A panel session—User experience with new software methods

SESSION CHAIRMAN—VICTOR R. BASILI
University of Maryland

Panel Members

Donald J. Reifer—TRW
Donn Combelic—ITT
J. A. Rader—Hughes Aircraft Company
C. M. Bernstein—Exxon Corporation
F. T. Baker—IBM Federal Systems Division
Susan Voigt—NASA

PANEL OVERVIEW—Victor R. Basili

The development of correct, reliable, less expensive software continues to be a major problem. A great deal has been written and said about various techniques and methodologies for software development and how they are meant to aid in the development process. Unfortunately, most of the available material is by the author of the technique, be it individual or company. This does not always allow the outside user of the technique a fair appraisal or full understanding of how good the technique is, how to use it, and how to adjust it to his environment. This is true for several reasons. First, the author's experience is often limited to a specific application or set of applications and specific environments. There are some genuine questions that arise when taking a technique and moving it to a new application or environment that the developer of the technique had not anticipated. Second, the author does not always tell the prospective user everything he needs to know. Often this is just due to a lack of documentation, or a set of basic assumptions and background that the author did not realize was even necessary. Lastly, one cannot normally expect the author to emphasize the weak points or problems in the methodology. That is just human nature.

The purpose of this panel and the following set of papers is to discuss a set of techniques available in the open literature, some very new, some that have been around for awhile, and ask for an analysis and evaluation by current users. Each of the panelists is not the author of the methodology but a member of a company that is using the methdology or overseeing a contract on the use of the methodology. Some of the users are old hands at the technique; some are novices. I have asked each of the panelists to prepare a short paper covering a brief description of the technique and an evaluation of the technique in a real environment. Suggested ideas to be included in the paper and the oral presentation are given below.

I. THE TECHNIQUE
The techniques you have been using.
A description of the project you are using it on.
The phase of the project in which it is used.
The phases of the project it affects.
An overview of the technique.
Why you chose it.
The extent to which you are using it.

II. EVALUATION OF THE TECHNIQUE IN A REAL ENVIRONMENT
What have you felt are its good points and how they have shown to be good.
What have you felt are the weak points and why.
How you have adapted it to your environment.
Would you use it again and if so how would you change or have you already changed the way it should be applied.
What would you recommend to someone else applying it.

Certainly lots of techniques could have been covered but there was limited time and space available. The techniques chosen were based partially on my own interest and partially on the availability of people willing to discuss specific techniques. Discussants and topics include:

PSL/PSA—Donald J. Reifer,
SADT—Donn Combelic, ITT Telecommunications
Structured Design—Dr. J. A. Rader, Hughes Aircraft
Jackson's Methodology—Clifford M. Bernstein, EXXON Corporation
Boeing's IPAD Methodology—Susan Voigt, NASA Langley Research Center
Correct Program Design—F. Terry Baker, IBM Corporation

The methodologies deal with various phases of the software development process, from requirements to program development, some emphasizing one specific phase and some covering several phases. PSL/PSA and SADT deal predominantly with the requirements phase. Structured Design, Jackson's Methodology, and Correct Program Design deal with various aspects of design and development. The Boeing IPAD Methodology covers the gamut from requirements to completed product.

What follows is a position paper for each of the panelists. Each is meant to be self contained, complete with references to the appropriate material.

EXPERIENCE WITH PSL/PSA—Donald J. Reifer

INTRODUCTION

This paper briefly describes the experience we have had in using the University of Michigan developed Problem Statement Language (PSL)/Problem Statement Analyzer (PSA).[1] We are using these computer assisted requirements tools to document and analyze operational flight software requirements developed for the Titan 34D segment of the Space Transportation System. The Titan 34D segment is providing real-time guidance, checkout and control requirements for implementation on the Interim Upper Stage. These boost phase requirements are highly time-critical and computation-bound. In addition, they must be documented in accordance to the Software Part I Specification format of MIL-STD-483.[2]

PSL/PSA DESCRIPTION

PSL is a machine processable language for expressing functional and performance requirements for a data system in a rigorous and uniform manner. PSL contains a set of simple declarative statements that allow you to name conceptual objects in a proposed system, describe properties of these objects and display relationships between them.[3] PSA is a software package that accepts PSL statements, analyzes each statically for correct syntax, generates a requirements data base from the input statements, performs consistency and completeness analyses on the data base and generates various kinds of reports and the requirements document. The command language with which users invoke the PSA services is also simple and user oriented.[4]

SELECTION CRITERIA

We selected PSL/PSA from several alternatives for several reasons. First, it is commercially available and supported. There is a commitment to keep the system current and fix errors. Second, it is well documented and good training is available. Third, it is operational on many large computing environments including the IBM 370, CDC 6000/7000 and Honeywell series. Lastly, it is currently being used by several diverse commercial and military users. The multiuser feedback has made the system mature faster than the alternatives. Readers are directed to Davis and Vick's paper[5] for an excellent comparison of PSL/PSA with other techniques.

We elected to use the tool on the Titan 34D application because we felt there was relative low risk inherent in their requirements. Titan 34D requirements are based on flight-proven equations and logic. We felt that the potential benefits justified the risk associated with using a new tool to document known and mature requirements.

EXPERIENCE

The ISDOS staff of the University of Michigan was retained by the Martin Marietta Corporation to install the PSL/PSA system on their CDC 6500 computer and conduct training classes. Installation began in April 1977. Several problems occurred as the package was adapted to the machine. First, the number of pages that resided in core had to be adjusted in order to make the system more efficient in terms of internal charging algorithms. Second, a machine-dependent executive routine that generated control instructions to do routine calling had to be developed. Training was held in the April/May timeframe with general orientation classes, user classes and maintenance classes being held.

The PSL/PSA system was then used to generate operational requirements for the Titan 34D segment of the Space Transportation System. The positive results of its use can be summarized as follows:

1. It forced the user to understand his problem and address it in a disciplined manner.
2. It documented requirements in a uniform manner and eased the task of document maintenance.
3. It assisted in the identification of errors primarily in the areas of incompleteness and inconsistencies.

The negative features of the system are as follows:

1. The system was too general to be used for a specific application. An internal groundrules document had to be developed to tell the user what to do and what not to do (e.g., limited number of attributes, naming conventions for decimal numbers, keyword conventions, etc.)
2. The PSL/PSA systems used large amounts of computer time that were not planned for.
3. The system is oriented towards a business environment and makes assumptions that make it difficult to describe requirements for realtime systems (e.g., concept of counters and interfaces hard to describe, notation is not compatible with scientific symbols, etc.).
4. The system was hard to sell to the users who were engineers. Therefore, changes had to be made to overcome user criticism (e.g., illegal characters like / in ft/sec had to be made legal).
5. The system had to be extended in order to provide I/O tables required by the B5 specification[7] format. A FORTRAN program was developed to search existing files and generate the tables.

The positive benefits resulting from use of PSL/PSA compensated for the negative experiences. Plans have been made to utilize the system to describe software requirements for other military systems.

ACKNOWLEDGMENT

The work reported in this paper was accomplished by the Martin Marietta Corporation under contract to the Air Force System Command's Space and Missile Systems Organization (SAMSO). I would like to thank them for their assistance in the preparation of this report.

REFERENCES

1. Teichroew, D. and E. A. Hershey, III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing," *IEEE Transactions on Software Engineering*, Volume SE-3, Number 1, January 1977, pp. 41-48.
2. MIL-STD-483, Configuration Management Practices for Systems, Equipment, Muinitions and Computer Programs, 31 December 1970.
3. Hershey, E. A., et al., "Problem Statement Language-Language Reference Manual," ISDOS Working Paper No. 68, University of Michigan, 1975.
5. Davis, C. G. and C. R. Vick, "The Software Development System," *IEEE Transactions on Software Engineering*, Volume SE-3, Number 1, January 1977, p. 74.
6. Johnson, L. *JSS Software Systems Engineering: Preliminary Evaluation of CARA*, Logicon Corporation, 1975.
7. MIL-STD-490, Specification Practices, 30 November 1968.

EXPERIENCE WITH SADT—Donn Combelic

BACKGROUND

SADT, Structured Analysis and Design Technique, is a registered trademark of SofTech Inc., Waltham, Mass. ITT has used the "Structured Analysis" part of SofTech's SADT since early 1974. In mid-1975 we began to develop for real-time switching software our own structured design technique, called FP2 for Functions-Processes-Flowgrams-Programs, based upon precepts and syntax of Structured Analysis. Thus the technique we are now using for analysis and design is called SA/FP2.

GENERALITIES

The principal basic ideas of the technique are: determine the "what" before the "how", docomposition from the top down to reveal successive levels of detail, output in the form of diagrams each of which gives a limited amount of detail, each diagram is critiqued in writing by one or more persons other than the author of the diagram, needed information unknown to an author is obtained by interviews with outside experts. Each diagram is comprised of boxes that represent "activities" interconnected by arrows that represent "data" used by an activity for input, output and control. A box plus its arrows constitutes the "context" of that activity—it is that (bounded) context which is decomposed to understand and show more detail in a diagram at the next level.

OVERVIEW OF SA/FP2.

SA/FP2 is carried out in five phases, one of which is concurrent with two of the others. The first phase is that of Structured Analysis (SA); the remainder are those of design, that is, FP2. A brief summary of each of the five phases is given in the following paragraphs.

Structured analysis phase

SA is ideally applied to the total system. However, in most applications we have applied it only to real-time switching software. In such a case, the primary inputs are a list of customer requirements plus functional specifications of the telephone hardware of the system. The output is a Functional Requirements Model (FRM) in the form of a set of activity diagrams many of which are accompanied by a page or two of explanatory text and definitions of terms. The FRM shows what functions the software must contribute in addition to those of the telephone hardware in order to fulfill the customer functional requirements. To the greatest extent possible, software design considerations, such as data layouts, scheduling, priorities, handling of queues, buffers and computer peripherals, are kept out of the FRM. Thus the FRM emphasizes the "what," not the "how".

Transfer phase

This is the first phase of design. Its principal inputs are the FRM and the software design constraints. Typical constraints are: choice and arrangement of computers, computer peripherals, programming language, requirements for traffic handling, engineerability, extensions, maintainability, etc. The outputs of the transfer phase are a high level data layout model and a single level "action group" model. (A software action is defined as a sequence of instructions which, once started, can run to completion without waiting because all needed inputs were available at its start.) For each action group, a convenient set of one or more contiguous activities, along with the data arrows at the boundaries, is selected at an appropriate level of detail from the FRM and transferred (as a single box) to the action group model. The action groups are interconnected as the corresponding sets of activities were interconnected in the FRM. The high level data layout model is developed before and during the transfer procedure. The transfer phase is complete when all activities of the FRM have been accounted for in the action group model.

Action group decomposition phase

All action groups are decomposed to the level of individual actions. The output is a set of activity diagrams where each box at the lowest level of detail represents an individual action. An additional output is further detail of the data layout. During this and the preceding phase the decomposition rules are the same as for the SA phase, but the SA syntax is augmented to handle action starts and completions.

Flowgram phase

Each action is decomposed, according to its implicit control flow sequence, down to the level of individual routines, each of which appears as a separate box on the lowest level diagrams. The control flow is shown on each diagram in a special syntax, hence the name "flowgram." The output of this phase is a "flowgram model" for each action. The previous syntax is augmented to handle control flow. It turns out that when the control flow sequence and the individual routines are coded the resulting set is a structured program. Thus there is a structured program for each action.

Coordinator phase

The coordinator is that software which, among other things, starts all actions and to which all actions return upon completion. It thus includes the functions of scheduling, handling of queues and management of memory. It is convenient to include within the coordinator the treatment of interrupts and the handling of telephone and computer peripherals. It is interesting to note that none of these functions relate directly to the customer's basic functional requirements, rather they all depend on the nature of the system. The functional requirements for the coordinator begin to emerge as early as the SA phase, become more clear by the end of the transfer phase, but cannot be known completely until the action group decomposition phase is finished. By that time the coordinator can be completely specified and designed. Note that the techniques described for the preceding phases can be applied to the analysis and design of the coordinator.

EARLY EXPERIENCE—1974-1976

Development of the FP2 design methodology reached the point where it could be used in practice only at the beginning of 1977. Thus all our prior experience was limited to Structured Analysis as taught by SofTech and refined by ITT and SofTech together. We adopted SA in early 1974 for two main reasons. First, it provided a disciplined way of understanding requirements in detail before starting design. Second, it offered a method which promoted teamwork. The latter was a particularly difficult problem on some of our projects in Europe where a team would consists of members with widely varying experience from up to eight different ITT companies speaking six different languages. Of the approximately twenty ITT projects where SA has been used, all but one are in Europe.

Strong and weak points

A partial list, derived from our early experience with SA, is as follows:

- SA estimated to decrease overall software development cost by at least 20 percent and significantly improve software quality—estimated 2 to 10 times less bugs found during integration testing, varies with project.
- Very definitely promoted teamwork.
- Hard to think all the time in purely functional terms.
- The written comments (by other than the author) required for each diagram resulted in continual review, in effect "walkthroughs." (Note : Commentators should normally be other authors in the same team.)
- Interviews of outside experts proved efficient method of obtaining specialized information.
- Forced making high level decisions early, thus providing a sound basis for later lower level decisions.
- Encouraged agreement on requirements before start of design.
- Lack of follow-through on design methodology (later overcome by FP2) was bothersome.
- Permitted non-software people to understand the contributions of software functions to system operation.
- Much more useful information per sheet (diagram) than with documents in prose.
- Provided easy way of measuring progress during analysis.
- SA excellent for many applications other than real-time switching software, but space does not permit elaboration.

Some mistakes made and lessons learned

A partial list follows:

- Method was oversold in the beginning as a panacea.
- Proper use of SA requires a fundamental change in mental outlook—difficulties of achieving such a change were underestimated.
- Mere "training" is inadequate—"education" is required.
- Potential authors must be selected on the basis of intelligence and willingness to try a new way rather than purely on experience.
- Constructive critics of the method are to be cherished, but destructive critics must be eliminated from the SA group as soon as they surface.
- Method takes much more time before design starts than previous methods. Inevitable if requirements to be thor-

oughly understood and agreed before design, but causes impatience in some participants and management.

- Need one "friendly" group to try method first. Afterwards it's better to "offer" the method to other groups than to try to "sell" it—hard sell doesn't work.
- "Discipline" in use of method is very important: syntax, conventions, rules, author-commentator cycle, completion of a level of decomposition before going to next, each diagram must increase understanding, etc.
- Large amounts of paper generated. Project librarian must be assigned, as recommended by SofTech.
- Formal training is lengthy—two to three weeks full time—but necessary.
- Potential authors must have a project in mind during training and be assigned full-time to it immediately thereafter.
- Follow on assistance by a trained "monitor" is obligatory during initial application of method.
- Monitor must confine his attention to proper use of the method rather than become involved in the substance of the analysis.
- Training plus monitoring by SofTech is costly: 20 to 40 thousand dollars per course for up to ten authors, but worth it.
- Ideal team during SA phase for our type of large switching projects seems to be two or four persons each from systems, hardware and software.

Recent experience—1977

It is our estimate that about two-thirds of the value of SA/FP2 is in the SA part. Nevertheless, the availability of a follow on design method, in our case FP2, makes the application of SA easier because the SA authors are more ready to defer design considerations to the FP2 phases.

The development and acceptance of FP2 has proved extremely difficult and its success has not yet been fully demonstrated. We wanted to use the same basic precepts and syntax as in SA so that the designers, some of whom will have also participated in the analysis by SA, would not have to learn a new syntax and set of principles—we wanted a sort of "continuous" method, starting with a list of requirements and constraints and ending up with detailed coding specifications. This "continuity" and similarity of syntax is important to software maintenance personnel and will assist comprehension by interested customers.

This section concentrates on our 1977 experience with FP2; subsequent experience will be covered during the presentation at NCC 78.

- Underestimated importance of providing detailed guidelines for carrying out the Transfer phase.
- Initially called the output of the Transfer phase, the "process model." "Process" has many meanings and caused great confusion. The neutral phrase "action group" conveys the intent without confusion.
- Software design still requires great skill, but FP2 permits easy comprehension after key design decisions have been made.
- FP2 criticized for not rendering high level software design "semi-automatic" or at least "semi-algorithmic."
- Direct coding from flowgrams is in most cases straightforward and can be done by programmers other than the designers.
- The fallout of a "structured program" for each action has proved very appealing.
- Test plans can be developed throughout FP2 in increasing detail and related directly to diagrams.
- Much debugging is done by reference to flowgrams instead of listings.

REFERENCES

1. *An Introduction to SADT*, SofTech, Inc., Waltham, MA, document 9022-78, Feb. 1976.
2. Ross, D. T. and K. E. Schoman, Jr. "Structured Analysis for Requirements Definition," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January 1977, pp. 6-15.

EXPERIENCE WITH AN APPLICATION OF STRUCTURED DESIGN—J. A. Rader

INTRODUCTION

The application of structured design to a 20 man year project which generated 100,000 lines of code is described. Included are a description of the project, productivity figures, and a discussion of strengths and weaknesses of the technique as practiced on the project.

THE APPLICATION

Introduction

The Computer Aided Design (CAD) Department in the Hughes Aerospace Groups contains about sixty employees. The department provides Computer-Aided Design/Test/Manufacturing software and services; it operates and supports a DEC system 10 computing facility; and it operates and maintains a Gerber photoplotter and several digitizers.

Most of the software is data manipulative in nature—files are read; fields are extracted from records and massaged; arrays are built, operated on and sorted; reports are generated; and data bases are accessed and updated. The primary language has been and continues to be FORTRAN. In addition, there is an extensive library of FORTRAN-callable assembly language routines to perform bit and character manipulations as well as other special functions. Where very heavy CPU utilization is expected, assembly language is also sometimes employed.

Conversion decision

Several years ago, a corporate decision was made to phase out the Honeywell G635, the computer on which the CAD System at that time ran. Among the many alternatives considered for rehosting the CAD System, the one finally chosen was to purchase a DEC system-10 and convert the CAD system to run on that computer. The primary reason for selecting the DEC-10 was a proven time-sharing capability.

The conversion from the G635 to the DEC-10 was a conversion only from the standpoint of function.

Existing programs were inventoried and for each it was decided which would be converted essentially as is, which would be modified, and which would be discarded.

Goals and advanced plan

As we started to plan, we recognized that it was very important to firmly establish our goals, and to determine very specific milestones. Thus we would be able to measure our progress and to report on it to our management, who was picking up the tab.

Succinctly stated, the major goals were: (1) to create a unified system tied together by a central data base; (2) to create software that was reliable and maintainable; (3) to provide a user interface that was easy to use and that was consistent across all software; and (4) to proceed in a manner that allowed us to measure how well we were meeting schedules.

In late 1973, an advanced planning and development activity was formed. The advanced planning group was to define the overall structure of the system, and to specify the procedures to be followed in specifying and implementing the system.

In November of 1973, the author attended a 6-day in-plant seminar on structured design. This seminar was taught by Larry Constantine and proved to be of immeasurable value. The value arose not from revolutionary concepts but rather from a well reasoned and coherent discussion of the relevant concepts. The ultimate result of attendance was the generation and documentation of a methodology for practicing structured design in our organization. This methodology is described in the next section.

Outputs of advance group

In the approximately 18 months of its existence, the advance group produced 4 basic outputs. First, it produced a system concept. Second, it defined the standards and procedures to be followed in rewriting the system. These were documented in a Standards and Procedures Manual issued to all programmers. Third, it defined most of the applications support software. Fourth, it provided a test vehicle for the standards and procedures defined, and provided the first productivity figures for the methodology. These figures were used in estimating the effort required to implement the main body of CAD software.

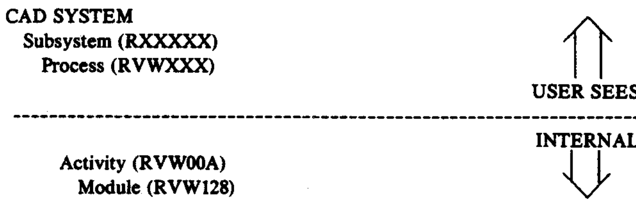

Figure 1—Software hierarchy

STRUCTURED DESIGN

The following hierarchy, illustrated in Figure 1, is used to identify levels of CAD software: system, subsystem, process, activity, and module. System means the whole CAD System. A subsystem is a major functional area. Examples of subsystems are routing, digital test and simulation, design capture, etc. A process performs an identifiable user function which may be simple or complex. The process level is the lowest level visible to the user.

A process consists of one or more activities, where an activity is an executable core image or job step. A module, which is the lowest level, is a single FORTRAN or assembly language subroutine. Our FORTRAN modules contain an average of 35-45 statements. Assembly language modules have an average length of 50-60 statements.

Each activity and module is uniquely identified by a six character code. Examples are shown in parentheses in Figure 1. Thus module RVW128 would be module 128 in the View (VW) Process of the Routing (R) Subsystem.

Structured design, a la Constantine, is applied either at the process or activity level, depending on the complexity of the process and constituent activities. Design documentation always includes a structure chart (hand drawn, as generated in the design process) and a completed module description form (MDF) for each and every module. A module description completely defines the *function* of a module so that it can be coded from this documentation alone, with perhaps reference to appropriate data structure documentation.

For an easily grasped function, e.g., the binary search of a single precision array, a description of the calling sequence may be adequate documentation. However, it has been our experience that many modules require either flow charts or pseudocode for unambiguous documentation. This is true even though the emphasis is on the function not the detailed coding of the module. Typically, this situation obtains for the higher level modules in a structure chart. At that level it is frequently the case that the function of the module and the flow of control within the module are not really separable.

RESULTS OF APPLYING STRUCTURED DESIGN

Productivity figures

The advanced planning activity implemented 336 modules of applications support software. These modules contained

	Advance Group (336 Modules)	Main Development Effort (2200 Modules)
Man Hours Per Module		
Specification	3.3	5
Structured Design	3.0	5.4
Coding and Integration	8.8	9.8
Total	15	20

Figure 2—Manpower statistics

11,104 lines of code, an average of 33 lines/module. There were 180 FORTRAN modules (average length 25 lines), and 156 assembly language modules (average length 42 lines).

The manpower figures were specification and analysis, 28 MW (man weeks); structured/implementation design, 25 MW; and code and integration, 74 MW. Figure 2 displays the corresponding figures per module.

The largest activity in this collection of software contained 117 modules (3551 lines), of which 105 (2936 lines) were FORTRAN. The distribution of calendar time for this program was: specification and analysis, 26 weeks; structured design, 6 weeks; and coding and integration, 20 weeks. Thus, although specification and analysis only accounted for 22 percent of the manpower it consumed 50 percent of the calendar time.

Column 3 of Figure 2 summarizes the figures for the bulk of the CAD software. This data represents approximately 18 man years of effort including supervision time. Accounted for are 27 processes, 53 activities, and over 2200 modules.

Problems encountered

Although our experience with structured programming has been strongly positive, we did encounter some problems. Moreover, looking back, we see areas where improvement is needed.

The biggest need for improvement has to be in the area of specification. Once a good specification has been generated things become very manageable. However, we have found it extremely difficult to write a specification which on the one hand a user can read and understand, and which on the other hand defines things well enough to allow design to begin in earnest.

A second problem area is related to one of the design goals. From the start it was impressed upon the programmers that they were to put design before efficiency. As a result a couple of activities were implemented which were very much more expensive to execute than they should have been. These subsequently, had to be modified for efficiency.

In most cases this efficiency problem might well have been avoided had we more strictly followed one of our own published procedures. We did not require each module to be reviewed by another programmer as we said we would. The excuse tends to be "we just don't have the time", and is used by programmers and supervision alike. In the face of tight schedules, this excuse is not easily dismissed. This problem will doubtless be struggled with for some time to come.

A final problem is the difficulty in training enough programmers to be good designers. The training problem is particularly troublesome because there is no way to give years of experience, and the attendant design maturity, to even a highly capable junior individual. This is important because the structured design of a large process requires the judgment to make numerous decisions, which involve tradeoffs between strict adherence to structured design principles, on the one hand, and effective use of human and machine resources on the other. The best solution is to employ the most qualified designers on critical designs, and to use less critical designs for training. But when schedule constraints force many important designs to overlap, less desirable compromises have to be made.

Summary of benefits

Although we did not have any solid prior productivity data, we definitely feel that structured design has improved productivity. This, however, was not a goal in our adopting structured design, but has been just a happy side-effect. What was anticipated were increased reliability, increased maintainability, and increased visibility.

There is no doubt that we have achieved increased visibility. Moreover subsequent experience with modifying structured programs convinces us that increased reliability and maintainability have been achieved.

It was found that structured design allowed us to make very good use of personnel. We have been able to really load up an activity in the module coding and checkout phases without introducing confusion. It has also been easy to quickly move a programmer from one activity to another, with little loss in effectivity. Moreover, we have gotten excellent productivity from beginning programmers.

Only mild reluctance to adopt structured design techniques was manifested by the staff. Junior personnel adopted easily with no apparent "loss of individuality" response. There was some initial thrashing with senior personnel as we all strove to understand the implications and tradeoffs of modularity. For instance, not everyone accepted at first that structured design was indeed distinct from what they were already doing.

A final word on interpreting our experiences with structured design. We feel that our experience is unique to our application and our environment. A different application in a different environment might yield better or poorer results. Nonetheless, we are confident that, for most applications, structured design will yield more reliable and maintainable systems, while providing good visibility of the design and implementation processes.

BIBLIOGRAPHY

1. Constantine, L. L., and E. Yourdon, *Structured Design*, Yourdon, Inc., New York, (February 1976).

2. Rader, J. A., *Structured Design—A Case History*, Infotech State of the Art Report on the Practice of Structured Design, London, (1976).
3. Stevens, W. P., G. J. Myers, and L. L. Constantine, Structured Design, *IBM Systems Journal*, No. 2, 1974, pp. 115-139.

EXPERIENCE WITH EXXON'S IMPLEMENTATION OF THE JACKSON PROGRAM DESIGN METHOD —C. M. Bernstein

In 1973, Exxon's Mathematics, Computers and Systems Department conducted an evaluation of the new technologies of program development. The project was motivated by the increasing manpower cost for software development and maintenance and the increasing business vulnerability to software failure. We concluded that a program's structure is the key to its effective development, enhancement, execution and support. Without a program's structure we could not reliably construct the program in parts, put the parts together such that their interactions would be predictable, and have the whole structure achieve its specified purpose.

Michael Jackson, now of Michael Jackson Systems Ltd., was then an ACM lecturer. We found that he had a teachable method for the logical design of structured programs and a precise notation to express them. In addition, his method was effective at attacking practical programming problems where the designer is not free to define input and output formats and user interfaces. Michael taught three program design courses at our Florham Park, New Jersey offices in the winter of 1973. The courses were well received and, together with Michael's consulting on specific applications, provided the knowledge we required. We adapted Jackson's Methdology to our environment and renamed it Program Structure Technology, PST.

It is important to remember that PST does not address the systems design process. PST is not concerned with defining the organization or contents of files, specification of input and output formats or transactions, designing data bases, or defining required processing. PST is concerned with the work of designing and implementing the program which meets those specifications. PST incorporates the technologies of top down design, structured programming, top down development and test, and structured walk-thrus.

We conducted our first in-house PST course in August of 1974 and aggressively fostered the assimilation of PST over the following three years. Over 1,000 programmer/analysts have been taught PST. All Exxon regions can now provide their own training and support. This includes training for supervisors, consulting, and the support of standards. PST is being used in both large and small installations to develop batch commercial and interactive data base applications, as well as minicomputer, process control and program product software.

The Jackson Program Design Method defines hierarchical program structures whose components are dissected. There are four basic component types:

Elementary, which are not dissected
Sequence, whose parts are executed once each, in order
Selection, one of whose parts is executed, the choice part depending on a condition
Iteration, which has only one part which is executed zero or more times

The four component types have exact analogies in data usage structures for files, records and in internal data. The program's structure is designed to correspond to the usage of the data to be processed. Alternative usages of data can be imposed on a given set of data and the alternative program structures evaluated. Each operation carried out by the program appears in an appropriate component. For example, "INITIALIZE CUSTOMER TOTAL" should appear in a component which happens once per customer. If no appropriate component exists, the program structure is deficient. Where there is a conflict between data usages to be processed by one program, the program is designed as if it were two or more separate programs. The separate programs are subsequently combined by the technique of program inversion. Where a data usage cannot be handled simply, more elaborate forms of iteration and selection must be used.

The major benefits of the Jackson Methodology are as follows:

Reduced program complexity
Eliminated logic errors at design instead of debugging in the testing stage or later
Identified sensitive points in the problem specification
Easily maintained programs
Effective program documentation as a by-product of design
Step-by-step methodology has enabled effective use of software to support the process

Four PST projects of varying size (.5, 1, 5.1, and 25 work years of effort) were evaluated and productivity was found to be above 7000 lines per work year in every case. This compares favorably with the New York Times Archives project's 9000/WY and the "Industry Averages" of 2000-4000 lines/WY. In the case where we attempted to measure program quality, we found less than one error per program during the first six months of production.

We have done little to the Jackson Methodology to adapt it to the Exxon environment other than minor changes in terminology and emphasis. We have complemented it with our own material on structured programming in PL/I, top down development and test, and structured walk-thrus. We have also employed different pedagogical techniques to enable programmers who are not experienced instructors to teach the PST course.

The Jackson Methodology has been extremely successful in Exxon and is now virtually a standard throughout the world. I recommend not underestimating the difficulty of teaching programmers "how to program properly" and develop an assimilation plan and a skilled staff to accomplish it.

REFERENCES

1. Jackson, M. A., *Principles of Program Design* Academic Press, 1975.
2. Pinsonneault, L. L., Course Overview. *Program Structure Technology Course Manual,* March 1975.

INITIAL EXPERIENCE WITH A METHODOLOGY FOR CORRECT PROGRAM DESIGN—F. T. Baker

In the forthcoming book, *Structured Programming: Theory and Practice,*[1] the authors describe three techniques which have been incorporated into a Methodology for achieving correct designs. These are:

1. A view of program correctness as a demonstration of a correspondence between the function of a program design (i.e., the set of ordered pairs corresponding to input states and output states) and the function required by its specification. This approach, when used with stepwise refinement, permits selective and incremental correctness proofs to be carried out, since it incorporates a procedure for verifying the correctness of the expansion of a specification into any one of a basic set of control structures. (The expansion of a specification at any level into a program design can thus be verified, contingent on the correctness of lower-level specifications and their expansions.) Furthermore, proofs can be carried out with varying levels of rigor, ranging from a set of questions the designer may use to validate an expansion to a formal demonstration recorded in a precise manner.
2. A method for incorporating specifications into program designs to support correctness demonstrations when desirable. Each specification (either initial, or those generated in the expansion process) is retained as a comment (logical commentary) directly associated with the control structure which refines it.
3. A design language (Process Design Language) to assist in the design process and to record the history of a design. PDL includes a standard "outer syntax" of essential control and data structures, and encourages development of an "inner syntax" appropriate to each design environment.

Figure 1 is an example of a design for a program which is to save the maximum value occurring in an input sequence. In that design, each of the paired brackets encloses logical commentary which is a functional specification for the control structure which follows it (sequence, ifthenelse or while do in this case). For each of the structures the appropriate proof procedure can be carried out to demonstrate the correspondence between its specification function and its program function. Furthermore, this can be done prior to the completion of the expansion process, or even on selected portions of the design.

```
[m:=maximum(inputseq)]
do
  [(inputseq=empty→m:=undefined |
        inputseq≠empty→m:=next(inputseq)]
  if
    inputseq=empty
  then
    m:=undefined
  else
    m:=next(inputseq)
  fi
  [m:=maximum(m,remainder of inputseq)]
  while
    inputseq≠empty
  do [m:=maximum(m,next(input)]
    temp:=next(input)
    [m:=maximum (m,temp)]
      if
        temp>m
      then
        m:=temp
      fi
   od
 od
```

Figure 1

The methodology is primarily aimed at the detailed design of a program. It covers the period between the formulation of an effective system design, and the translation from the design language into an implementation language. It was developed to introduce more precision into the design process and to encourage more consistent expression of designs. Whether or not formal correctness demonstrations are carried out, the stress on viewing programs as functions, developed from specifications through a rigorous refinement process, should help achieve the goal.

Experience to date suggests that the methodology is capable of being practiced in the application development environment. We believe that the control structures inherent in PDL are sufficient to support all levels of the design process, from system and module specification down to precise algorithms. The invention and use of logical commentary direct attention to specification and program functions, as they were intended to do. In particular, they encourage the designer to specify and deal with boundary conditions and anomalies which frequently are poorly attended to and which sometimes lead to difficult-to-find errors. Finally, the view that each progrm should be designed as if it is to be proved correct, means that even if correctness demonstrations are not formally carried out, the program has a greater likelihood of properly embodying its specification.

Experience to date also indicates several areas where more work is needed. The nature of the expansion process, and the desire to record the design history, mean that much copying is done as the design is developed. An interactive support tool appears useful to assist designers in this expansion and recording. The data structures in PDL (stacks, queues, sequences and sets) are useful ones, but better proof techniques to validate operations with them must be developed. There is a notational problem inherent in specifying functions precisely, particularly in nonmathematical environments, which must be solved through a combination of inventing better notations and abstracting operations. Finally, the ability to demonstrate correctness does not mean that it is appropriate to do so in all cases; better guidelines for applying the varying degrees of rigor possible in the methodology must be developed.

REFERENCE

1. Linger, R. C., H. D. Mills, B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, 1978, to appear.

EXPERIENCE WITH THE IPAD SOFTWARE DEVELOPMENT METHODOLOGY—Susan Voigt

NASA is sponsoring the development of a computer-aided design system for use by the aerospace industry. The system, denoted IPAD, is being designed and implemented by Boeing Commercial Airplane Company and Boeing Computer Services. IPAD is a software system to enhance the computer complexes of aerospace companies to improve speed, efficiency and reliability of the design process for complex aerospace vehicles. The contract calls for application of an effective software engineering approach to minimize programming and software design errors, as well as to produce highly portable software.

THE TECHNIQUE

NASA established general guidelines for phases of the development and release of software in stages, to allow early user testing and experience. The development phases are:

1. Definition of the Problem (namely the Aerospace design process)
2. Requirements Definition (integrated information processing and functional requirements)
3. Development of IPAD System Specifications
4. Preliminary Design
5. Detailed Design, Code, and Test for each incremental release
6. Acceptance Test and Demonstration with sample problems for each release
7. User Training and feedback
8. Software maintenance during remainder of development contract

Several plans and specific documents were called for in order to encourage a systematic approach and a well documented product:

1. Management and Technical Plans to describe general approach
2. Configuration Control Plan to control and track all changes to requirements, design, code and documents
3. User Involement Plan to insure the sytem developed is satisfactory to the users
4. Test Plans to establish systematic procedures for development and acceptance testing
5. Software Standards Handbook
6. Requirements and Preliminary Design Documents
7. Preliminary User's Manual written during design so early user feedback can be obtained
8. User and Demonstration Manuals
9. Installation and Maintenance Manuals

As the basis for the IPAD software engineering methodology, the Boeing IPAD Development Team selected the Boeing Computer Services "Systematic Software Development and Maintenance (SSDM)" approach. SSDM is basically a set of general guidelines for all phases of the software life cycle, and it corresponds well with the NASA requirements.

An Industry Technical Advisory Board (ITAB) was established at the start of the contract to closely involve the prospective user community. They have helped review and critique the requirements definition and software design phases. Subsequently, they will have the opportunity to install and test the software at their own computing facilities.

EXPERIENCE AND EVALUATION

At the time of this writing, the development is in phase 4, Preliminary Design. The techniques used to date and an assessment of their usefulness in the software development process is described below.

In phase 1, a reference aircraft design process was documented in flow diagrams indicating activities and decision points, with accompanying discussions. Also communications between various disciplinary groups participating in an aircraft design project were diagrammed to illustrate the complex network of interfaces. Separate volumes were written to document the interactions between designers and the manufacturing organization and the activities in managing a product development. These three volumes written by engineers representing potential users defined the problem to be addressed with the IPAD system.

The requirements were defined by the engineers in phase 2. The BCS technique SAMM (Systematic Activity Modeling Method) (Reference 1) was used to chart the inputs and outputs (description and quantity) for each activity in the flow-charts of phase 1. The user's view of his requirements for computer-aided support in the aerospace design process also was documented.

The results of phases 1 and 2 represent a very thorough definition of the aerospace design process and CAD users' needs. These were well-received by the user community and have provided guidelines for their own analyses within their respective companies. The flow diagrams of the design process and the SAMM charts of the data flow are well correlated and provide a very systematic look at the problem.

Phase 3 was done by the software team, assisted by the engineers. They developed a concise set of IPAD requirements based upon the engineering documents and NASA requirements for the IPAD system. A formal analysis checked that each requirement was complete, correct, unambiguous, precise, consistent, relevant, testable, traceable, free of unwarranted detail, and manageable. The engineering team developed criteria for acceptance testing for each requirement. Each test criteria was summarized in a paragraph which accompanies the requirement statement in the IPAD Requirements Document. The requirements were reviewed carefully by both NASA and ITAB, with considerable feedback and revision resulting. A set of IPAD system specifications were not produced, per se; the IPAD Requirements became the baseline for further development.

The formal analysis of requirements was not successful. Ambiguities, inconsistencies, and redundancies were very difficult to eliminate, especially between similar requirements. These arose from using secondary sources in developing the IPAD requirements and giving them weight equal to the primary source, the engineering definition of the problem. A satisfactory set of requirements was obtained through careful reorganization and joint review by software and engineering team members and NASA. The inclusion of the summary for acceptance test was very helpful in clarifying the intent of a requirement, as well as setting the stage for later testing. It also forced the engineering and software teams to collaborate, and is highly recommended for future projects.

In Preliminary Design, phase 4, a user interface model was developed using state transition diagrams (Reference 2). This included a set of user functions correlated to the IPAD requirements. These state diagrams have been used to walk through "user scenarios" to illustrate the functioning of IPAD from a user point of view in performing a specified set of tasks. The other major components of IPAD are: the executive, the information processor, and the other systems interface and are currently undergoing design by separate subteams. Coordination among the various subteams in producing an integrated design has been difficult.

While the approach used has been successful in achieving a good set of requirements, IPAD is not yet developed and the design methodology is unproven. The basic concepts appear to be an effective approach and further assessments can be made when the software is developed.

REFERENCES

1. IPAD Document D6-IPAD-70012-D, "Integrated Information Processing Requirements," June 24, 1977 Boeing Commercial Airplane Co., under contract NAS1-14700. (Data Modeling Method, SAMM)
2. Parnas, David L., "On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System," *Proceedings, 24th Conference ACM*, 1969, pp. 379-385.