EDP ANALYZER

©1977 by Canning Publications, Inc.

JULY 1977
VOL. 15, NO. 7

GETTING THE REQUIREMENTS RIGHT

It is still too often the case that computer-based application systems are developed behind schedule, over cost, do not do as much as promised, and do not satisfy their users. After twenty years of concentrated attention, why do these troubles continue to arise? A good part of the answer to this question is: because the requirements for these application systems were never stated accurately and completely in the first place. If the requirement statements are erroneous or incomplete, how can the resulting application systems be expected to perform satisfactorily? At long last, this problem area is beginning to receive the attention it deserves. Here is what we found in our study of how organizations are trying to get the requirements right for their application systems.

Mattel Toys, a division of Mattel, Inc., with headquarters in Hawthorne, California, is a major toy manufacturer. Annual sales are in the order of $250 million and the company employs about 10,000 people in peak seasons. The company's data processing is performed on an IBM 370/155 operating under OS. There are 15 programmers and 8 system analysts in the management information services department.

In 1972, Mattel was performing a growing number of one-time applications, to the point where they decided to look for some software to help them. These applications were statistical analyses of field survey data. After investigating several software systems, they purchased the MARK IV system from Informatics Inc., Canoga Park, Calif.

Then in 1974, they investigated a number of data base management systems on the market and selected ADABAS, marketed in the United States by Software AG of Reston, Virginia.

As things have turned out, these two software systems—MARK IV and ADABAS—have been useful to Mattel for getting their application system requirements right. But there have been a few surprises along the way.

The MARK IV system proved to be very useful for the one-time applications for which it was obtained. So the question naturally arose: could MARK IV be used effectively in other situations, to ease the development of new application systems? Mattel decided to experiment with it. One rather extreme case was interesting.

In this particular case, a manager in a user department asked for a new, moderately complex application system. The request seemed reasonable, the benefits appeared sufficiently attractive, so the project was approved. A system analyst who had been trained in the use of MARK IV was assigned to the job. The analyst and the user department manager defined what the new system should do—its inputs, its outputs, and its processing. The analyst then set up the new system quickly, using MARK IV, and began giving the manager output reports. Some changes were needed, and were easily made with the MARK IV system. Very soon, the system was giving the manager what he wanted. Overall time to implement the system was about one month.

The system was used by this manager for a time, and then use began to fall off. Later the manager was transferred to another department—and use of the system stopped altogether.

Looking back at this project, the people at Mattel feel that there was something fundamentally wrong with how the project was conducted. They had no complaint with the software tool; MARK IV provided a means of easily setting up and changing an application system. The fault lay, they feel, with the trial-and-error approach that was used. Yes, it could easily correct minor errors in the system design. But it did not get at the basic problem. "It cured the itch, not the disease," they said to us.

Mattel feels strongly that it is preferable for an analyst to spend more time on analysis of requirements and design of the new system, with relatively less time spent by programmers on coding and testing. Good software tools can only supplement the analysis phase, not replace it.

Now, when a detailed study of a user request is authorized, the system analyst is expected to look at the problem in an overall context. The analyst assumes that the system will be built the way the customer has requested it, but still looks to see if there is a better way. The analyst is expected to make a basic problem identification and definition. What is the business problem? What is user department management trying to accomplish with the new system? Where does the problem really lie? Is this a one-time problem or a recurring one?

When the analyst feels that he understands the problem, he develops a preliminary design of the new system. Often, a "typewriter simulation" of the new system is used. That is, the analyst types up sample output reports, using real data, just as would be produced by the new system. The analyst goes over these reports with the users, to find out how the users will actually use the reports. This step usually uncovers more new requirements and more changes in design, we were told.

With the preliminary design developed, the analyst is in a position to estimate development costs, operating costs, and maintenance costs. If the system is to be a production system, it is designed to minimize people time, machine time, and maintenance time. The choice is made at this point as to what software tools will be used—MARK IV, COBOL, ADABAS, or other.

But no matter how well the analysis has been performed up to this point, it is almost certain that changes will be required before the system is completed. The people at Mattel ruefully cite two typical user comments. Just as programming is about to begin, a user says casually, "Oh, by the way, can you also give me . . ." Then, while conversion to the new system is underway, a user slaps his head and says, "Oh, my God, I am going to need . . ."

Both MARK IV and ADABAS provide a good degree of the flexibility needed to handle such changes, we were told. And when production systems have been programmed in COBOL, it has proved to be reasonably flexible, too, they said.

With MARK IV, it is very easy to change report formats, add new fields to a report (assuming the data is in the file), and so on. Also, a number of user departments have learned to use MARK IV for setting up their own application systems and developing new reports from existing systems.

With ADABAS, it is much easier to add new fields to records in the data base than it is to add new fields to records in conventional application files, say the people at Mattel. With conventional files, adding data fields may require changes to all or many of the programs in the application system. With ADABAS, adding data to files involves no modification to the programs that make no use of that data; they never see the new data. So the changes in requirements that come to light as systems are being developed are much easier to make with an ADABAS data base.

Both MARK IV and ADABAS are providing Mattel with more flexibility for adjusting to changes in their application system requirements.

The need for flexibility

Just how important is this need for flexibility in adjusting to changes in system requirements?

The user's requirements for a new application system should state *what* the new system should do, in order to solve the user's business problem. The most basic requirement, therefore, is that the new system address the right problem.

But even when the problem has been correctly identified and is being addressed, it is still possible to have errors and omissions in the requirements. (No, that statement is not quite right; it is likely that there *will* be errors and omissions in the requirements.) As the new system is being implemented, these mistakes come to light and have to be corrected. Here are some examples of the types of requirements mistakes that we have witnessed.

Missing or incomplete requirements. A particularly discouraging type of mistake occurs when the users and system builders concentrate so heavily on the main functions of the new system that they completely overlook an essential sub-system. We remember a case that happened some years back in which a manufacturing company was developing a new computerized production control system. The project team was well into programming when someone asked why no financial data was being collected by the system. It was suddenly realized that the whole financial subsystem had been overlooked. Most oversights are not of that magnitude, of course, but they still can cause considerable rework.

When the users begin to get output reports and documents from the new system, the need for additional data fields often becomes apparent. This generally means new data fields have to be picked up on input, carried in the records, and produced on the outputs. The reason for the oversight may be that the new system is doing things quite differently from the old system and the need for those data fields was not recognized. Associated with these data fields would be the logic for handling the new data, which might involve some subtle complexities. Adding the data and the logic can be difficult after the system structure has been established.

Other types of oversights include timing considerations (when things must happen), security and internal control requirements (which we discussed last month), terminal operating needs for on-line systems, and testing requirements for the new system.

Ambiguous requirements. Ambiguous requirements are ones where the user's interpretation differs from that of the system builders, due to imprecise statements. One area of difficulty is the transition period when converting from the old system to the new one—and what to do with the transactions that are in the pipeline during this period. The user may assume that the new system will take them over, while the system builders may assume that those transactions will be completed by the old system. Another problem area occurs when a point in time is treated as if it were a time period, or vice versa. Thus, "the first of the month" might have to be defined as 12:01 a.m. on the first day of the month, and not the whole first day.

Requirements errors such as these are common. As they come to light, the new system has to be changed to accommodate them. If the new system has been implemented with flexible software tools, the changes are much easier to make. We have already mentioned ADABAS and MARK IV. Comparable software tools include System 2000 and ASI-ST, to name two—and, of course, there are numerous other packages that should be considered.

With software packages such as these, it is usually easy to define new data fields and record types, and it is easy to change those definitions. Data fields can be added or deleted, lengthened or shortened, and so on. When such data definition changes are made, only those programs that use those data definitions may need to be changed; other programs accessing other parts of the data files are unaffected. Powerful retrieval and output formatting languages are available, for rapidly creating the programs for output reports and documents. So it is often possible to define a data file, load the data, and create user reports very quickly, sometimes in a matter of hours. And if the user wants changes made in these outputs, these can be made quickly, too.

Let us go back to the question of mistakes in the user requirements. Just how serious a problem is this?

The problem with requirements

The question must be asked, of course: do user requirements really represent a difficult problem in the development of application systems? Some people say No, the requirements are not the problem. The users simply state their requirements; the trouble is that the implementors are not able to meet those requirements. The main difficulties lie not with the users but with the implementors, say these people.

This report will support the viewpoint that the user requirements *are* the source of much of the difficulty. Typically, requirement statements are filled with a variety of errors, as described above. When there are errors in the requirements statements, it is unreasonable to expect that the resulting application system will be error-free.

The Second International Conference on Software Engineering, sponsored by the Association for Computing Machinery, by the Computer Society of the Institute of Electrical and Electronic Engineers, and by the U.S. National Bureau of Standards, was held in San Francisco, California in October 1976. A series of sessions at this conference considered the problem of system requirements and different solutions for getting the requirements right. Reference 1 is the proceedings.

In his paper presented at this conference, Harlan Mills gave the rationale (we believe) of why some people feel that requirements are not a problem. With manual systems, said Mills, the people who operated the systems used common sense to make the systems work. Managers could state vague requirements; over a period of time, the people operating the systems would make necessary adjustments to compensate for the vague statements of requirements would be suitable for agers have expected that the same type of vague statements of requirements would be suitable for computer-based systems. But that is not the case. Computers have no common sense; they just follow orders. So faulty or missing instructions can wreak havoc, said Mills. Getting complete and accurate statements of requirements for computer-based systems *is* a problem.

What is wrong with the conventional approach for stating requirements for computer-based systems? D. Teichroew and E. A. Hersey, at this same conference, reported that they had reviewed the literature covering many approaches to system studies and had found no agreement on the phases of a development project. Each organization used its own methods and standards. There was a consensus pattern, however. After a project was requested and initially evaluated, a detailed study would be authorized. A senior system analyst (generally) would study the existing system, develop the requirements for the new system, and then lay out the preliminary design of the new system. The users would then be asked to review this preliminary design and to indicate all necessary changes. Out of this would come the system requirements report that would provide the basis for the detailed construction of the system. By studying a number of these system requirements reports, Teichroew and Hersey reported that the average report consisted of about 30% text in natural language, 50% in lists and tables, and 20% in graphs, flowcharts, and drawings. But natural language is not sufficiently precise for stating requirements; further the reports tend to be voluminous, it is hard to assure consistency, and it is hard to tell if something is missing. In addition, continual change makes these problems worse.

Mills added another point to the difficulties encountered with the conventional approach. The user department people who really understand the present system—and who are in the best position to state requirements for the new system—are usually too busy to participate. So surrogate experts with more time available (guess why!) are assigned to the projects by the user departments. These surrogate experts give amateur opinions, which lead to missing and incorrect requirements.

D. T. Ross and K. E. Schoman, Jr. (Reference 1) pointed out other weaknesses with the conventional approach. One weakness is the inability of the people on the project to clearly see what the problem is, much less to measure it and visualize a workable solution. This is because the complexity of the application makes it difficult to understand. Another weakness is that the project members tend to think of the system architecture in terms of devices, languages, record formats, and so on, rather than thinking of the solution of the basic problem.

M. N. Jones (Reference 6) points out another difficulty with the conventional approach. The user communicates with the analyst who in turn may communicate with the system designer who in turn communicates with the program designer who then communicates with the programmers. Misunderstandings can and do occur at each level of interface, she says. Also, users miss good opportunities for getting more value from the systems because they are too busy to participate to any great extent and they do not understand what the computerized system can do for them.

What do all of these weaknesses result in? T. E. Bell and T. A. Thayer (Reference 1) report on studies they have made on requirement errors. Two projects were studied in detail, where it was possible to keep track of the errors as they were uncovered. The types of errors, as they occurred on these two projects, were: (1) incorrect requirements, (2) missing/incomplete/inadequate requirements, (3) unclear/ambiguous requirements, (4) inconsistent/incompatible requirements, (5) new/changed requirements, (5) requirements outside of the scope of the project, and (6) typographical errors in the requirements statements. On the smaller of the two projects, the requirements statements consisted of 48 pages of text and charts; by the end of the design phase, over 50 errors had been found, more than one per page. On the larger project, the requirements statement consisted of some 2,500 pages with over 8,000 uniquely identifiable requirements. Two major requirements reviews were conducted and these uncovered almost 1,000 uniquely identifiable problems, some serious enough to lead to the failure of the system to perform its mission.

One can conclude that it is very difficult to state requirements accurately and completely. If errors are likely to exist in requirements statements, what is the impact of these errors?

The impact of requirements errors

W. W. Black (Reference 1) points out that the things that put a development project behind schedule are the things that were not on the task list. The items on the task list are generally done close to schedule, he says. Incorrect and unclear requirements can lead to rework; missing and incomplete requirement statements mean that items must be added to the task list. So errors in the requirements statements might well be responsible for a good part of the schedule slippages that have occurred in development projects.

B. W. Boehm, in remarks given at the Software Engineering Conference, commented on the cost impact of errors in requirements. From the study of four projects, it was found that the relative cost of correcting an error increases exponentially with the project phase in which the error is detected. A requirements error that is not found until the testing phase can cost 10 to 100 times as much to fix as it would cost if it had been found during the specification phase. Further, conventional thinking says that perhaps 40% of development costs occur during the design phase, another 20% occur during coding, and the remaining 40% occur during testing. Looking at the actual costs over the life cycle of a system, these figures just are not right, says Boehm. Instead, design represents only about 12% of the costs, coding 6%, testing 12%—*and maintenance 70% of the costs.* Much of this maintenance is due to not getting the requirements right in the first place, he says.

So the rework of systems, due to errors in the requirements, leads to schedule slippage, cost overruns during development, and continuing maintenance costs. We are not saying that *only* the requirements errors lead to these; certainly errors are injected during design and construction. But the requirements errors are fundamental and appear to have a large impact.

In addition, requirements errors undoubtedly lead to user dissatisfaction with the application systems. Computer-based systems are not self-adjusting as are manual systems, as Mills points out. If the requirements are not stated correctly, then the systems are not going to perform to the user's satisfaction.

Conclusions about requirements errors

Requirements errors are a real and a serious problem. Further, they are quite difficult to prevent and to eliminate. The only safe assumption to make is that any conventional requirements statement (and resulting system specification) is filled with errors. Steps must be taken to detect and remove those errors as soon as possible.

Requirements errors can and do adversely affect a project in terms of schedule, costs, and system performance.

It is clear that much more attention must be paid to "getting the requirements right."

New methods for determining requirements

From our study of this subject, we see a seven element program for reducing the number of errors in application system requirements. These are the seven elements:

- Awareness of the types of errors
- User involvement
- Ways to handle complexity
- Formal inspection procedures
- Definition of performance
- Formal language for requirements
- Automated tools for analysis

We will discuss each of these briefly.

Awareness of types of errors

T. E. Bell and T. A. Thayer (Reference 1) have analyzed the types of errors found in requirements statement documents, as we mentioned earlier in this report. The analysis was made on one relatively small project, conducted in a university environment, as well as on a large military project. More recently, they have checked the results found from these first two projects on a third project and have encountered reasonably similar results.

The relative percent of errors in requirements found on these projects was as follows:

	% of Total Errors
Incorrect requirements	34%
Missing/incomplete/inadequate	24%
Unclear/ambiguous	22%
Inconsistent/incompatible	9%
New/change	3%
Outside scope of project	4%
Typographical errors	4%

Also as we indicated earlier, Bell and Thayer found an average of between one-half and one requirements errors per page of detailed requirements statements.

The point to be made here is that data processing management should recognize that a substantial number of errors will exist in most requirements statements unless specific action is taken to identify and remove them. The percentages found by Bell and Thayer may not apply in any particular other situation; we will discuss below how each organization might develop its own figures. But these figures do give some idea of where the worst part of the problem probably lies—in incomplete, missing, and unclear requirements.

It is true that errors can be introduced in the later stages of a project, such as in design, construction, or even during testing. Two comments are appropriate. The earlier the errors are introduced and the longer it is before they are caught, the worse the impact of them will generally be. So requirements errors might be considered to be the worst kind. Also, the methods proposed below for catching requirements errors can also be useful for catching errors introduced during the later stages of a project.

Thus if both general management and data processing management recognize that requirements statements are error-prone and that these errors lead to later difficulties, the stage is set for effective corrective action.

User involvement

It is trite to say that user management should be involved in the development of requirements—but, as the saying goes, "trite is right."

It simply is not adequate for user department management to spend only an hour or two with the system analyst(s), telling what is wanted in the new system, and then expect the system to meet their needs and desires. It also is not adequate for user department management to turn this responsibility over to a staff person.

A consultant friend of ours, whose firm has audited many system development projects, said that he thought that user management ought to think in terms of a two-day session for developing requirements for a typical business application system. This amount of time matches our experiences in the past, in developing system needs. But even this amount of time is not sufficient for determining *all* of the requirements. So "review sessions" or "walk-throughs" or "inspections" are needed in the specifications and design phases. In such sessions, which are usually two hours or so in length, user management can catch additional requirements errors and omissions.

Ways to handle complexity

Complexity is perhaps the root cause of requirements errors. This applies to both application systems and generalized software systems, such as operating systems. It is too often the case that such systems have too many details for one person to fully comprehend.

The point here is that unless the person(s) preparing the requirements statements fully understand the system under study, errors will be introduced. Some requirements will be omitted, some will be stated incorrectly, some will be unclear, and so on.

So the question is, how can the users and analysts *gain understanding* of what the system must do and must not do, in the face of the complexity that exists?

Here are several approaches that have been used successfully for gaining this understanding in the presence of complexity.

Progressive approach. The progressive approach is the "divide and conquer" approach. We discussed it in our October 1970 report and in several subsequent reports. The concept is that a big project is sub-divided into a number of smaller, stand-alone projects, none of which requires more than six or nine months to accomplish. Also, it is better if each project is done by a team of not more than three to six people.

Complexity will be reduced by tackling the big project as a series of relatively small steps.

It probably will be desirable to have at least a preliminary design for the overall system, so that each sub-project will integrate reasonably well with previous sub-projects. We have seen this done. It is hard to say if it can be done effectively in all cases—but we suspect that it can be used much more widely than is currently true.

Top-down analysis. One concept that can be used is described in IBM's Study Organization Plan (Reference 4). Start with an overall view of the organization—its products or services, its market, its competitors, the resources available, and so on. Then divide the organization into goal-oriented activities—cohesive groups of functions that aim to satisfy a specific part of the organization's market. The goal-oriented activities, in turn, are sub-divided into their specific component operations, data, data flows, and so on.

The concept of this particular form of top-down analysis is that understanding will be increased by viewing the specific operations performed by people in the organization in terms of what the organization is trying to accomplish in its marketplace. (And whether the organization is commercial, governmental, educational, or other, it *does* have a market for its products or services.)

Another currently-popular concept of top-down analysis has been given names such as functional decomposition or levels of abstraction. We discussed levels of abstraction in connection with structured programming, in our June 1974 issue. The idea is that the analyst first takes a broad view of the overall function to be performed. The analyst tries to identify all of the inputs, all of the resources, and all of the outputs that are appropriate for this top level view. In addition, the analyst attempts to list all of the component sub-functions that make up this top level function.

The process is then repeated for each of the sub-functions at the second level. Each sub-function is analyzed in terms of its inputs, resources, outputs, and component sub-sub-functions. By concentrating on one function at a time and deferring any lower level details for later considering, understanding is enhanced.

What results is a hierarchy of functions. IBM's HIPO and SofTech's SADT use this approach of the successive decomposition of a complex function into a hierarchy of understandable sub-functions.

Information flow and data analysis. This approach to the handling of complexity might be considered the opposite of the functional decomposition method just described. It aims at giving a complete picture of the function being studied, in both breadth and depth.

Information flow analysis of a function would be shown in flow diagram form; the process is described in Reference 3. For a large, complex function, the flow diagram might be very large, covering one or more walls of a room. Boundaries of existing applications within the overall function might be indicated on the diagram, thus indicating how the different applications tie together to make up the overall function.

Data analysis might be documented by a series of data definitions, tied to the information flow analysis. We saw one instance where the data definitions were written on small cards and attached to the walls of the study room, grouped by the applications within the overall system.

Information flow and data analysis are not as popular today as is, say, functional decomposition. A little later in this report we will discuss some of the advantages and shortcomings of each of these methods. Suffice it to say here that information flow and data analysis have been used successfully for handling complexity.

Iteration/convergence. This approach to the handling of complexity assumes that it is not possible to get all of the requirements right for a complex system before that system is built. This assumption holds even if one or more of the above-described methods of handling complexity are also used. This approach uses tools which make it easy to construct the new system, get it running, and to change it when errors are uncovered. By successive refinements of the system, it gradually converges toward the system that the users desire.

It is important for the system development organization to select some method for handling complexity. Actually, several of the above-described methods might be used in combination. There is no reason why the progressive approach, top-down analysis, and iteration/convergence cannot all be used. In fact, if we interpreted his remarks correctly, that was just what Harlan Mills was advocating at the Software Engineering Conference mentioned earlier.

Formal inspection procedure

B. W. Boehm, in his remarks at the Software Engineering Conference, reported on studies that showed inspections and walk-throughs were the most effective means for catching errors early. Formal requirements languages, standards, simulation, and automated tools were less effective (but still useful) for catching the errors.

Formal inspection procedures would thus seem to be mandatory if data processing management truly wants to get rid of requirements errors.

Formal inspection does have a connection with a project management system. Project management systems generally specify formal checkpoints during the phases of a project, as well as the work products that must be completed before each checkpoint. Project progress is measured when the checkpoints have been successfully passed. During a checkpoint review, management might be looking for any significant changes in project costs, schedules, or expected benefits. As far as the work products are concerned, the main question of the review is: "Have each of the required work products been completed?"

Formal inspection goes one step further. It asks: "What is the quality of those work products?"

M. E. Fagan (Reference 2) argues that formal inspections are more effective than walk-throughs. In a walk-through, someone (say, a programmer) describes to a selected, small group of people the flow of control and the handling of data in a program being reviewed. The group of people is expected to question the person and point out possible flaws, possible omissions, and so on. But a walk-through is largely an educational process, the participants get sidetracked into discussing design alternatives, and the process is not self-improving, says Fagan.

Fagan discusses his inspection procedure in terms of the inspection of program design and coding. It can also cover test planning and execution, documentation, rework, and other activities. We see no reason why it could not also be used at the end of the requirements and specification phases, for catching errors as early as possible.

Fagan's inspection program has five parts. (1) First is a short overview of the work to be inspected, presented by the analyst or designer who did the work. This is the educational, familiarization step. (2) Next, each of the inspection team participants is given copies of the documentation and is expected to do "homework" on it. Each one tries to obtain an understanding of the work by studying the documentation. Some errors may be caught at this point, but not the bulk of them, says Fagan. (3) The third step is the inspection meeting itself. The goal of this meeting is to *find errors*. It is up to the moderator to see that the discussion does not get sidetracked into, say, considering alternative designs. There should be no hunting for solutions to the errors. Just find the errors, says Fagan—and classify them by type and estimate their severity. Following the inspection meeting, the moderator is expected to write up an inspection report, listing all of the errors. (4) The next step is the rework of the work, to get rid of the errors. (5) Finally, at least the moderator (and perhaps the whole team) must perform a follow-up to see that all fixes have been made and made properly. Depending upon the number and severity of the errors, another inspection meeting by the whole team may be required.

In addition to the error report, there are several other outputs of this inspection procedure, says Fagan. One important output is the building up of a checklist of error types, their frequency of occurrence, and their severity. Coupled with this checklist will be a procedure of how to look for errors, particularly those that occur most frequently and/or are most severe. Because formal inspection reports are prepared, the inspection process itself can be analyzed, to see how it can be improved. Not only can these outputs be used by future inspection teams to improve inspections, they can also be given to individual analysts, designers, and programmers to show them what errors to guard against in the future.

Fagan cautions data processing management on this last point. Do not use these inspection reports for employee performance appraisal, he says. If they are used in performance appraisal, then the employees will look on inspection meetings as a threat rather than as an aid. They will fight the system instead of supporting it.

What about the inspection team? It should be a small team, of perhaps four people, says Fagan. The moderator is the key person and the success of the inspection process is very much dependent upon this person. He or she should be competent in the subject area but not necessarily in the particular application being reviewed. The moderator must keep the whole inspection process moving along and not let it get sidetracked.

The other team members might be (for a program that is being inspected) the designer, the coder, and the tester of that program. If one person does two or three of these functions, then find qualified substitutes for the other one or two spots on the team. If the program interfaces with other programs, then the programmers of those other programs can be on the team. As we say, Fagan discussed the procedure in terms of the inspection of programs. A comparable team can be visualized for inspecting the requirements for a new system.

It seems to us that a formal inspection procedure of this type is mandatory in any management program designed to detect requirements errors. As Boehm reported, it appears that inspection is the most effective single way for detecting and eliminating errors.

Definition of performance

There are really two distinct points to be discussed here.

Define performance of the present system. If management expects that the performance of the new system will be "better" than that of the system it replaces, some baseline is needed. The way to get this baseline is to measure the performance of the present system.

IBM's Study Organization Plan (Reference 4) provides a convenient method of documenting the performance of the existing system. The performance measures include the volume of transactions handled, the elapsed time for handling a given transaction, the man-hours required for the different operations, and so on.

Analysts frequently object to studying the present system in detail because "it is wasted time since the new system will do things differently." It is true that some of the "how" of the present system will be useless, as far as the new system is concerned. But the "what" and "how many" and "how long" of the present system *are* meaningful.

Develop performance evaluation tests for the new system. Qualitative requirements statements generally are unsatisfactory, such as "we desire more flexibility in adapting to changes in customer orders." Any such statements are unclear. They should be replaced by specific statements of what performance is expected of the new system.

M. Alford, at the Software Engineering Conference (Reference 1), pointed out that when a person attempts to write the validation tests for a performance requirement, he is forced to ask questions about that requirement and begins to uncover errors or omissions in the requirements statement.

To uncover requirements errors, every requirement should be stated in measurable terms and a test should be written to validate that requirement. That may not be easy—but it probably is a lot better than all of the rework and unhappiness that comes from building systems wrong due to requirements errors.

Formal language for stating requirements

Natural language is not well suited for stating requirements. Since natural languages allow for imprecision, there can be gaps, loose ends, and misunderstandings in the requirements statements.

A formal language for stating requirements has the advantage of precision. It is a constrained language, without all of the flexibility of natural language and has carefully defined constructs. When a series of requirements statements is made in a formal language, the likelihood is much greater that they will not be misunderstood, as compared with the same statements in natural language.

Formal languages for requirements statements can be in three forms. A *narrative language* uses a concise sentence structure and a defined vocabulary. Examples are the problem statement language (PSL) developed at the ISDOS Project at the University of Michigan (Reference 7), and the requirements statement language (RSL) developed at TRW (described briefly in Reference 1 and in more detail in Reference 9). A *graphic language* uses charts that must be developed according to a strict discipline. Examples are IBM's HIPO system (Reference 6) and SofTech's SADT (Reference 5). Finally, a *tabular language* expresses requirements in table form. Examples of this method include IBM's SOP (Reference 4) and NCR's ADS (Reference 8).

The use of a formal language by itself can be helpful, by encouraging more accuracy and completeness in the statement of requirements. In addition, if a narrative formal language is used, it is possible that it can also be used with automated analysis tools.

Automated tools for analysis

Two sets of automated analysis tools were briefly reviewed at the Software Engineering Conference mentioned above—the problem statement analyzer (PSA) of the University of Michigan and the requirements engineering and validation system (REVS) developed by TRW.

If the requirements have been stated in a formal narrative language, these statements can be input to the automated analysis tools. These computer programs can then help to detect such things as gaps in the information flow, all uses for each data item (including unused data items and conflicting uses for the same data item), conflicting names, and so on. In addition, they can be used to prepare management summary reports, for indicating how much of the overall requirements statements have been completed.

The debate on methodology

Getting the requirements right is a difficult and complex problem. As might be suspected, there is no clear consensus on how best to solve this problem.

As the above discussion has hinted, there are some significant differences of opinion on how to go about stating requirements. We would like to briefly review some of the chief schools of thought that we encountered in our study.

Hierarchy of functions vs. information flow

As we discussed earlier, the decomposition of functions and sub-functions into a hierarchy is proposed by some as the best way to handle complexity. Examples of this approach include IBM's HIPO, SofTech's SADT and Dijkstra's levels of abstraction method of structured programming. This approach concentrates on one level at a time, until it is fully understood, before moving on to the next lower level of detail.

At the other extreme is the information flow analysis, that traces the flow of all inputs through the system, until all outputs have been produced. The idea here is that the project team people can see the whole system, in its breadth and depth, and thus gain an understanding of it.

The adherents of functional decomposition argue that the information flow analysis can lead to huge charts which are too big for any one person to grasp mentally. And because of the size of the charts, it is very difficult to see errors of omission and commission. It is much better, say these people, to analyze one level of function at a time so as to minimize the errors of omission or commission.

The adherents of information flow analysis (for example, see Alford in Reference 1) point out some weaknesses of functional decomposition. It is hard to trace the flow of control for one input message through the hierarchy of functions. Also, it is hard to construct test cases for one specific sub-function in the hierarchy. Particularly in a real-time system, it is important to follow the flow of control for each type of input message, to make sure that the time constraints will be met under all expected conditions.

We have seen both methods used successfully but as yet have no general guidelines of when to use and when not to use either method. We are simply pointing out here that, while both are useful for handling complexity, each has its shortcomings.

Exhaustive study vs. interative/convergence

These two approaches might be sub-titled "do it right" versus "do it over."

On one side of the debate are the people who say: when you undertake a large system project, you must do an exhaustive requirements study—so that you do not find half-way through the project that you are doing something fundamentally wrong. It is just too expensive in time and money to have to change directions in the middle of a big project.

On the other side of the debate are the people who say: never undertake a big project in the first place. Divide a large project into manageable, smaller sub-projects. Moreover, build these sub-project systems in a manner so that they are easy to change.

However, as we shall see shortly, this statement of the two sides of the debate does not really express what is being debated. To set the stage for what is being debated, let us first consider the "do it over" approach.

In his remarks at the Software Engineering Conference, Harlan Mills claimed that human ambition (perhaps coupled with some of the requirements statement techniques discussed at the conference) tend to promote large projects. In fact, he said, ambition tends to promote very large projects, grandiose requirements, and grandiose specifications, where it can take several years just to develop the specifications. This is not the way to go, he said. Instead, undertake only smaller projects that can be managed effectively. Break a big project into a number of smaller ones. "We can conquer the world, three to six programmers at a time," he said.

With the smaller projects, the new systems can be set up quickly, the users can be given some outputs, any changes requested by the users can be made quickly, new outputs are delivered, and the cycle is then repeated. Since changes are relatively easy to make, no attempt is made to obtain an exhaustive set of requirements before the subproject system is designed.

This, then, is the iterative/convergence approach. It divides large projects into smaller ones and uses an iterative method to converge on the system solution that the user desires.

The "do it right" proponents do not disagree with the concepts of this iterative/convergence approach. The problem is, they say, that it is often not clear just how it can be used with a big system. User management may not be able to see how the overall system can be divided, in order to build one part of the system at a time and yet be able to integrate those parts later. It is difficult to foresee at the beginning of a large project all of the ways that the multiple parts will impact one another. These interactions of the parts can be very subtle. It may be necessary to do a detailed analysis of requirements for the whole system in order to be able to sub-divide the total system intelligently.

So, say the proponents of the exhaustive study, we do not disagree with the idea of sub-dividing large projects and implementing the parts quickly. But on large, complex system projects, we just do not know how to use that approach in its entirety. We feel that we must often make an exhaustive study of requirements before we can do the sub-dividing.

Another argument against the "do it over" approach is that the user may see only the effects of a basic problem and not the cause of that problem. So the user conveys, and the analyst accepts, a superficial analysis of the problem. Trial-and-error problem solving is not the way to approach a complex, challenging problem. The first iteration may be so far wide of the mark that convergence to a good solution does not occur. Moreover, this approach tends to promote laziness and sloppy thinking on the part of the analysts. Also, users are already too inclined to change their minds about what they want in application systems; this approach just amplifies such vacillations. So say the proponents of the exhaustive study approach.

We think that both sides make valid points. If the application system under consideration is not particularly large, then sufficient analysis of the requirements must be done to make sure that the right problem is being worked on. User management needs must be determined, by obtaining user involvement at the appropriate times. Then the iterative/convergence approach can be used.

On the other hand, if the system under consideration is basically a huge one—for example, as was the first computerized airline reservation system—then it may very well be necessary to make a thorough study of requirements before sub-dividing and implementing. The same comment can be made about a generalized application system that is supposed to serve tens or hundreds of user organizations, of the type that we discussed in our January 1977 report.

How to get the requirements right

What is a no-frills program that an organization can use to get the requirements right for new application systems and for major revisions to existing systems? Following is what we see as a minimum program.

Recognize the types of errors

The first step, it seems to us, is to recognize that requirements statements will normally have errors, and lots of them. Moreover, the longer it takes to detect these errors, the more serious are the time and cost penalties for correcting them.

Somehow, this fact must be communicated to all levels of management in the organization. This probably will not be an easy task. Further, in all likelihood, this message will not be well received. Top management and user department management may think that stating requirements is easy—and that if errors occur, they are the fault of the data processing people. It may not be easily accepted by these managers that many errors are likely and they are just as much the responsibility of the user departments as they are of the data processing people.

As we will mention below, a formal inspection procedure should be a part of even a no-frills program. As errors are detected, a list of error types will be developed. This list would seem to be the best possible argument to convince management about the errors that normally occur in requirements statements.

Get user involvement

As the list of types of requirements errors grows, it should become much easier to obtain the necessary involvement of user management. They can see how these errors in the past have caused project delays and added costs.

Even before this list is available, however, an attempt should be made to get the key *managers* (not just staff members) of the user departments to spend two full days or so in a "requirements session." This may not be easy to do, and it certainly requires the support not only of those managers but also of their superiors. But it can be done and the results are valuable indeed. We used this approach on numerous occasions some years back and it always provided very useful results.

Select an approach for handling complexity

Earlier in this report, we discussed a number of ways for handling complexity. All have worked well in specific instances. Here are our preferences for a no-frills program.

Progressive approach. Do everything in terms of small, short term projects which are done by small groups of people (3 to 6 people maximum) in a short period of time (6 to 9 months maximum)—but integrated into an overall program. If a big application is to be implemented, develop a master design, divide into a series of small projects, and develop these sub-systems individually.

Functional decomposition or information flow analysis. We do not have strong opinions that favor one of these approaches over the other. Currently, the functional decomposition approach is receiving a lot of attention, one reason being the wide exposure of IBM's HIPO method. But some means is needed for analyzing and documenting *what* the new system must do, what the performance of the present system is, and what the performance of the new system should be.

Particularly if the functional decomposition approach is used, a formal (graphical) language for stating requirements will be involved. It may also be desirable to translate from this graphical language to a narrative language, such as PSL, in order to use automated analysis tools. We see this as an optional step, depending upon the size and complexity of the total project (covering all subprojects), and not necessarily a part of a no-frills program.

Tools to make changes easier. The types of changes that will be involved include adding new data fields to the files so that they can in turn be added to reports, changing field lengths, changing relationships among data items, changing report formats, and changing processing logic. Experience has shown that some of the file management and data base management systems offer more flexibility for such changes than do conventional programming and data handling methods. Some people might disagree and say that conventional methods are sufficiently satisfactory—but we feel that some of these new tools are properly a part of a no-frills programz

Formal inspection program

As B. W. Boehm has pointed out, it appears that a formal inspection program (or the less formal walk-through program) is the most effective single method for detecting errors in requirements and design. So such a program should certainly be a part of a no-frills methodology.

M. E. Fagan has described a formal inspection program that has worked effectively at IBM. Others might take a somewhat different approach. We would think that Fagan's approach would be a good way to start an inspection program.

Define expected performance

Another powerful way for catching requirements errors is for the analyst to develop performance validation tests for all stated requirements.

This step does a number of things. For one thing, it identifies qualitative requirements. Such requirements either should be converted into quantitative requirements or, failing that, eliminated. Also, it detects all open ended requirements which are only partially defined. Open ended requirements are frequently stated in terms of examples, and it is left up to the "common sense" of the implementors to fill in all of the missing cases; this usually leads to troubles. This step also forces the analyst to determine just what constitutes good performance and what is unacceptable performance, for each of the requirements. As Alford commented at the Software Engineering Conference, trying to write the performance validation tests really points out what a person does not know about the requirements.

Conclusion

Progress *is* being made in the application of computer technology—but the problems of schedule slippages, cost overruns, and dissatisfied users still arise. Because of these problems, the data processing function has often lost credibility in the eyes of top management; management just could not depend upon the promises that were made. We believe that the situation is improving, as data processing departments have decided to limit themselves to smaller projects and not to promise as much. But the situation, while improving, still has not reached a satisfactory level, in our opinion.

Quite possibly the most basic cause of the schedule slippages, cost overruns, and dissatisfied users has been the errors that typically exist in requirements statements. It is also quite possible that this condition has not been properly recognized by the technologists until fairly recently. Most of the attention on how to get improved systems projects has been devoted to system design, programming, and data base design methods. It is finally becoming recognized that more attention must be given to ways for getting the requirements right.

The no-frills program that we have outlined for getting the requirements right, based on what we found in our study of the subject, does not seem to be too demanding. One change is in the frame of mind—simply acknowledging that numerous errors probably exist in any requirements statement. And most computer-using organizations have already obtained a degree of user involvement and have adopted methods for handling complexity. The changes that have costs attached to them are the software tools for providing flexibility to change systems, the use of a formal inspection procedure, and the use of performance validation tests.

"Getting the requirements right" is an area that deserves priority attention by data processing management.

REFERENCES

1. *Proceedings of 2nd International Conference on Software Engineering, October 1976*, in two volumes. Order from IEEE Computer Society (5855 Naples Plaza, Suite 301, Long Beach, Calif. 90803) or from ACM Order Department (P.O. Box 12105, Church Street Station, New York, N.Y. 10249). Be sure to get the 147-page second volume in addition to the 639-page first volume. Price $20.
2. Fagan, M. E., "Design and code inspections to reduce errors in program development," *IBM Systems Journal* (order through local IBM Branch Office), Vol. 15, No. 3, 1976, reprint G321-5033; price 50¢.

Packaged methodology referenced in this issue:

3. Hartman, W., H. Matthes, and A. Proeme, *Management Information Systems Handbook*, McGraw-Hill Book Co. (1221 Avenue of the Americas, New York, N.Y. 10020), 1968, price $29.50. This book describes the ARDI (analysis, requirements, design, implementation) approach developed by N. V. Philips-Electrologica, Apeldoorn, The Netherlands.
4. Glans, T. B., B. Grad, D. Holstein, W. E. Meyers, and R. N. Schmidt, *Management Systems*, Holt, Rinehart and Winston (383 Madison Avenue, New York, N.Y. 10017), 1968. This book describes the Study Organization Plan approach to system studies, developed at IBM.
5. For more information on SADT (Structured Analysis and Design Technique), contact SofTech, Inc. (460 Totten Pond Road, Waltham, Mass. 02154). SADT is a comprehensive methodology for doing functional analysis and system design, using functional decomposition and a formal graphical language.
6. For more information on HIPO (hierarchy plus input-process-output), see local IBM Branch Office; for a general coverage, order Form No. GC 20-1851; price $3.20. For a discussion of HIPO in use, see Martha N. Jones, "HIPO for developing specifications," *Datamation* (1801 S. La Cienega Blvd., Los Angeles, Calif. 90035), March 1976, p. 112, 114, 121, 125.
7. For more information on PSL/PSA (problem statement language/problem statement analyzer), write ISDOS Project, 231 W. Engineering Building, University of Michigan, Ann Arbor, Mich. 48104. PSL is a formal language for stating requirements and specifications, and PSA is a software system for analyzing sets of PSL statements.
8. For information on ADS (accurately defined systems), write NCR Inc. (Dayton, Ohio 45479). The method was written up in *ACM Data Base*, Vol. 1, No. 1, Spring 1969, which is now out of print. ADS is essentially a tabular method for stating requirements, and was one of the methodologies drawn on by the ISDOS project in developing PSL, above.
9. For more information on RSL/REVS, write TRW Defense & Space Systems Group, Huntsville Facility, 7702 Governors Drive, Huntsville, Alabama 35805. One pertinent publication is TRW-SS-76-02, "A Flow-oriented Requirements Statement Language," by T. E. Bell and D. C. Bixler, April 1976.

Additional reading

10. Mills, H. D., "Software development," *IEEE Transactions on Software Engineering* (IEEE Computer Society, address above), December 1976, p. 265-273; price $10.
11. Nunamaker, J. F. Jr., and B. R. Konsynski Jr., "Computer-aided analysis and design of information systems," *Communications of the ACM*, (ACM, address above), December 1976, p. 674-687; price $5 prepaid.
12. Leavenworth, B. M., "Non-procedural data processing," *The Computer Journal*, (British Computer Society, 29 Portland Place, London W1N 4HU, U.K.), February 1977, p. 6-9.

PRICE SCHEDULE

The annual subscription price for EDP ANALYZER is $48. The two year price is $88 and the three year price is $120; postpaid surface delivery to the U.S., Canada, and Mexico. (Optional air mail delivery to Canada and Mexico available at extra cost.)

Subscriptions to other countries are: One year $60, two years, $112, and three years $156. These prices include AIR MAIL postage. All prices in U.S. dollars.

Attractive binders for holding 12 issues of EDP ANALYZER are available at $6.25. Californians please add 38¢ sales tax.

Because of the continuing demand for back issues, all previous reports are available. Price: $6 each (for U.S., Canada, and Mexico), and $7 elsewhere; includes air mail postage.

Reduced rates are in effect for multiple subscriptions and for multiple copies of back issues. Please write for rates.

Subscription agency orders limited to single copy, one-, two-, and three-year subscriptions only.

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____________________