resilient nor very efficient when used with large amounts of data or large numbers of users. Their documentation is usually sketchy. They often are best described as experimental systems.

For example, we have come across an undocumented prototype that was released into production. Since this organization did not have the necessary funds available to develop full documentation, it now must rely on a single individual (the one who developed the system and thus knows it) to maintain

the system. Another prototype was documented at some expense and the system was installed in an interactive multi-site environment. However, it had to be withdrawn rather soon, as response time and general performance 'hit the bottom' when large numbers of users started using it.

Uncontrolled, endless iterations may also create unsatisfactory results. It is simple, easy and very impressive to 'knock together' programs, files, and simple data entry routines and then hand the system over to the user for experimentation. It is less impressive when, after several modifications, nothing works, file contents cannot be relied upon, or several routines have been accidentally deleted. Then, too, the iterative process becomes boring and ineffective when constant change is the norm and no permanent result is produced.

It is wise, therefore, to separate the process of trying out the prototype from making major modifications to it. This allows time for both designer and user to reflect on the experiment and may prevent unnecessary changes. It is also wise to set a deadline for developing, modifying, and experimenting with the prototype.

Prototyping is only possible when some of the basic tools listed earlier are available. Prototyping acquired its tools 'second hand,' since they were originally designed for other purposes. This makes them in some sense inefficient for the prototyping process. Specifically, the tools provide very little information about the characteristics of the system that is being prototyped; for example, it is not easy to experiment with different response times. The tools, as yet, do not provide the designer with an easy way to record the essential features of the prototype that must be built into the final system. Neither do they provide facilities for tuning and measurement. These analytical features would be most helpful in creating more humanly-engineered and efficient systems.

With the possibility of so many things going wrong or being inefficient, the question must be asked: why use prototyping at all? Past experiences indicate that systems more often go wrong for reasons of inadequate specification than for bad technical engineering. As long as the requirements are those of humans, the method of 'getting the requirements right' must allow for iteration, for the presentation of visual images, and for the use of examples. Further, the method must promote a rapid, two-way communication process between designers and users.

Prototyping, with all of its shortcomings, is the only genuine iterative method available to us to meet these needs.

SUBJECTS COVERED BY EDP ANALYZER IN PRIOR YEARS

1978 (Volume 16)

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

1979 (Volume 17)

Number	*Coverage*
1. The Analysis of User Needs	H
2. The Production of Better Software	H
3. Program Design Techniques	H
4. How to Prepare for the Coming Changes	K
5. Computer Support for Managers	C,A,D
6. What Information Do Managers Need?	C,H
7. The Security of Managers' Information	L,C,A
8. Tools for Building an EIS	C
9. How to Use Advanced Technology	K,B,D
10. Programming Work-Stations	H,B
11. Stand-alone Programming Work-Stations	H,B
12. Progress Toward System Integrity	L,H

1980 (Volume 18)

Number	*Coverage*
1. Managing the Computer Workload	I
2. How Companies are Preparing for Change	K
3. Introducing Advanced Technology	K
4. Risk Assessment for Distributed Systems	L,E,A
5. An Update on Corporate EFT	M
6. In Your Future: Local Computer Networks	F,B
7. Quantitative Methods for Capacity Planning	I
8. Finding Qualified EDP Personnel	J
9. Various Paths to Electronic Mail	D,M
10. Tools for Building Distributed Systems	E,B,F
11. Educating Executives on New Technology	K
12. Get Ready for Managerial Work-Stations	C,A,B

1981 (Volume 19)

Number	*Coverage*
1. The Coming Impact of New Technology	K,A,B
2. Energy Management Systems	M
3. DBMS for Mini-Computers	G,B
4. The Challenge of "Increased Productivity"	J,K,A
5. "Programming" by End Users	C,H,B,G
6. Supporting End User Programming	C,H,B,K
7. A New View of Data Dictionaries	G,B,
8. Easing the Software Maintenance Burden	H,B,G,
9. Developing Systems by Prototyping	H,B,G
10. Application System Design Aids	H

Coverage code:

A Office automation
B Using minis & micros
C Managerial uses of computers
D Computer message systems
E Distributed systems
F Data communications
G Data management and database
H Analysis, design, programming
I Computer operations
J Personnel
K Introducing new technology
L Security, privacy, integrity
M New application areas

(List of subjects prior to 1978 sent upon request)

Prices: For a one-year subscription, the U.S. price is $60. For Canada and Mexico, the price is $60 *in U.S. dollars,* for surface delivery, and $67 for air mail delivery. For all other countries, the price is $72, including AIR MAIL delivery.

Back issue prices: $7 per copy for the U.S., Canada, and Mexico; $8 per copy for all other countries. Back issues are sent via AIR MAIL. Because of the continuing demand, most back issues are available.

Reduced prices are in effect for multiple copy subscriptions, multiple year subscriptions, and for larger quantities of a back issue. Write for details.

Please include payment with order. For payments from outside the U.S., in order to obtain the above prices, *use only an international money order or pay in U.S. dollars drawn on a bank in the U.S.* For checks drawn on banks outside of the U.S., please use the current rate of exchange and add $5 for bank charges.

Editorial: Richard G. Canning, Editor and Publisher; Barbara McNurlin, Associate Editor. While the contents of this report are based on the best information available to us, we cannot guarantee them.

Missing Issues: Please report the non-receipt of an issue within one month of normal receiving date; missing issues requested after this time will be supplied at the regular back-issue price.

Address: Canning Publications, Inc., 925 Anza Avenue, Vista, California 92083. Phone: (714) 724-3233, 724-5900.

Microfilm: EDP Analyzer is available in microform, from University Microfilms International, Dept. P.R., (1) 300 North Zeeb Road, Ann Arbor, Mich. 48106, or (2) 30-32 Mortimer Street, London WIN 7RA, U.K.

Declaration of Principles: This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional service. If legal advice or other expert assistance is required, the services of a competent professional person should be sought. — — *From a Declaration of Principles jointly adopted by a Committee of the American Bar Association and a Committee of Publishers.*