Behavior Research Methods & Instrumentation
1981, Vol. 13 (2), 163-171

Structured system design: Analysis, design, and construction

GARY B. FORBACH
Washburn University, Topeka, Kansas 66621

A simple state-of-the-art systems/program design tool is described, demonstrated, and evaluated. The tool, the Warnier-Orr diagram, is derived from Boolean algebra and set theory and is a major analysis, synthesis, and documentation aid in Structured System Design (SSD), a current structured programming methodology. Proponents of SSD claim it provides a conceptual framework that facilitates writing programs that usually run the first time. Skeptics typically agree (eventually). Several sources of SSD information (articles, books, technical reports, etc.) are identified.

Associated with psychologists' and other non-computer professionals' extensive and extremely varied uses of computers are several general problems that we have inherited from the computer programming industry: (1) cumbersome, sometimes imprecise design tools (e.g., flowcharts, decision tables, pseudolanguages, Nassi-Schneiderman charts, etc.); (2) program documentation techniques that have traditionally failed to produce simple, accurate, easily updated program documentation; (3) nonsystematic program maintenance procedures; and (4) nonsystematic program testing procedures (i.e., while it is possible to demonstrate that a program produces correct output, it is virtually impossible to prove that a very large program could never produce an unanticipated result).

The computer programming industry has dealt with these problems and some related problems by attempting to formulate a comprehensive "theory" (methodology) of computer program development that systematically integrates all the necessary and sufficient behaviors involved in producing a correct computer program. As might be expected, no single methodology has emerged as "the theory" from the last 25 years of theoretical work in software development. Instead, a number of methodologies have been developed in parallel, often sharing one or more basic concepts. These basic concepts represent many program design and implementation strategies that have been shown to be extremely valuable when put into practice. Most are theoretically based, some are simply examples of remarkable insight. Some, possibly most, are at least somewhat familiar to anyone who programs regularly and include the following: restricted choice of program control structures, elimination of GOTOs, modularity of program segments, reduction of complexity by breaking programs into smaller pieces, functional analysis of a problem, top-down decomposition of problems, hierarchical organization of problem components, data flow analysis, and data-oriented design (DeMarco, 1978; Dijkstra, 1976; Jackson, 1975; Mills, 1971; Orr, 1977; Warnier, 1976; Yourdon & Constantine, 1978; Ross, Note 1).

Perhaps the most important common feature of the methodologies cited above is their emphasis on some set of systematic recursive problem decomposition procedures that are directed toward breaking down complex problems into successively more manageable units. The rigorous application of these procedures represents a disciplined attempt to remove computer programming from the realm of mystical, inspirational, erratic artistry and establish it firmly in the domain of a reliable, consistent, theory-based science. Canning (1979), in a thorough review and analysis of current software development methodologies, has noted that the term "structured" is often applied to this type of systematic approach to software development.

A second characteristic of several current structured methodologies is that specific program development tools and procedures are used as the basic building blocks of a "theory" that is oriented toward efficient and correct systems development. This has come about due to the exponentially increasing complexity of modern computer systems and the need for managerial tools that make it feasible to design, implement, and maintain these extremely complex systems. Thus, included in many methodologies that are marketed or disseminated as systems design methods are also the program design and development techniques. A summary and comparison of three popular, currently available program design methodologies (two of which are

I wish to thank Ken Orr for his willingness to discuss with me many of these ideas and for inviting me to participate in the Structured Systems Design User's Conference/5, October 6-8, 1980, which enabled me to gather a great deal of first-hand information from SSD users and provided me the opportunity to discuss with J. D. Warnier his ideas on the relationship between systems design and human cognitive function. Also, Ken's staff, especially Karen Brown, Marlene Orr, and Bob Otey, were especially helpful in providing technical reports, preprints, and prepublication manuscripts that would not otherwise have been available. Reprint requests should be sent to Gary B. Forbach, Department of Psychology, Washburn University, Topeka, Kansas 66621.

0005-7878/81/020163-09$01.15/0

also systems design methodologies) are presented by McNurlin (1979).

A third important feature of structured approaches is that, in general, they work. A large body of evaluation data from the computer software industry indicates that conscientious application of structured programming techniques results in fewer program errors, less maintenance, and generally improved efficiency in the use of computing resources. However, despite the potential benefits, those of us who are not computer professionals may find it impractical to pursue an in-depth knowledge of a structured approach due to time and/or financial constraints. For example, it takes about 2 years to determine if implementation of a structured methodology is successful in a large company (McNurlin, 1979), and the training will cost $200-$700 per person (Stevens, Note 2). The purpose of the present paper is to describe, demonstrate, and evaluate a technique called Structured System Design (SSD), one of several currently available methodologies. Use of SSD could potentially aid in the solution of all four of the general problems listed above. Yet it represents a practical, cost-effective, and realistic approach for computer users who are primarily psychologists, not professional systems analysts/programmers. The primary focus of this discussion of SSD will be on program development techniques, although SSD is also a systems development methodology.

INTRODUCTION TO STRUCTURED SYSTEM DESIGN

Overview

The SSD methodology is the cumulative, continually evolving product of approximately 15 years of work, first by Jean-Dominique Warnier and his colleagues in France, and more recently by Kenneth T. Orr (and his colleagues) of Langston-Kitch & Associates in the United States. It is a method of logical analysis, design, and construction derived from Boolean algebra and set theory. SSD shares some features of other structured approaches (e.g., hierarchical analysis, top-down decomposition, restricted control structures, and elimination of GOTOs), but it also has several unique features that warrant close examination. The major assertion of SSD is that the process of program (system) development has two logically discrete stages, design and construction. Therefore, the act of coding a program is not synonymous with designing a program, and coding may in fact actually induce errors, especially if the programmer alternately designs and codes small sections of a program or system. Such an overlap of the design and construction phases must be strictly avoided.

A second SSD assertion is that design consists of two discrete segments, the hierarchical analysis of a problem and the synthesis of a logical process that will provide a correct problem solution. Hierarchical analysis is a "top-down" approach requiring successive decomposition of a problem into subproblems (processes) derived from set-subset relationships intrinsic in the problem structure, until it is no longer possible to subdivide a process. Each of the smallest subdivisions (called functions) is a discrete action. Hierarchical analysis is always systematically implemented to carefully preserve and demonstrate explicitly all set-subset relationships between functions. Synthesis consists of developing a structured process. A structured process is one that explicitly organizes functions in one of three logical control structures, sequence, repetition, and alternation (see Figure 1). (It has been shown by Böhm and Jacopini, 1966, that any program can be developed using some combination of these three logical control structures.) Synthesis is a "bottom-up" process that starts with desired outputs and works backward, putting together a structured process that provides the desired outputs. The complete structured process thus consists of a hierarchical organization of functions linked by one of three logical control structures and, as a whole, represents the necessary and sufficient actions for a specified problem solution. The final stage of the process is construction, that is, converting in the specified order all derived functions to statements that a given computer accepts. Figure 2 summarizes SSD program development components and shows implementation sequence.

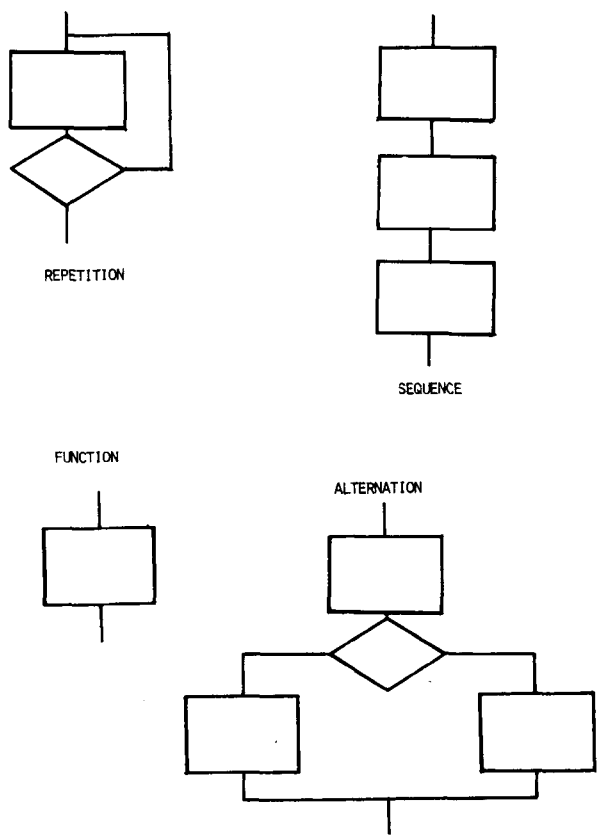


Figure 1. Basic building blocks of a structured process.

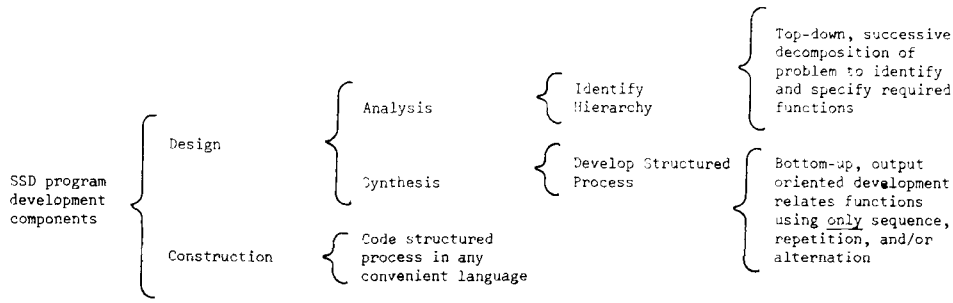


Figure 2. Summary of SSD program development components and implementation order.

SSD Notation

Set theory and Boolean algebra provide the formal, mathematical basis of SSD and help formalize the identification of hierarchical relationships. To facilitate the hierarchical analysis process, a powerful yet simple notational scheme called the Warnier-Orr diagram has been developed. Figure 2 is a Warnier-Orr diagram purposely introduced before any explanation of symbols to illustrate the ease of understanding the diagram. When pressed for an explanation of the diagram, a naive reader will usually read from left to right and from top to bottom (correctly). It is often apparent that some understanding of the hierarchy is quickly grasped, since the reader frequently compares the diagram with a more familiar method of representing hierarchy such as an outline or a tree diagram. In addition to the open brace[1] that separates levels in the hierarchy, four logical operators are employed to indicate the possible relationships between functions and/or structured processes. They include exclusivity, concurrency,[2] sequence, and negation. Finally a set of parentheses below a process or function name indicates the number of times the process/function is to be performed. Table 1 summarizes the notational symbols and describes and defines each.

Using SSD

Higgins (1979, chap. 3) presents several examples that show how to use Warnier-Orr diagrams to design and develop simple "programs." Figure 3 is a modification and extension of one of Higgins' examples, a trivial but informative use of a Warnier-Orr diagram to show how to change a flat tire. Examination of this diagram reveals that a great deal of information about a complex process can be broken down into small, understandable pieces, yielding a better understanding of the total process. And nontrivial problems may be dealt with using the same approach. However, nontrivial computing problems usually require something not in the above example, output data. One of the most powerful features of SSD is its data driven methodology; SSD is output oriented. This is not a new design concept, but an early design approach that was replaced by "better" approaches due to both an inherent inflexibility once the desired outputs are set and the difficulty of communication with users that is accurate enough to indicate what output is actually wanted. However, Orr (in press) has suggested that output-oriented systems are complete, minimal, efficient, understandable, and appropriate (assuming outputs are defined correctly)—strengths that are more than enough to outweigh the weaknesses. Thus the cardinal rule in SSD is "design processes backward." This means it is necessary to start with the finished product (the desired program output) and work backward through all the actions that led to the desired correct output. The result of strict adherence to this strategy will be a program/system

Table 1
Summary of SSD Notation

Symbol	Name	Description
	Connectives	
⊕	Exclusive "OR"	Do either one or the other, not both.
+	Inclusive "OR" (concurrency)	Do either one or the other or both.
Blank (or ⊙)	"AND" (sequence)	Do first one, then the other, in order.
"process"	"NOT" (logical negation)	Do the process that is logically the complement of "process."
	Number-of-Times Indicators	
(0,1)	(Selection)	Do zero or one times.
(0,n)	(DO WHILE)	Do zero to "n" times.
(1,n)	(variable DO UNTIL)	Do one to "n" times.
(c)	(constant DO UNTIL)	Do "c" times.
Blank (or 1)	(perform)	Do one time.

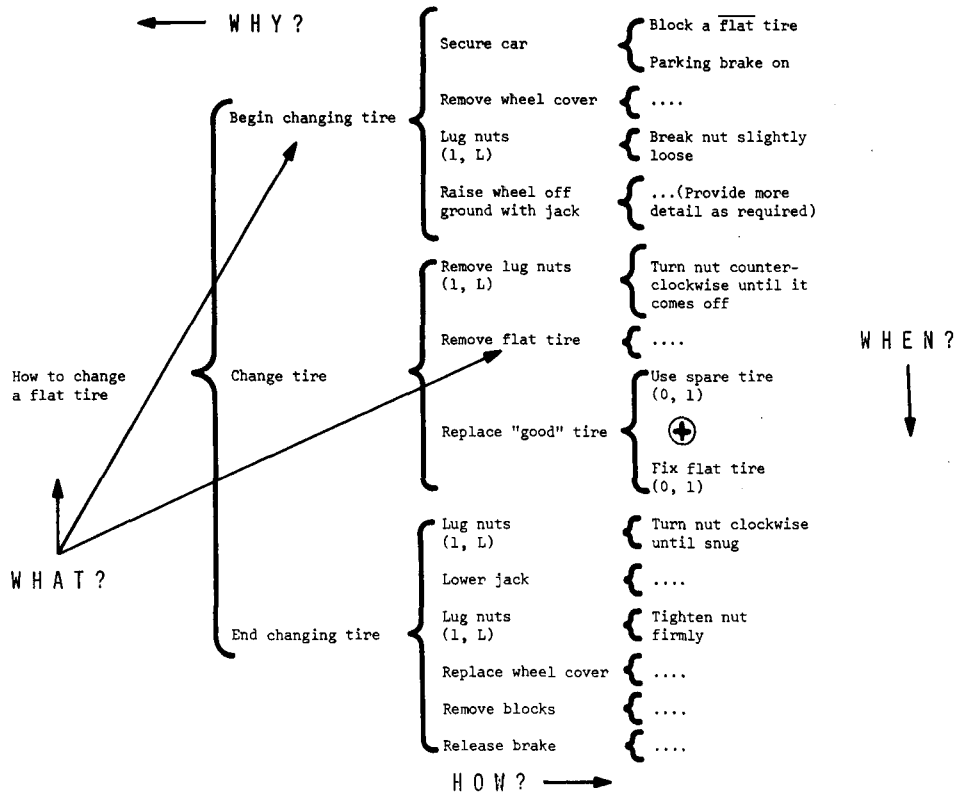


Figure 3. Example of a Warnier-Orr diagram. Note that the boldface interrogations and arrows are not normally included but have been superimposed to show how the diagram represents complex information. Note also that if the right side of the diagram is expanded until no process can be further subdivided (i.e., only functions remain), coding (construction) can proceed directly from the diagram, and the diagram fully documents the code regardless of the language involved.

that always performs the correct sets of actions on the correct sets of data at the correct times. Applying these criteria may also lead to the development of a program "proof-of-correctness" theorem, for if a program always performs only the correct sets of actions on only the correct sets of data at only the correct times, is it not correct?

SAMPLE PROGRAM

Problem Parameters

Assume we want a program that runs a laboratory exercise for a semantic priming lexical decision experiment. The student should sit down at the terminal, start the program, and classify the second letter string in each of a series of 100 prime-target pairs. Interval between prime and target, intertrial interval, number of stimuli, and so on, will be specified by the student. Output should consist of number of errors, mean reaction time, and the standard error of the mean for correct responses to associated words, unassociated words, and nonwords. All instructions, prompting for parameters, and so on, should be provided. Assume a coded, randomized file of prime-target pairs is available. Print out a summary of results.

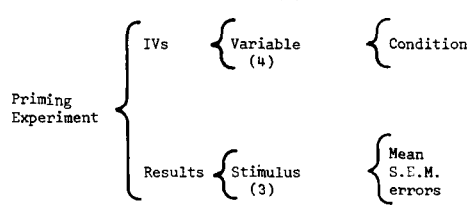
PRIMING EXPERIMENT SUMMARY
FOR SHARON SUSAN DWINK

INDEPENDENT VARIABLES	CONDITION
PR (TARGET IS A WORD)	.50
PR (TARGET IS ASSOCIATE OF PRIME/TARGET IS A WORD)	.50
PRIME DURATION	2 SEC
INTERSTIMULUS INTERVAL	.5 SEC

RESULTS

	MEAN RT	S.E.M.	# ERRORS
ASSOCIATED WORDS	694	11.3	0
UNASSOCIATED WORDS	773	14.7	1
NONWORDS	961	17.9	2

(a)



(b)

Figure 4. Step 1: Define the process outputs. (a) The process starts by specifying the desired output as precisely as possible. (b) The logical data structure summarizes the desired output. It serves as the "bones" of the process and completes Step 1.

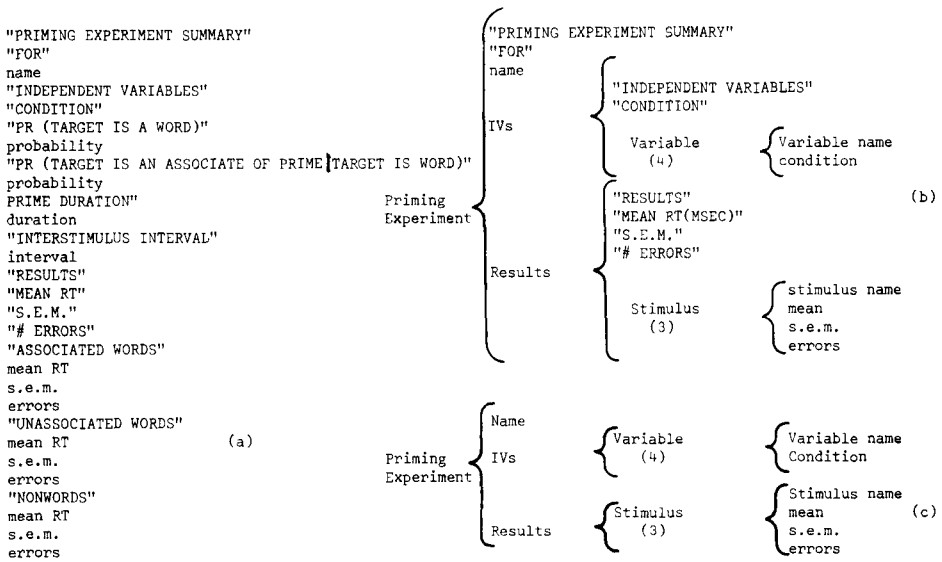


Figure 5. Step 2: Define the logical data base. (a) List all of the data elements that appear on the report, both data fields, and headings (in capitals enclosed by quotes). (b) The logical output structure (LOS) is developed to show the place in the data structure at which each element will occur. (c) Now remove computed data elements and eliminate redundant data elements from the LOS to produce the logical data base.

Entity	Attributes	
Stimulus	Stimulus Name, Statistics	
Statistics	Mean	$\Sigma X/N \leftarrow$? Potential Problem?
	MSe	$N, \Sigma X, \Sigma X^2$
	Errors	Σ Errors

1. Could N = zero? Yes, if no correct responses were made. But if N = zero, then a zero divide exception will occur and cause the program to fail. Therefore, eventual design will have to prevent a zero divide from occurring.
2. If N = zero MSe calculation will yield zero divide exception. Therefore, make sure it cannot ever occur.
3. The event analysis also suggests another problem. We do not actually need to store the mean, MSe, and errors (as is implied in Figure 5c). What we really want is the quantities necessary to compute the statistics.
4. What about instructions to the subject? How does the subject input four conditions? This information will have to be added to the logical data structure.
5. Etc.

Figure 6. Step 3: Define events. This step is done to rule out possible data exceptions, find hidden hierarchies, or identify any errors in the design. Event analysis proceeds by listing entities (hierarchy names) and attributes of entities in order to determine what real-world events could affect attributes. Event analysis is required for each entity in the hierarchy. To save space, only part of the event analysis is shown. Note also that findings in event analysis may require changes in an earlier design stage output. If so, the process starts over and, thus, is self-correcting.

SSD Procedure

There are six steps involved in developing the structured process that will provide the desired program (Higgins, 1979, chaps. 4-10): (1) Define the process outputs, (2) define the logical data base, (3) define events (real-world changes), (4) develop the physical data base, (5) design the logical process, and (6) design the physical process. Figures 4-9 present the result of each step, and Figure 9 presents the completed structured process for the desired program.

EVALUATION

User Reports

Evaluation of SSD has proceeded, and continues to proceed, along several paths. For example, there are several "experiential" sources. A number of case histories of organizations that have adopted SSD are available from Ken Orr and Associates, Inc.,[3] as part of the marketing information for their courses in SSD. The usual report describes successful implementation of and satisfaction with SSD. There is almost always an enthusiastic testimonial referring to an n-page "COBOL program which ran the first time without error," where n is always a large number. In a similar vein, I made extensive use of SSD (particularly the Warnier-Orr diagrams) in converting the 90K MESS 3-SC (EXPER SIM) package to WISER, which runs on an IBM 360/30 65K machine (Forbach, 1979). And although WISER did not run the first time it was tested, the conversion (described by one experienced EXPER SIM user as "impossible") was greatly facilitated by use of Warnier-Orr diagrams. Finally, speakers and participants in the 5th Annual SSD User's Conference[4] echoed the common sentiment that while use of Warnier-Orr diagrams does not guarantee a magic solution to all programming problems, consistent, disciplined application of SSD is producing a noticeable impact on data processing in their organizations.

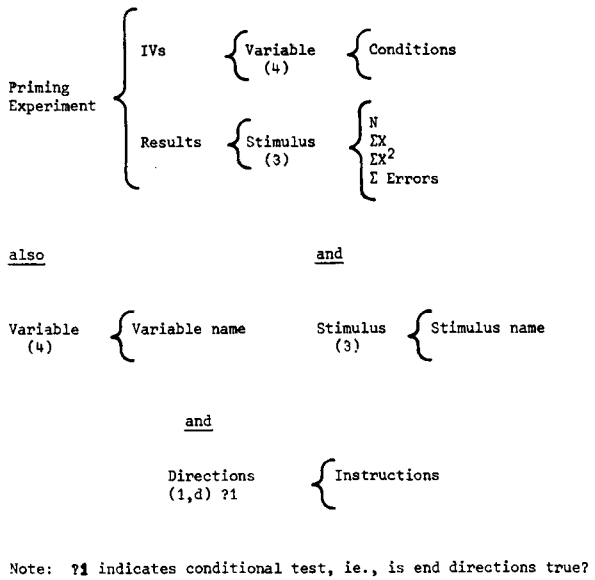


Figure 7. Step 4: Develop the physical data base. Start with the logical data base and remove to another location all primary elements that change infrequently. Note the additions to the data base resulting from event analysis.

Relative Comparison Ratings

A more quantitative type of evaluation of SSD is presented in a 70-page report commissioned by the National Bank of Detroit that compares five structured methodologies (Stevens, Note 2). The methods evaluated were IBM, YOURDON (Constantine), SADT (structured analysis and design technique), SSD, and JACKSON. The five were rated on a scale from 1 to 5 (5 is maximum rating) relative to each other for each of 10 relevant criteria. Table 2 indicates that SSD had the superior ratings and, in fact, was only 3 points short of a perfect score. A short summary of this report is available (Stevens, Note 3).

SSD Survey

A third evaluation strategy involved surveying SSD users who had been practicing the procedures for at least 1 year. Most respondents are managers in organizations (both industry and government) that paid for SSD training for one or more analysts/programmers in the organization, and thus responses generally reflect organization behavior, not individual behavior. Respondents were asked to indicate on a continuously drawn scale with six equal intervals (between 1 and 7) the degree to

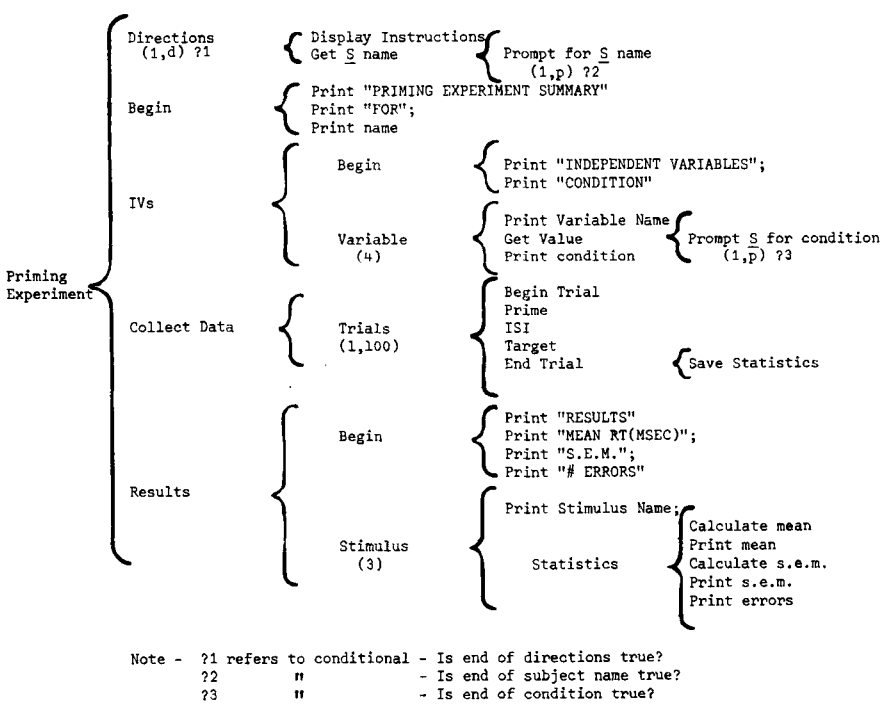


Figure 8. Step 5: Design the logical process. It is now time to design the logical process that will lead to the desired output. As required, each level in the hierarchy is segmented into begin process, process, and end process. Start by completing end process for the first level, and then move to the right through the end process for the other levels. Then fill in the begin process for the highest level and move back to the left, finishing with the begin process in the first level.

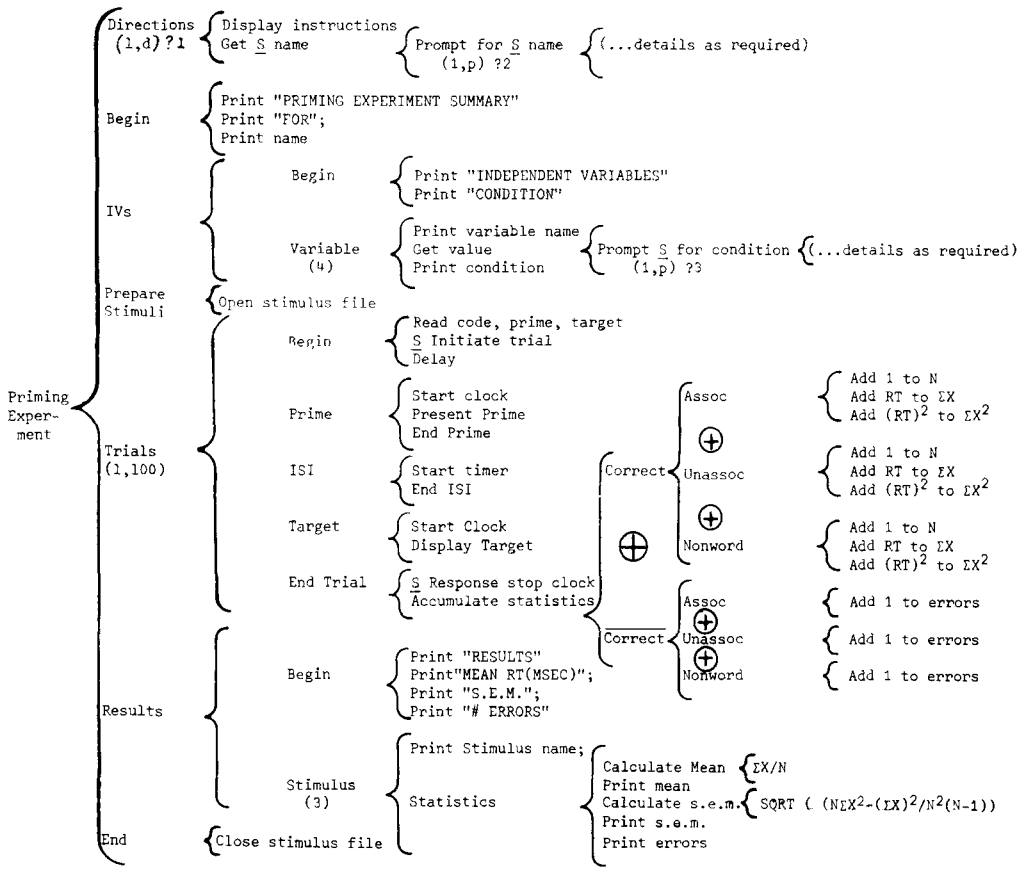


Figure 9. Step 6: Design the physical process. Complete the design process by adding control breaks and necessary file-handling instructions.

which they agreed with each of 12 survey statements. Respondents were explicitly encouraged to check noninteger values on the scale if a noninteger value best described their assessment. The low end of the scale (1) was anchored by labeling it "strongly disagree," the middle (4) was labeled "neutral," and the high end was anchored by labeling it "strongly agree." Thus it was expected that a respondent with no relevant experience or no particular belief would check "neutral" (4). However, the individual with a particular belief about SSD would indicate both the direction and strength of the belief with the response. One-third of the statements were worded so that disagreeing indicated a positive view of SSD. The remaining two-thirds required agreement to indicate a positive evaluation of SSD. A cover letter identified the author as a psychologist affiliated with Washburn University and assured confidentiality of responses. Of the 24 questionnaires sent, 19 were completed and returned.

Survey questions were constructed to try to answer

Table 2
Stevens' Comparative Analysis

Criteria	Method				
	IBM	YOURDON	SADT	SSD	JACKSON
Independence of Hardware and Software	5	5	5	5	5
User Interface (Participation/Comprehension)	2	3	4	5	1
Control/Consistency (as Design Aid)	2	3	5	4	1
Customer Acceptance	3	4	4	5	4
Teachability	2	2	3	5	4
Aid to Programming	2	3	1	4	5
Ease of Drafting	2	3	4	5	1
Reference	2	3	5	4	1
Maintenance	3	2	4	5	1
Ease of Use	2	1	4	5	3
Totals	25	29	39	47	26

four general questions. First, are the assertions of SSD proponents regarding advantages of the methodology supported by the responses of SSD users? Second, is it possible to estimate the amount of time required for successful SSD implementation? Third, are there any indirect indicators of SSD efficacy (e.g., increased programmer consistency, savings of time, positive benefit-cost ratio, etc.). And fourth, is there evidence that veteran programmers resist adopting SSD as is sometimes the case with "new" methods?

Survey results and discussion. Table 3 presents the survey statements in four groups that correspond to the four questions above. Included in the table for each statement is the predicted mean response, the actual mean response, and a t value for a directional test for difference from the "neutral" response. Alpha was set at .01 per group of statements. Thus alpha for any one test was .01 divided by the number of questions in the group. This rather stringent criterion was adopted to provide adequate protection, since four sets of tests served as the basis for conclusions.

Inspection of Table 3 shows that respondents generally concurred with the five assertions of SSD proponents regarding SSD methodological advantages. All ts were significant and in the predicted direction. Thus managers apparently believe that use of SSD in their shops does produce some benefits.

The average response for each of the two statements regarding implementation time were both in the predicted direction, but only the second was significantly different from "neutral." Thus while respondents agreed that SSD methods continued in use after 1 year, they were not willing to concede general use (acceptance) after 3 months.

As can be seen in Table 3, the attempt to assess SSD effects indirectly was partially successful. Respondents agreed that SSD improved the consistency of users' work, and they strongly denied unwillingness to recommend the expense of SSD training. They did not appear to believe that SSD saves time over the life of a large project or that the investment in SSD training is returned several times in reduced maintenance problems.

Finally, the average response to the last statement in Table 3 indicates that SSD acceptance is not simply a function of experience as a programmer. This is not consistent with several industry reports that have described resistance to some "new" methods, primarily by veteran programmers. Possibly, SSD does not evoke as much resistance as some other structured programming techniques.

Conclusions. Many potential users on first exposure to SSD feel that it is too simple and too straightforward to be of any real value in designing complex programs/systems. However, it is hoped that this very brief intro-

Table 3
Summary of Survey Information

Survey Statements	Mean Response Predicted	Actual	t*	α
		SSD Claimed Advantages		
SSD is completely hardware independent.	> 4	6.06	6.32**	
SSD is easily taught to novice programmers.	> 4	5.62	5.68**	
Readability of program documentation is rather poor with Warnier-Orr diagrams.	< 4	2.03	−7.87**	.002/Test
SSD concepts are reasonably simple to understand.	> 4	5.77	6.90**	
SSD concepts are reasonably simple to practice.	> 4	5.38	4.71**	
		Implementation Time		
Three months after initial exposure to SSD, most of our programmers continued to use the SSD approach.	> 4	5.11	2.80	.005/Test
One year after initial exposure to SSD, few of our programmers continued to use the SSD approach.	< 4	2.53	−4.17**	
		Indirect Benefits		
Use of SSD has improved the consistency of our users' work.	> 4	5.12	3.71**	
Using SSD actually saves very little time over the life of a large project.	< 4	3.14	−2.16	.0025/Test
The investment of time and money into the SSD training program has been returned several times due to fewer program maintenance problems.	> 4	4.77	2.12	
If I went to a new company, I would probably not recommend that the company incur the expense of SSD training.	< 4	1.84	−6.58**	
		Veterans Resistance		
SSD is quickly adopted by veteran programmers.	< 4	4.36	1.07	.01

*Note—Probability level varies as a function of the number of tests per set of statements. *df = 18. **Indicates significant t.*

duction to SSD concepts will discourage this tendency and that the positive evaluation results will encourage some further reading and trial usage, especially of the Warnier-Orr diagrams. The consistent application of these techniques could potentially solve each of the four problems identified in the introduction. And the time investment required to learn SSD is small enough to make it a realistic possibility, even if one is not a professional analyst/programmer.

RECOMMENDED READING

The reader who would like a fairly concise source of background information on structured techniques in general is referred to McNurlin (1979) and Rudkin and Shere (1979). For more information pertaining to SSD, read Higgins (1977a, 1977b, 1978). An inexpensive paperback by Higgins (1979) is probably the best single reference book. It is especially useful if you program in BASIC on a microcomputer. Two books by Orr (1977, in press) are also excellent reference sources, although it will be several months before the new book is available. Finally, the English translation of Warnier's (1981) newest book is now available and is highly recommended as a general reference for data-driven structured systems. The book reflects several years' work by Warnier's group at Cii Honeywell Bull in France, and it demonstrates why Warnier has an international reputation as a systems design consultant.

COURSE INFORMATION AND SUPPLIES

Ken Orr and Associates offer in-house, public, and video cassette SSD courses. They are expensive, but they may represent a cost-effective solution for some users, particularly those with computer networks or data-base management problems. A quarterly publication, *Future(s)*, updates course information, disseminates information on SSD improvements, and in general keeps users up to date. Also available is a "Warnier-Orr template" with braces, connectives, and standard computer device symbols ($3) and "structured documentation forms" ($4/pad of 50). These drafting aids are of course not necessary, but they facilitate quick production of a neat, clean file copy of the design documentation. For information on courses, *Future(s)*, or documentation supplies, contact Ken Orr and Associates (see Footnote 3).

REFERENCE NOTES

1. Ross, D. T. *Principles of structuring* (Report No. TP082). SofTech, Inc., Waltham, Mass., 1978. (Available from SofTech, 460 Totten Pond Road, Waltham, Massachusetts 02154.)

2. Stevens, B. M. *Structured system design review.* Unpublished manuscript, 1977. (Available from Ken Orr & Associates, 715 E. 8th, Topeka, Kansas 66607. May require copying fee.

3. Stevens, B. M. *Structured techniques: Comparative analysis.* Unpublished summary, 1977. (Available on request from Ken Orr & Associates, 715 E. 8th, Topeka, Kansas 66607.)

REFERENCES

Bohm, C., & Jacopini, G. Flow diagrams, turing machines and languages with only two formation rules. *Communications of ECM 9,* 1966, 5, 366-371.

Canning, R. G. (Ed.). The production of better software. *EDP Analyzer,* February 1979, pp. 1-13.

DeMarco, T. *Structured analysis and systems specification.* New York: Yourdon Press, 1978.

Dijkstra, E. W. *A discipline of programming.* Englewood Cliffs, N.J: Prentice-Hall, 1976.

Forbach, G. B. EXPER SIM: Review and update. *Behavior Research Methods & Instrumentation,* 1979, 11, 519-522.

Higgins, D. A. Structured program design. *Byte,* October 1977, pp. 146-148; 150-151. (a)

Higgins, D. A. Structured programming with Warnier-Orr diagrams. Part 1: Design methodology. *Byte,* December 1977, pp. 104; 106; 108-110. (b)

Higgins, D. A. Structured programming with Warnier-Orr diagrams. Part 2: Coding the program. *Byte,* January 1978, pp. 122-126; 128-129.

Higgins, D. A. *Program design and construction.* Englewood Cliffs, N.J: Prentice-Hall, 1979.

Jackson, J. A. *Principles of program design.* New York: Academic Press, 1975.

McNurlin, B. C. Program design techniques. *EDP Analyzer,* March 1979, pp. 1-13.

Mills, H. D. Top-down programming in large systems. In R. Rustin (Ed.), *Debugging techniques in large systems.* Englewood Cliffs, N.J: Prentice-Hall, 1971.

Orr, K. T. *Structured systems development.* New York: Yourdon Press, 1977.

Orr, K. T. *Structured requirements definition.* Topeka, Kan: Ken Orr & Associates, in press.

Rudkin, R. I., & Shere, K. D. Structured decomposition diagram: A new technique for system analysis. *Datamation,* October 1979, pp. 130-133; 136; 140; 144; 146.

Warnier, J. D. [*Logical construction of programs*] (3rd ed.) (B. M. Flanagan, trans.). New York: Van Nostrand Reinhold, 1976.

Warnier, J. D. *Logical construction of systems.* New York: Von Nostrand Reinhold, 1981.

Yourdon, E., & Constantine, L. *Structured design* (2nd ed.). New York: Yourdon, 1978.

NOTES

1. Note that although the redundant right (closed) brace is deleted from each level of the diagram to produce less clutter and improve readability, it is implicitly present, consistent with the need to delimit set domain.

2. Concurrency is actually more a systems concept than a program concept, since two functions never actually execute concurrently (simultaneously). However, since a system may require keeping track of concurrent activities, it is provided for in the notation.

3. Formerly Langston Kitch & Associates, now Ken Orr and Associates, 715 E. 8th, Topeka, Kansas 66607; telephone (913) 233-2349.

4. The enthusiasm of the speakers was not entirely unexpected. However, approximately 60 users paid $450 each to register for the conference, indicating a fairly strong commitment to attend the conference and keep up with SSD developments and applications.