A METHOD FOR DESIGNING PROGRAMS AND ITS APPLICATION TO INTRODUCTORY COMPUTER SCIENCE COURSES

by

Lynne J. Baldwin

August 1974 TR-39

This paper constituted the author's dissertation for the Ph.D. degree at the University of Texas at Austin, August 1974.

THE UNIVERSITY OF TEXAS AT AUSTIN

DEPARTMENT OF COMPUTER SCIENCES

ABSTRACT

A method for designing programs is proposed as an organizational framework for the programming task. As such, it is intended to be taught to the introductory student and utilized by him at both the elementary and advanced levels of programming.

The proposed method for designing programs has the following features. First, it encompasses the entire programming task, Second, it incorporates general problem solving techniques and focuses them on the specific tasks in programming, Third, it defines the major steps in programming along with the specific tasks which must be accomplished during each step, Finally, the method is concerned not only with the overall approach used in carrying out the programming task but also with the design quality of the resulting program; therefore, considerations of the desired program characteristics of correctness, ease of debugging, adaptability, readability and efficiency are included as an integral part of the method.

The incorporation of this method for designing programs into an introductory course affects both the course content and the course structure. A description of such a course is given as an illustration of the applicability of the proposed method.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Topic Description and Goals

The first goal of this paper is to present a method for designing programs which

1) encompasses the entire programming task,

2) incorporates general problem solving techniques by focusing them on the specific tasks in programming,

and 3) defines the major steps in programming along with the specific tasks which must be accomplished during each step.

The main purpose in presenting this method is to provide an organizational framework for the programming task which can be taught to introductory students and utilized by them at both the elementary and advanced levels of programming.

The second goal of this paper is to show the applicability of the proposed method by presenting an example of an introductory course in programming. This course is comparable to an introductory computer science course at the college level. The course content and the course structure are different, however, from that of the typical introductory course since the proposed method for designing programs is an integral part of both the course content and the course structure.

1.2 Motivation and Justification

An introductory course in a computer science curriculum, as outlined in "Currivulum 68" [ACM, 1968, p. 156], is designed "to provide

the student with the basic knowledge and experience necessary to use computers effectively in the solution of problems;" all subsequent courses depend on such an introductory course. In order for the student to be able to use the computer in the solution of problems, he is going to need information about the computer and how to use it, such as what a computer language, a program, an algorithm and data are and how to use them in implementing solutions to problems. This type of information is adequately provided in introductory courses which parallel that outlined in "Curriculum 68." However, the student is also going to need information about solving problems and designing programs, such as what should be done in analyzing a problem and how this analysis relates to the desired program. This latter type of information is usually left to the individual student to assimilate for himself, mainly through trial-and-error experience with programming problems, so that the student is learning about solving problems and designing programs at the problem solving level, which is the highest level of learning [Gagné, 1970, p. 214]. Programming is such a complex task, requiring problem solving behavior at many points throughout, that is seems desirable to provide the student with an overall plan of attack for designing programs which are to be solutions to problems; there are many other problems he will have to solve besides trying to formulate a general plan for programming by himself. Merrill [1970] has indicated that the learning level can be adjusted to any desired level for any subject matter. By defining an overall plan, in this case the method for designing programs proposed in this paper, and by presenting it to the student for him to use throughout the introductory course, the learning level may be lowered from that of problem solving to that of

principle learning or even lower. He will have at hand at least one method of approaching a programming task when he reaches the end of the course — a method which he can use in subsequent courses and can build on or alter to suit his own personal problem solving abilities as he becomes more experienced.

1.3 Relationship of Programming and Problem Solving

Problem solving is considered to be the most complex of the thought processes. Simon [1969, p. 26] describes it as "a search through a vast maze of possibilities, a maze that describes the environment. Successful problem solving involves searching the maze selectively and reducing it to manageable proportions." Another definition, and one which aligns itself with the intent of incorporating the method for designing programs into an introductory course, is that given by Gagné. He views problem solving as "a process by which the learner discovers a combination of previously learned rules that he can apply to achieve a solution for a novel problem situation. It is also a process that yields new learning" [Gagné, 1970, p. 214]. He further maintains that what emerges from problem solving is a higher-order rule [Gagné, 1970, p. 224]. Simon's statement develops a feeling for the complexity and immensity of the search process in problem solving. Gagné's statement, however, attempts to simplify the view of this complex process by stating exactly what it is that the problem solver is working with and what it is that he has as a result of his having solved the problem. This complex process of problem solving is, then, a search process through a set of rules in order to find that combination of rules which works. It results in

learning for the problem solver which can, in turn, be used for solving other problems.

The relationship between problem solving and programming is rather complex. First, programming belongs to the realm of problem solving. Programming requires that a solution in the form of a computer program be produced for a given problem, where the problem is assumed to be novel in some, if not all, of its respects. (If this were not the case, an existing program could be used as a solution.) Because the process of producing a program is rarely limited to the simple application of one or two rules, the required behavior reaches problem solving proportions. The programmer is faced with the task of discovering the combination of rules which will produce the required program; the rules he chooses from are many and complex in themselves.

Second, programming is a special kind of problem solving. It is special because the product of programming is of a singular nature — it is a computer program. The characteristics and requirements of this product give programming its special niche in problem solving.

Third, programming is a complex form of problem solving. It is a case of problem solving within problem solving. For example, the final product is to be a program which will be the solution to the given problem — this is the overall problem for the programmer. However, within this overall problem are at least two other "subproblems." First, there is the subproblem of creating a solution plan for the given problem; and secondly, the solution plan must be translated into the desired program. Such subproblems exhibit special characteristics and requirements within the overall problem of programming, just as

programming exhibits special characteristics within problem solving as a whole.

Thus, programming is a kind of problem solving and as such is subject to general problem solving methods. Programming also requires problem solving methods for its subproblems. It therefore seems advantageous to view various general problem solving methods which have been observed and described in the literature in order that the advantages of using these methods can be incorporated into programming. The deliberate and conscious application of these methods to the programming process as a whole and to the various subproblems, while taking into consideration the characteristics and requirements of the subproblems, should increase the efficacy of programming efforts throughout the design process.

The literature — problem solving methods and strategies

The sequence of events involved in problem solving has been described by several investigators. Several early investigators suggested the sequence of events in problem solving as being preparation, in which extensive study of a problem is done; incubation, in which little or no contemplation about the problem is done; and finally, illumination, in which the solution to the problem appears in the mind's eye all at once. This sequence of events is supported by accounts of the personal experiences of scientists, mathematicians, and others; one collection of such experiences appears in an essay by Hadamard [1945]. Later descriptions have expanded this sequence. For example, Osborn [Sackman, 1970, p. 231] gives the following: orientation to the problem, preparation by gathering data about the problem, analysis by breaking down relevant material,

hypothesis by generating alternatives, incubation to invite illumination, synthesis by putting pieces back together and, finally, verification by judging the results.

A description of the events which make up problem solving does not offer specific actions which can be applied. Working rules based on experiment and experience can, however, offer guidance. Hyman and Anderson [1965] discuss several general rules designed to aid the problem solver in problem solving situations. Their rules fall into two categories. The first is concerned with what the problem solver should do when first confronted with a problem. It contains the following rules:

1) Go over the elements of the problem rapidly several times until a pattern encompassing all problem elements emerges; a total picture of the problem should be available before work begins on the details.

2) Instead of accepting the first hypothesis or interpretation that is formulated as a possible solution, suspend judgment about the way to proceed for a time.

3) The arrangement of the elements of the problem, when it is first encountered, may hide familiar patterns; therefore, exploration of different arrangements of the problem elements, both temporally and spatially, may suggest a solution.

The second category is concerned with what the problem solver should do when he is blocked in his attempt to solve the problem or wants to find a better solution than the present one. The following rules apply to this category:

4) When no progress is being made on a problem, abandon the present approach and concentrate on another aspect of the problem; persistence in a direction which fails to yield a solution tends to inhibit the openmindedness needed to discern new, potentially rewarding, directions.

5) It may be possible to find a better, or more creative, solution to a problem if a second solution is produced after the first one.

6) In order to find a better, or more creative, solution a critical evaluation of personal ideas, i.e., looking for weak points, and a constructive evaluation of others' ideas, i.e., looking for strong points, can be helpful.

7) It is often possible to find a new, more helpful, relationship among the problem elements if a different representational system is used; this can mean changing a verbal representation to a graphical one, an abstract to a concrete one, and so forth.

8) If, after a thorough exploration of the possibilities that a present approach to the problem offers, the problem solving task appears to be blocked, a break away from the problem can be helpful; this allows for a period of "incubation" hopefully to be followed by an "illumination" that will yield a solution. It is probably useless to take a break unless sufficient exploration of the problem has been done beforehand.

9) Extensive internal critical analysis of the problem can fail to produce a solution. At such a time it may be helpful to discuss the problem with someone else; this forces a complete explanation of all parts of the problem and the possible ways of solving it. Even if the other person does not understand the problem, the very act of communicating about it forces the mind to express the problem in a slightly different way which may provide new insight.

These nine suggestions are, obviously, broad strategies to be used for any kind of problem solving task and, as such, can be applied to problem solving in programming. However, they do not give specific actions which are helpful when dealing with the problem itself, for example, where to start when confronted with a problem.

One author, Polya, does offer specific actions which are helpful throughout the process of solving problems. Although his work has been done in the context of mathematical problem solving, the methods of solution useful in mathematics are also applicable to other kinds of

problems. Polya [1957, p. xvi] gives an overall method of approach for the problem solver. Briefly, it consists of these four parts:

1) Understanding the problem: determining unknowns, data, and conditions of the problem.

2) Devising a plan: finding the connection between the data and the unknown and eventually devising a plan of action.

3) Carrying out the plan: checking each step in the plan as it is carried out to insure that it is correct.

4) Looking back: examining the solution obtained to see if it can be obtained differently, to see if the result or the method is useful for some other problem.

Suggestions for gaining insights into a problem include restating the problem by drawing pictures and setting up equations, decomposing and recombining the parts of the problem, and considering related and auxiliary problems and their application to the problem at hand. The method given above is concerned with "problems to find" in which the object is to find the answer to the problem. There is another kind of problem — a "problem to prove" — in which a proof or disproof of an assertion is required. Polya [1957, pp. 154-157] also gives a method for problems to prove which parallels that for problems to find. In this method, determining what the hypothesis and conclusion are, so that the connection between them can be constructed as a proof, replaces the unknowns, data, and conditions between them as given in the "problems to find" method. Both kinds of problems are found in the programming task.

Besides overall methods of problem solving as described by Osborn and specified by Polya, there exist other problem solving strategies useful within an overall method. Especially important are

the strategies associated with the generation, evaluation, and validation of hypotheses where hypotheses are generated by the problem solver as being possible solutions, where evaluation is carried out by applying a hypothetical solution to the problem to see how well it works, and where validation is carried out by confirming that the hypothesis does, in fact, meet the requirements of the problem and is, therefore, a solution.

There are several strategies that can be used in generating hypothetical solutions. The most general of these is the generation of the set of all potential solutions after which each element in the set must be evaluated. This is identified by Newell and Simon [1972, pp. 95-97] as set-predicate formulation.

A second strategy is to generate all possible solutions while eliminating the generation of those which are obviously or probably incorrect. One way of doing this is by the use of preselection as described by Wirth [1971] in which the number of generations is reduced logically before generation begins. For example, the generation of prime numbers can be done by generating all positive integers and testing each one, or the generation effort can be reduced by assuming that two is prime and that all other primes can be found by generating only the odd integers and testing them.

When it is not possible to take advantage of obvious reductions in the generation scheme, some form of heuristic search [Newell and Simon, 1972, p. 98; Polya, 1957, pp. 129-134] is essential. Heuristic search involves the generation of a hypothesis which is subsequently evaluated; then, depending upon the evaluation, a new hypothesis may be generated.

There are several search strategies useful in heuristic search. The first one is to break the problem apart into subproblems so that they can be attacked individually. This is called factorization [Newell and Simon, 1972, pp. 74-75] or decomposition [Wirth, 1971; Polya, 1957, pp. 75-85]. The solutions to the individual problems, once they have been evaluated and validated, are combined together to give the solution to the original problem. The combined solution must then be evaluated and validated as well.

The strategies of working forward and working backward each proceed from one end of a solution chain to the other end. Using information from only the starting end of the chain, each then proceeds to the other end, which is used to establish that the chain is complete. Working forward considers that the starting end is the problem itself. Working backward starts with the solution and proceeds to the hypothesis. These are sometimes referred to as top-down and bottom-up, respectively.

A related strategy is means-ends analysis [Newell and Simon, 1972, p. 416]. This strategy is based on the assumption that there is a discernable difference between the given problem and its solution, or end, and that the solution of the problem requires the application of some means, in the form of functional operators, which will reduce the difference between the problem and its solution. This strategy requires the generation of a hypothesis of how to reduce the difference discerned and an evaluation of the results of the application of the hypothesis. Consideration of both ends of the solution chain is therefore necessary when using means-ends analysis for both generation and evaluation of a hypothesis.

The generation of alternate hypotheses can be carried out using various search strategies. Such strategies can be described in terms of

a hypothesis tree. From any one hypothesis, or node, in the tree one or more hypotheses may be generated which have a potential of leading to the desired solution. One search strategy used is the breadth first strategy in which the nodes are expanded (hypotheses are generated) in the order in which the nodes are generated [Nilsson, 1971, p. 45]. Generation and evaluation of hypotheses continues node-by-node until the solution is reached. A second strategy is the depth first strategy in which the most recently generated hypothesis (node) is expanded first [Nilsson, 1971, p. 48]. Expansion continues to take place on the most recent node until a solution is found. If a solution is not found before a maximum depth is reached in the tree, backtracking can be used to start a new path through the tree from an earlier node. Backtracking requires that former paths be remembered so that it will be possible to back up to a previous node and start a new path from there. Also, formerly explored paths must be remembered so that they will not be reexamined. Both the breadth first and depth first strategies will eventually generate all possible solutions paths for finite trees; however, for large trees and for infinite trees the search is usually modified by the adoption of heuristic search strategies in order to reduce the amount of searching to practical limits. Such strategies involve the evaluation of the potential associated with the various nodes so that the nodes with the greater potential are expanded first. Such strategies are applicable to both breadth first and depth first search patterns. They are, of course, more complex due to the necessity of including an evaluation scheme.

Hypothesis generation can be formulated by any one of the above strategies. Usually, however, these basic strategies are combined into more complex strategies where one strategy is applied for a time, then another strategy is used on the results of the first strategy, and so on.

The evaluation of a hypothesis as a solution or a solution part is governed by varying criteria depending upon the generation scheme used; but, in general, it determines how well a possible solution meets the requirements of the given problem. If a possible solution fails to meet the requirements, then an evaluation must be made to determine why it failed. For example, in the case of means-ends analysis, evaluation determines what the difference is between the hypothesis and the desired solution; then, further hypothesis generation is made to reduce the evaluated difference. If a tree search strategy is used and it is found that a hypothesis path does not lead to a solution, evaluation of alternate paths is necessary. Such evaluation is involved in backtracking to a former node in the tree when a new path is to be chosen on its evaluated potential.

Once a hypothesis or set of related hypotheses have been evaluated as being a solution to a given problem, then a validation is necessary to determine that, in fact, a solution has been found. This is accomplished by checking that all conditions in the problem are met by the proposed solution. Validation may be considered to be a proof, however informal its form.

There are several other strategies useful in solving problems in general besides those found in hypothesis generation. One is the utilization of analogy [Polya, 1957, p. 37]. A previously solved analogous

problem, while not possessing all the conditions of the given problem, may be close enough so that either the answer to the analogous problem or its method of solution may be applied in the solution of the given problem.

The utilization of variables and parameterization are especially useful in extending the specific solution for a given problem to a general solution for a class of problems. This allows the problem solver to solve all problems of a given class, the specific solutions of which can be found by manipulation of variables and parameters in the solution, rather than by re-solving the specific problem.

Finally, the introduction of auxiliary elements [Polya, 1957, p. 46] is common in problem solving. Auxiliary elements are elements not included in the given problem but are introduced during problem solution. An auxiliary element may be a drawn figure based on the problem definition, it may be an equation based on the conditions in the problem, or it may be an auxiliary problem introduced in an effort to simplify the given problem in some way.

There is one aspect of problem solving which can perhaps expand problem solving abilities as well as yield a better solution to a just-solved problem. This aspect is re-analysis — re-analysis of both the answer to the problem and the solution process used to find the answer [Polya, 1957, pp. 61-64]. A review may yield a better answer for a problem for which there is more than one satisfactory answer. A review may yield a shorter, more compact, solution process — something which can be important in the design of programs. A review of the process used to solve this particular problem may uncover alternative ways of

solving the problem which can be applied to the solution of other problems. It may even expand the problem solver's repertoire of strategies if he makes a conscious effort at analyzing his own problem solving behavior.

A summary of problem solving and programming

The problem solving methods and strategies discussed above are related to programming at three levels. At the top level, problem solving methods can be applied to the overall task of programming in order to establish a working framework of actions which can be used in programming. The method proposed here applies initially at this level.

At the second level, they can be applied to the specific problem to be programmed in order to get a solution plan. They can be applied to the task of producing a program from the solution plan, and they can be applied to the problems encountered in testing and debugging. Specific actions at this level are the main part of the proposed method for designing programs.

At the third level, the various problem solving strategies can be incorporated into the program as part of its processing scheme. For example, heuristics are used whenever trial-and-error generation is needed, i.e., whenever there is no formula or algorithm, in the strict sense of the word, available. Programs using heuristics include game playing programs when the choice of moves exceeds practical dimensions and programs in which human behavior patterns are being simulated. One strategy of problem solving which is automatically used in programs is the utilization of variables and parameterization. This third level is not explicitly treated in the proposed method for designing programs because it is specific to particular programs rather than to the programming process in general.

1.4 Designing Programs

The programming task can be broken roughly into three main parts. The first part is analyzing the problem and creating a solution plan. The second part is creating an algorithm that represents the solution plan in a programmable form. The third part is creating a running program based on the algorithm. A method for designing programs which applies to the entire programming task must, of course, encompass all three parts of the task in a unified and comprehensive manner so that the application of the method can start with the problem statement and carry through to the end of the programming task. Several contributions have been made in the literature which are intended to increase the quality of the programs while decreasing the time and effort spent on programming. All have their advantages at certain points throughout the programming task; however, none seem to provide guidance for the entire task.

The literature — programming contributions

Structured programming with stepwise refinement has been discussed extensively by Wirth [1971, 1973] and Dijkstra [1969, 1971]; on-the-job evaluation of it has been given by Baker [1972]. Of the programming techniques described in the literature, this one seems to be applicable to more of the programming task as a whole than the others. Structured programming is based on the utilization of a set of basic control structures which can be used in verifying the correctness of algorithms and the resulting programs. Stepwise refinement is basically the general problem solving technique known as factorization or top-down

decomposition; it works especially well with structured programming where more detailed structures are substitutable for more generalized ones. Structured programming with stepwise refinement allows for a gradual decomposition of the problem into more specific parts while simultaneously building up the parts of the desired program. Some disagreement has been expressed with this programming technique in that it is strictly top-down and does not necessarily work at all times [Naur, 1972]. Wirth [1973, p. 126] himself has pointed out that programming is not performed purely top-down or bottom-up, but may use both. Thus, while this proposal does not formally specify the use of problem solving strategies other than top-down, other strategies are useful. Specific advice as to how a problem should be approached is not given except for the implicit use of decomposition. Thus, structured programming with stepwise refinement does not encompass the entire programming task, especially with regard to analyzing the problem that is to be programmed.

Naur [1969] has proposed the use of action clusters in programming. This proposal can be classified as a bottom-up strategy of problem solving. This proposal does contain some advice for analysis of the problem; namely, that for each global requirement (invariant condition) in the problem statement, a set of action clusters (sequences of algorithmic statements) should be derived. Basing the derivation of the clusters on the requirements of the problem in this way insures that the compatibility between the resulting program and the requirements can be proved. Unfortunately, this proposal assumes the explicit presence of the global requirements and does not, therefore, specify how to analyze a problem in order to separate out such requirements. Also, the proposal reflects

only one problem solving strategy. Programming by action clusters, while being somewhat more specific about relating the derivation of the program parts (clusters) to the statement of the problem, does not encompass the entire programming task nor specify other problem solving strategies which are useful.

Structured programming with stepwise refinement and programming by action clusters are similar in one respect: both give implicit organization to the resulting programs. Structured programming with stepwise refinement, by allowing the problem to be broken down gradually while building up the program gradually, produces an after-the-fact program organization where the subprograms or modules are the result of the ongoing process of refinement, rather than from analysis of the problem before production of code begins. Programming by action clusters starts with the coded clusters and builds a program around them. Again the program organization tends to be after-the-fact. In contrast to this, Parnas [1971a,1971b,1972a,1972b,1972c,1972d] has proposed a paradigm for software modularization which emphasizes the need to completely determine the program organization before coding begins on any part of it. Parnas has given some general advice on what to look for in the given problem in order to determine how to organize the parts of the desired program. His proposal is concerned mainly with the way in which the program parts or modules are defined; this definition then allows the entire modularization of the program to be checked before any coding is begun. While some suggestions are given as to what to look for in the given problem in order to establish program modules, Parnas's method of software modularization does not offer extensive advice on how to

establish the modules, nor is his method concerned about the creation of the program code for the modules other than that it reflect the elements and actions defined for the modules. Thus, this method does not encompass the entire programming task nor does it suggest problem solving strategies particularly useful for this method.

Other proposals in the literature tend to be even more specific to one small part of the programming task than the ones discussed above. For example, Neely [1973] has suggested a method of indentation to be used in program coding which would increase the readability of a program by directly reflecting the control structures of the program. Other proposals focus on flowcharting techniques which support the style of programming being utilized. For example, Nassi and Schneiderman [1973] suggest a flowchart technique which is intended for use with structured programming.

A summary of design proposals

While these and other proposals are applicable at various points throughout the programming task, none are general enough to give perspective to the entire task, nor is any combination of these able to span the task. The most serious omission is the lack of direction for approaching the problem to be programmed; i.e., little is said about the specific elements to look for in the problem or how parts of the problem relate specifically to the parts of the program to be created. Secondly, only implicit suggestions about solving problems are given. Since these proposals are primarily for experienced programmers, they can, of course, assume that problem solving methods are already known and

need not be dwelled upon. However, because of these reasons it is apparent that if a method for designing programs is to be taught to the beginning student, it needs to encompass the entire programming task (allowing for the utilization of the various proposals given above if desired) and to incorporate various problem solving techniques for use in the overall task of programming as well as for use in solving the problems that are to be programmed. Such a method for designing programs is presented in this paper.

CHAPTER 2

A METHOD FOR DESIGNING PROGRAMS

Before proceeding to the specific details of the proposed method, brief discussions of the following are given in anticipation of the method. First, a definition of programming is presented in order to establish the bounds of the programming task.

Second, the characteristics of a well-designed program and their importance are outlined. It is desirable that any method for designing programs be concerned with the quality of the program that is to be produced and, thus, with these characteristics.

Third, the constraints on programming are considered. These constraints affect the difficulty of the programming task as well as the design quality of the program.

Then, a brief survey of the method and its relationship to the desired program characteristics are given. This is followed by the detailed specification of the method for designing programs.

2.1 Definition of Programming

Programming is defined as the transformational process by which a human, when given a programming problem, produces a computer program which is a solution to the problem. A programming problem requires that a program be written which will realize some function. Ideally, the statement of the programming problem, i.e., the problem definition, defines the function so that there is explicit pairing of inputs and outputs for the program. In this ideal case, the programmer transforms

the functional definition into a program. However, this ideal situation rarely exists. It is often the case that at least one of the following is not well-defined in the problem definition:

1) inputs

2) outputs

3) relationship between inputs and outputs

In this case, the establishment of a well-defined functional definition with its attendant inputs and outputs is an integral part of programming.

For the purposes of this paper, it is assumed that the subject matter of the programming problems is unlimited; however, the size and complexity of the problems are assumed to be in the range that can be solved by one person. No specific programming language is recommended for program construction, although it is expected that the higher-level languages will be used primarily.

This paper is not concerned with those aspects of programming which are specifically related to team and contract programming. Therefore, discussion of management procedures, problem clarification procedures, and extensive and formal documentation procedures that are related to these projects is not included.

2.2 Characteristics of a Well-Designed Program

There are five main characteristics which affect the design quality of a program. These characteristics are correctness, efficiency, ease of debugging, adaptability and readability.

Correctness is the most important of these characteristics. If a program does not represent the desired function, i.e., if it does

not produce the correct output for each input, it is not a solution to the given programming problem. The presence of this characteristic is necessary for any program.

The other four characteristics are especially important for programs which fall into these categories:

1) large and complex
2) intended for repeated use
3) intended for use by many people

For programs which fall into the opposite categories, these characteristics are relatively less important; however, they are still applicable. The relative importance of these four characteristics is discussed below.

Debugging is a time consuming process for both human and computer. The ease with which a program can be debugged directly affects both the time and effort required in producing a correct program. Therefore, ease of debugging is a desirable characteristic for any program, although it is relatively more important for those which are large and complex.

Program efficiency is especially important for those programs that will be used repeatedly. For such programs, keeping the cost as low as possible for each program run is desirable. For programs which will be run only a few times, the increased cost per run of the program must be weighed against the increase in the overall cost of program production that will result from the extra programmer time needed to increase efficiency. This is not to say that efficiency considerations should be ignored unless the program is to be used many times; however, program efficiency is a characteristic whose importance is relative to the specific program and its intended use.

Readability is most important for those programs that will be used by many people, some of whom may require that modifications be made in the program during its lifetime, and for those programs that are large and complex. In general, any program that is going to be examined by someone, for whatever reason, needs to be readable. Those programs that may be modified by someone other than the original programmer must have adequate documentation, and this includes internal documentation reflected both in commentary and in structure. This does not exempt the program from needing to be readable even if it is intended for the private use of the programmer. He, too, is going to need to read the programm. This is especially true during the testing and debugging of the program. Readability, for any size of program, can reduce the effort for the programmer during this phase of programming.

For the program that is used over a period of time during which it will be altered for a slightly different purpose or for a different system, adaptability becomes an important characteristic. All programs are modified to some degree during their lifetimes, if only during testing and debugging. Therefore, all programs are subject to considerations of adaptability.

2.3 Constraints on Programming

There is a basic set of constraints within which a programmer must work regardless of the particular program he is working on. This set of constraints consists of

1) the programming problem

2) the programming language

3) the computer system

4) the knowledge and psychological characteristics of the programmer

Each of these constraints contributes to the amount of difficulty encountered in creating a program with the desired characteristics. Each affects the design of the program to some degree as well. Although they influence the design process, they do not modify the overall pattern of programming activity.

The programming problem

Obviously, the more difficult the problem is, the more difficult the programming process becomes, both in finding a solution plan and in inplementing that plan as a program.

One aspect of the problem that can increase programming difficulty is the representation of the problem as it is given. The representation of a problem may be traditional for its type; however, this traditional representation may not readily lend itself to the programming environment. In such a case, the programmer is faced with accepting the representation of the problem as it is given or with finding an equivalent alternate representation which can be more easily implemented.

Another aspect of the problem which complicates the programming process is the clarity of the statement of the problem. The more ambiguous the statement, the more assumptions the programmer will need to make about various facets of the problem. These assumptions influence the decisions made throughout the design process and thus influence the design of the program itself.

Programming languages

Programming languages directly affect the way that a problem is conceptualized because programming languages shape the thought processes, not only for small details, but also for the larger aspects of program organization, program processes, and data organization [Weinberg, 1971, p. 238]. (This is obviously so when comparisons are made of programs written in different languages which are designed to be solutions to the same problem.) For example, if only one language is available, and it provides only simple variables and arrays as allowable data structures, then a programmer will tend to consider representations of the problem which lend themselves to the available data structures; otherwise, he will have to implement special data routines himself. For instance, he may think that list structures better represent some aspect of the given problem. He is then faced with the choice of developing list structure representation and list processing routines so that he can use list structures or of abandoning list structures for the data structures already available. It is likely that he would not even consider data structures other than those available in the language, thus limiting himself to a smaller set of problem representations. If other languages are available which offer a choice of data and program structures, then a larger, more varied, set of problem representations and subsequent solution plans are possible. Therefore, both the number of languages available and the characteristics of the individual languages restrict the options that can be considered during the design process.

Computer system

The computer system is a constraint on the programming task in various ways. Among the factors that contribute to the limitations of the system are the following:

1) the availability of various languages
2) the availability of various types of input and output communication
3) the speed of computer operations
4) the size of its memory, both core and auxiliary
5) the availability of library and utility routines
6) the support of the system by the staff

The availability of several different languages allows the programmer to choose one which will work best for his specific problem. The fewer languages there are, the fewer options he has for problem representation.

The availability of several types of input and output communication allows the programmer more options for designing the input and output of his program. He also has a more varied choice for testing and debugging his program.

Concern over efficiency can be less when the system is large and fast. As the program approaches the limits of any system, however, efficiency considerations become more important, if not for the cost of the program itself, at least for increasing the priority of the program within the system.

The availability of library and utility routines can reduce the magnitude of the programming task when these routines can be incorporated into the program.

The support that the staff gives the system is, at times, as important as the power of the system itself. Unless there are adequate documentation and maintenance services available, it becomes the programmer's job to find out for himself about the problems he has with certain system facilities. He, in fact, will not be aware of the available facilities unless their descriptions are made accessible to him by the staff.

Knowledge and psychological characteristics of the programmer

The knowledge and psychological characteristics of the programmer are as important as the other constraints. How much the programmer knows about various facets of programming can make a difference in the amount of difficulty he has with the programming task and in the design quality of the programs he constructs. These facets include

1) the programming problem
2) programming constructs
3) language(s)
4) the computer system
5) psychological factors influencing programming

The knowledge that the programmer has about the problem influences the quality of the design of his program. If he can see alternative ways of representing the problem, then he has alternative solution plans available to choose from. Since the solution plan directly affects the characteristics of the problem, having alternative solution plans increases the potential for program quality. If the programmer overlooks or misunderstands a part of the problem, then the longer he remains unaware of that fact the more difficult the programming

task will become for him. At whatever point in the task that he does recognize his error or oversight, he will be forced to reevaluate everything he has done up to that point before he can proceed. Such backtracking may require extensive changes or even starting over in order to maintain design quality.

His knowledge of how to create and use programming constructs (i.e., those construction techniques which transcend any individual language), such as iteration, recursion or subprograms, limits the alternatives that the programmer has for conceptualizing the processes and data structures needed for the program. Similarly, the number of languages he knows and the degree of familiarity he has with each of them limits the alternatives he has for problem representation and program construction. By using a language with which he is already familiar, the programmer can make program construction, insofar as using the language is concerned, easier; however, by not taking advantage of a less familiar language that has features more compatible to a given problem, he restricts the potential quality of the program.

The more a programmer knows about the computer system, the more alternatives he has for the choice of a language, for incorporating library and utility routines into his program, for input and output for his program, and for testing and debugging aids. The more he knows about the intricacies of the system and how the language he uses works within the system, the better he can evaluate the effects of his program.

Increased alternatives, whether in the choice of a problem representation, a language, a programming construct or a system facility, offer the programmer a greater potential for a well-designed program

since each alternative offers a chance of greater efficiency, readability, adaptability, and ease of debugging. By carefully choosing among his alternatives, the programmer can more easily construct a well-designed program. He restricts the alternatives himself when he does not possess the necessary knowledge.

The entire programming process can become easier for the programmer if he is aware of psychological factors which can influence the process, especially as these factors relate to him personally. Among the psychological factors assumed to be relevant in programming are the aptitudes of memory — both short term and long term, deductive reasoning, and problem solving ability. Even personality factors, such as the ability to work under stress and the ability to admit mistakes, are applicable here. Some of these factors are significant for the entire programming process while others have a strong influence in only one or two parts of the process.

For example, memory is important throughout the entire process. Because programs must be expressed exactly, only very short and simple programs, can be held in human memory. Even programmers with good memories realize their shortcomings and compensate for the inability to retain the details of large programs by using written records.

Another factor influencing the programming process is psychological set. Psychological set is that phenomenon in which the categorization of a perceived pattern, either concrete or abstract, is based on factors other than those existing within the pattern itself. Set can sometimes produce undesirable results for the programmer. For example, he may have a program in which there is an error whose location is unknown. His task is to analyze

the program code, i.e., the pattern, in order to determine which part is incorrect, i.e., to categorize it. Often the programmer will be unable to find the location of the error due to his mental attitude, or set, concerning the program code. For instance, if one of the subprograms in the program had run earlier on a set of test data, he would assume that the subprogram still works and, therefore, could not be the source of the error. Yet, in fact, it might contain the error. Or, he may have invested such an inordinate amount of time on a certain part of the program that he feels that it must be correct, when it perhaps is not. In these cases it is not what the programmer actually sees when he looks at the program code and analyzes it, but it is his predisposition toward the code that allows him to categorize it incorrectly. The debugging process is especially susceptible to this kind of influence.

By recognizing how these and other factors can affect the entire programming task as well as particular parts of the task, the programmer can use these factors to his advantage whenever possible. More importantly, he can guard against those which work to his disadvantage and try to compensate for them in some way. In this way he can eliminate some of the difficulties of the programming task.

2.4 The Method for Designing Programs

The proposed method for designing programs encompasses the entire transformational process by which a human produces a program that is a solution to a programming problem. The method divides the transformational process into ten major steps. These steps are

1) Problem Definition
2) Problem Analysis
3) Solution Plan
4) Establishment of Algorithms
5) Algorithmic Description
6) Program Coding
7) Program Testing
8) Debugging
9) Documentation
10) Re-analysis

The division of the programming task into these ten steps is not intended to imply that each step is independent of the others. In fact, each step is interrelated with the other steps and some steps may be carried out in parallel. However, each step is composed of a set of related tasks which must be initiated, if not completed, before the succeeding steps can be initiated.

By dividing the transformational process into ten clearly defined steps it is possible to

1) explain the purpose of each step in relation to the entire programming task
2) explain the purpose of each step in relation to the program produced
3) explain how the tasks which must be at least initiated, if not completed, for each step influence the design of the program and its characteristics.

It is possible, therefore, to be explicit in presenting the method for designing programs to a student. He will have a limited number of steps, each with a specific set of tasks, that he can apply to any programming task. In this way the student is not left on his own to assimilate for himself an overall approach to programming. Instead he is given an approach to use. The intent in doing so is to reduce the learning level from that of "problem solving" to "principle learning" for as much of the task as is possible. In essence, the student learns a set of rules which he can apply to any programming task and is therefore freed from having to create such rules himself.

The specific details for the steps in the method for designing programs are presented as a collection of ideas for use by the teacher in adapting the method to the particular course involved. Before proceeding to the specification of the method, a brief survey of how a programmer works through the ten steps is given followed by a summary of how each program characteristic is related to the ten steps.

A survey of the steps in the method

This survey of the steps in the method is intended only to give the overall relationship of the steps in the method. Each step in the method and the relationship among the steps is considerably more complex than outlined below.

When confronted with a problem to program, a programmer must first study the problem in order to understand exactly what the problem is and to make sure that it is clearly defined. This takes place during the Problem Definition step. As his understanding of the problem becomes

clearer, he then proceeds to analyze the problem in order to separate out subproblems which are easier to solve individually than the problem as a whole; such analysis is done during Problem Analysis. While analysis of the problem and its subproblems is being done, the programmer is also searching for solution plans for the subproblems he has found. This search for solution plans for subproblems, and ultimately for the entire problem, takes place during the Solution Plan step. Once he has at least one solution plan for the problem, the programmer then creates an algorithm for the plan — this being done during the Establishment of Algorithms step. During the Algorithmic Description step, the algorithm is given a formal description from which language statements can be easily formulated during the Program Coding step. Once the entire program has been coded, it is tested for errors during Program Testing and if any are found, they are located and corrected during the Debugging step. When the program is completed, documentation can be furnished which describes the program and its purpose; this is done during Documentation. At this point the task of producing a program and its documentation is complete; however, a tenth step called Re-analysis has been included. This step is one which, while not directly affecting the program at hand, is an important one. In providing for the re-analysis of the program and its documentation, as well as of the overall process used to produce the program, this step can contribute valuable hindsight knowledge to a programmer's repertoire of program design skills and thus better prepare him for his future programming tasks.

Relation of desired program characteristics to the steps

The importance of the characteristics of correctness, efficiency, readability, adaptability and ease of debugging was previously discussed. In order to construct a well-designed program these characteristics must be planned for throughout the design process. Given below is a summary of the considerations included in each step of the method for each of the characteristics.

For program correctness, the following are considered in the various steps:

Step 1 - Problem Definition

establishing the problem definition as correctly as possible, especially the correct output

Step 2 - Problem Analysis

refinement of the desired output;
identification of a set of subproblems which encompass all parts of the given problem adequately

Step 3 - Solution Plan

verification that solution plans suggested are actually solutions

Step 4 - Establishment of Algorithms

verification that the algorithm produced represents the solution plan for the given problem;
establishment of error traps for conditions which violate algorithm invariants

Step 5 - Algorithmic Description

verification that the description used corresponds to the intended algorithm

Step 6 - Program Coding

verification that the code produced represents the algorithmic description

Step 7 - Program Testing

verification that the program produces the correct output

Step 8 - Debugging

correction of errors

Efficiency considerations are made in a relatively few steps:

Step 3 - Solution Plan

consideration of efficiency questions for the solution plan to each subproblem and consideration of efficiency questions for the solution plan to the entire problem, including overall problem structure, data structures, and algorithmic processes

Step 4 - Establishment of Algorithms

consideration of the performance characteristics of each algorithm for each subproblem and of the algorithm for the solution plan to the entire problem, including program structure, data structures, and programming constructs used to define the algorithmic processes

Step 6 - Program Coding

consideration of the language structures which make the best use of the language being used, including data structures, program structure, and processing control

Step 7 - Program Testing

determination that the program actually performs as well as expected

Readability considerations start at the very beginning of the design process. The utilization of names which reflect the contents of subproblems, solution plans, algorithms and subsequently subprograms should be carefully chosen starting in Step 1 - Problem Definition and be carried through Step 6 - Program Coding. Names which reflect the purpose of data structures as they are developed are chosen in Step 1 through Step 6 inclusive. Comments in the program body and organizational schemes, such as indentation, are included in Step 6 - Program Coding.

Adaptability considerations are related to the modularity of the program structure and to the straightforwardness of the data structures. Program structure is dependent to a degree on the conceptualization of the given problem as it is seen during Step 1 - Problem Definition. The breakdown of subproblems as carried out in Step 2 - Problem Analysis has the greatest influence on establishing the program structure. Step 5 - Algorithmic Description considers the structuring of subprograms so that some modification of the program structure may occur here.

Data structure considerations related to adaptability fall mainly in Step 3 - Solution Plan through Step 5 - Algorithmic Description. It is in Step 3 that the general descriptions of potential data structures are first considered. Step 4 and Step 5 are also important in that efficiency questions are considered so that the tradeoff between efficiency and adaptability becomes apparent here.

Debugging considerations are also related to the modularity of the program structure and the straightforwardness of the data structures, and they therefore parallel those of adaptability. Thus, Step 2 - Problem Analysis and Step 5 - Algorithmic Description contain considerations about program structure and Step 3, Step 4, and Step 5 contain considerations about data structures. Step 7 - Program Testing bears directly on ease of debugging to the extent that the test routines and organized testing schedule take advantage of program structure, thus reducing the complexity of the debugging process.

Specification of the steps in the method for designing programs

The steps in the method for designing programs were listed previously. This specification of the method includes the name and

description of each step along with the relation of each step to the other steps. The relationship between steps is expressed in two ways. First, a discussion of how the steps overlap with each other in time and second, a description of the requirements from previous steps for each step is given. The nucleus of each step is the specification of the tasks to be accomplished within the step. The specification of the tasks is embodied in a set of questions to be asked and in the structuring of the answers for use in other steps. The ordering of the tasks within any one step's specification in intended only as a guide to the order in which they are performed.

These specifications are a collection of ideas, the details of which are adaptable to any particular programming situation. While it is expected that the programmer will work through the steps in the method he will not necessarily carry out all of the specifications in detail. For example, some specifications indicate the need for written records. Some programmers may wish to keep extensive notes while others need only a short note. For some programming tasks, certain specifications are not applicable. For instance, if the language is predetermined, then those specifications concerned with the choice of a language will not apply. The steps and their specifications appear below.

Step 1 - Problem Definition

This first step in the transformational process is concerned with the clarification and understanding of the given problem. Unless the problem itself is understood, all effort on the problem may be lost. Too often major parts of a problem are misinterpreted or, worse, the

entire problem is understood to be something quite alien to the original problem. Therefore, before any effort is given to analyzing the problem or working toward the solution of specific parts of the problem, it is advisable that the problem statement in its entirety be clearly defined.

In his discussion of problem solving, Gagné [1970, p. 215] states that the main activity during problem definition is to distinguish the essential features of the problem. The "essential features" of any problem depend, to a great extent, on the context within which the problem is bound. For example, Polya [1957, p. xvi] lists three essential features to look for in solving mathematical problems. These are (1) the unknowns, (2) the data, and (3) the condition or conditions linking the unknowns and the data. Once these three features have been identified, only then can one proceed to look for a solution to the given problem.

Similar essential features exist for problems which are to be solved within the context of computer programming. Corresponding to the unknown in the mathematical context is the output from the program. Corresponding to the data is the input data for the program. And, of course, there is some relationship linking the input data and the output just as there is at least one condition linking the mathematical unknown and its associated data. The relationship between the input data and the output is ideally expressed as a function which explicitly pairs the inputs and the outputs; however, this ideal situation rarely exists. Therefore, the clarification of the relationship, or mapping, between the inputs and the outputs as well as clarification of the inputs and outputs themselves must be initiated in this step.

The main interest in program output during this step is to determine that output which is considered to be the solution to the problem (or that which in some way indicates that the program is a solution). This type of output is identified as solution output. A second type of output which is also considered is auxiliary output. This type of output, while not being the solution output itself, is necessary in order that the solution output can be properly identified. Auxiliary output may be in the form of headings, labels, or anything else which clarifies the solution output. Both the solution output and the auxiliary output are intended either for consideration by a human, a device, or another program, so the selection of output media for these may be an important factor to be considered in certain problems.

Of these two types of output, the solution output must be identified during the Problem Definition step. The auxiliary output may be tentatively identified here, but its composition will probably become clearer during the Problem Analysis and Solution Plan steps.

The input data for a program can be described as that information given in the problem upon which operations are to be performed so that the solution output will result. The input data may exist in the program in two ways. It may actually be input (read in) to the program, especially if the specific values are to change from one program run to another (e.g., payroll information). On the other hand the input data may be of such a nature that it can exist as part of the program. In this case the data can be initialized within the program if it will not change between program runs (e.g., a table of constant values). Whether the input data is to be read in or initialized is a question which is usually resolved

during the Solution Plan step; however, if any restrictions appear in the problem statement concerning this question, they should be noted during Problem Definition.

The relationship between the input and the solution output may be expressed in the problem statement in various ways. It may center around a formula or a set of explicit rules, or it may be only implied in a verbal description of the inputs and outputs. As a first step toward establishing the mapping of inputs and outputs, the relationship, in whatever manner it is expressed, should be represented as a series of broad actions or steps, which start at the input and proceed to the desired output.

Summarily, Problem Definition is concerned with establishing an understanding of the problem. This is done by distinguishing the essential features of the problem. In the context of programming these features are the input (as a given), the solution output, and the relationship between the input and the solution output.

Specifications for Problem Definition

1. What are the outputs from the program to be?
    a. List the solution outputs in general terms, along with output media, if applicable
    b. List the auxiliary outputs in general terms
    c. If the solution and auxiliary outputs are to be output in a specific pattern, illustrate it with explanation
    d. List any restrictions imposed on the output

2. What is the input data for the program?
   a. List the data to be read in, along with the input media if applicable
   b. List the data to be initialized in the program, along with the initialization process if applicable
   c. List any restrictions on the input data
3. What is the relationship between the input data and the solution output?

   Starting with the input data, list the broad actions or steps necessary to get to the desired output, i.e., establish the preliminary mapping of inputs and outputs

Step 2 - Problem Analysis

The Problem Analysis step is concerned with the exploration of the essential features outlined during Problem Definition. This exploration is intended to isolate as many of the subproblems contained within the given problem as possible. Not only are the subproblems to be found, but they too are to be explored as to the presence of other subproblems. Also, it may be possible to break up the given problem into subproblems in more than one way so that alternative sets of subproblems may exist for the given problem. Each such set will need to be explored as well. Besides determining subproblems, this step involves investigating the sufficiency of the information given in the problem to solve the problem and determining if there are any inconsistencies or omissions in the problem statement. As work proceeds on the programming task, such difficulties can arise at almost any step; however, it is in this step that such

considerations should begin and of course continue throughout all further steps. In order to accomplish both the isolation of subproblems and the investigation of problem consistency, the essential features from the Problem Definition will probably need further clarification by expansion of those features which have been described in general terms. The features should be made as explicit as possible with regard to the requirements of the program, especially the input data and the solution and auxiliary outputs.

The relationship of the Problem Analysis step to that of Problem Definition has been implied in the preceding discussion. Problem Analysis is a natural extension of the work started during Problem Definition. Problem Analysis then works directly on the results from the Specifications for Problem Definition, i.e., the specifications given for inputs, outputs, and the preliminary mapping of inputs and outputs. The dividing line between Problem Definition and Problem Analysis lacks sharpness; however, the former is more concerned with establishing what the essential features of the problem are and the latter, with the breakdown of those features into subproblems through extensive analysis.

The demarkation between Problem Analysis and its successor step, Solution Plan, is even less sharp than that between Problem Definition and Problem Analysis. In fact, there is considerable interplay between the two, since in order to decide if a potential subproblem lends itself to being solved under certain conditions, an exploration of the potential solution plan(s) must be done. This in turn can lead to the isolation of other subproblems. Therefore, the working relationship between Problem Analysis and Solution Plan is quite close; however, since some basic activities can be isolated in each step, they are discussed separately.

Specifications for Problem Analysis

1. What exactly are the outputs from the program to be?
   a. List each solution output giving its
      1) name or identification
      2) form (tabular, graphic, etc.)
      3) type (real, character, etc.)
      4) media (teletype, printer, etc.)
      5) range of values or attributes
   b. List for each solution output its associated auxiliary output giving the latter's
      1) name or identification (or format)
      2) form
      3) type
   c. If the solution and auxiliary outputs are to be output in a specific pattern, describe it
   d. List assumptions which have been made concerning the outputs, i.e., what things have been assumed from implications in the problem statement, but which are not explicit in it. This may include the following:
      1) name
      2) form
      3) type
      4) media
      5) range of values or attributes
      6) output pattern

e. Describe any relationships which exist between solution outputs and their associated auxiliary outputs, i.e., if the values or size of one solution output entity changes, describe how this change affects both its auxiliary output and the other solution outputs and their auxiliary outputs

2. What exactly is the input data for the program?

   a. List the data given in the problem to be read in, giving its

      1) name or identification

      2) form (table, list, etc.)

      3) type (real, character, etc.)

      4) media (teletype, CRT, card, etc.)

      5) range of values or attributes

   b. List the data given in the problem that is to be initialized, giving its

      1) name or identification

      2) form

      3) type

      4) range of values or attributes

      5) process used to initialize data (e.g., assignment of constant values, assignment of generated values, etc.)

   c. List assumptions made concerning the input data, i.e., what things have been assumed from implications in the problem statement. This may include

1) name

2) form

3) type

4) media

5) field specifications

6) range of values or attributes

7) process used to initialize data

d. Describe any relationships which exist between input data, i.e., if the value or length of one data entity changes in some way, describe how the change affects other input data.

3. What logical divisions, or subproblems, of the problem are evident from the problem statement and from the results of the Problem Definition step, especially concerning the preliminary mapping of input data and solution output?

a. List any problem previously solved which is analogous [Polya, 1957, pp. 37-46] or related [Polya, 1957, pp. 110-112] to the given problem giving

1) the analogous relationship

2) the breakdown of the subproblems in the analogous problem

3) the estimated degree of usefulness of the analogous problem and its subproblem breakdown with respect to the given problem

b. List at least one tentative set of subproblems for the given problem. This set may be given as a list of

1) events

2) functions

3) procedures

4) combination of 1), 2), and 3) which are expressed in a definitive way (but not necessarily so well defined as to be an algorithm) which can be used as an outline to guide further analysis. (If helpful, the list may be represented diagramatically as a flowchart when the ordering of the subproblems has been established.)

c. For each subproblem, list

1) a general description of what the subproblem is

2) its "input" and "output" in general terms not tied to a specific data representation

3) its relation to immediate predecessor(s) and successor(s) in the list of subproblems, where a predecessor is a subproblem whose solution affects the present subproblem, and a successor is a subproblem which depends upon the solution of the present subproblem

4) the mapping in the subproblem relating its inputs and outputs

d. For each subproblem, list the subproblems found in it along with the elements listed in b. above. Continue this isolation of subproblems until they are of a manageable size. It may be advisable at this point to diagram each set of subproblems, along with any alternative

subproblem configurations so that the given problem can be visualized as a set of related subproblems. This diagram may be a block type diagram similar to a Module Linkage Chart [Maynard, 1972] in which each block represents, in this case, a subproblem, the levels of the blocks reflect the different depths of the subproblems, and the lines connecting the blocks indicate which subproblems are related to other subproblems. No attempt is made in such a chart to indicate conditional relationships among the subproblems; however, if conditional relationships are desired then the flowchart with conditionals can be used [Forsythe, et. al., 1969, p. 45].

4. What inconsistencies exist in the problem statement or in the subsequent subproblems, i.e., what omissions or insufficiencies exist?
   a. Does each set of subproblems constitute all the parts of the total problem? If not, list what is lacking.
   b. Is each set of subproblems consistent with the problem statement? If not, describe inconsistencies and how they might be remedied.
   c. What assumptions have been made about the problem itself and subsequently about the subproblems in order to compensate for any omissions in the problem statement (or Problem Definition). Describe each assumption and its relation to each set of subproblems.

Step 3 - Solution Plan

The Solution Plan step is probably the most difficult step in the transformational process in that it requires that at least one solution plan to the given problem be discovered or invented. This production of a solution plan has been given a starting point by the reduction of the given problem to a set of subproblems, which, hopefully, will be easier to solve individually than the original problem as a whole. If a set of subproblems which represents the original problem is complete in that all aspects of the problem are accounted for, then it can be assumed that the set of solution plans to a set of subproblems is a solution plan for the original problem.

The general approach used to find a solution plan for any one subproblem is to explore tentative solution paths. Such exploration includes considering alternative solution processes, alternative data representations, alternative inputs and outputs for each subproblem, and even alternative representations of the subproblem itself. These latter representations may already exist from the production of various sets of subproblems during Problem Analysis, or they may become apparent during attempts to produce a solution plan for a given subproblem. Once a set of tentative solution plans for a subproblem have been found, the solution plans need to be carefully examined to determine which one or ones meet the requirements which are desired in the final product, e.g., how much processing is to be done, how much storage approximately will be needed, if the solution process is straightforward, etc. Each tentative solution plan should, of course, be checked for consistency with the problem statement and its conditions to insure that the solution plans actually solve the given problem.

At this point in the transformational process, nothing has been implied about the choice of language or computer to be used for the final program. The reason for this is that such choices can be and should be dependent upon the result of the Solution Plan step. By considering the tentative solution plans and the desired characteristics of the final product in tandem and comparing the features needed with available language and computer options, an intelligent choice can be made which will reduce the amount of work necessary to translate the solution plans into the final program. Even if the computer and the language are predetermined, alternative solution plans are useful in that by comparing such alternatives, the better solution plan can be chosen based on the desired characteristics of the final product, e.g., efficiency of processing or minimal storage space.

The representation of the various solution plans can be expressed in several forms. It can be an extension of the form used during Problem Analysis or it can be something else depending upon the preference of the programmer. Representations commonly used include flowcharts, verbal outline descriptions, decision tables, and module charts. Whichever type of representation is used, it should allow for a complete description of all phases of the solution plan; it will probably be some combination of flow diagram and verbal description.

In summary, the result of this step should be one set of solution plans for the subproblems which comprise the given problem. In order to produce one such set, alternative solution plans for each subproblem in each set of subproblems from Problem Analysis should be explored as to their suitabilities for practical implementation as well as their consistencies with the given problem.

In exploring solution paths for any subproblem it is natural to encounter new subproblems not previously envisioned during the Problem Analysis step. Because of this, it can be seen that the Problem Analysis and Solution Plan steps are not entirely sequential but in fact are alternating activities. As new subproblems are encountered and recognized as such, the exploration of solution paths becomes necessary. Therefore, the Solution Plan step requires the isolation of some subproblems during the Problem Analysis step and expects that at least one set of such subproblems exists. During the quest for solution plans for the subproblems a form of problem analysis is ongoing so that new subproblems are added to those previously found during the initial phase of Problem Analysis. These new subproblems are explored for solution plans and the processes continue until all subproblems have been solved in some way.

Specifications for Solution Plan

1. What solution plan(s) exist for each subproblem given its general description, its inputs, its outputs, the mapping of its inputs and outputs, and its relation to its successors and predecessors?
    a. List any problem previously encountered and solved which is analogous to or is related to each subproblem giving
        1) the method used to solve the associated problem
        2) the applicability of the solution and the solution plan for the given subproblem
    b. List each tentative solution plan for each subproblem giving its complete description as far as possible including the

1) assumed data representations internal to the subproblem
2) assumed representations for the inputs
3) assumed representations for the outputs
4) assumed mapping of the inputs and outputs of the subproblem
5) valid ranges of values or valid attributes of the data representations
6) verbal and/or diagrammatic general descriptions of the solution process

c. For each solution plan list any new subproblem that has been encountered along with its
1) general description
2) inputs
3) outputs
4) mapping of inputs and outputs
5) relation to predecessors/successors

2. How "good" is each solution plan for each subproblem, and which is the best one in the context of solving the problem practically?

a. For each solution plan, answer the following questions
1) Approximately how many times will the steps in the solution process be executed?
2) Approximately how much storage is required for the data representations used internally (to the subproblem) as well as the process itself?
3) How straightforward is the solution process?

4) Can the range of values anticipated to occur in the solution process be computed on the computer(s) being used?

5) Does the solution process depend upon any external (to it) information other than the given inputs to it?

6) Does the solution plan correctly solve the subproblem?

7) Does the proposed solution plan mesh with the solution plans proposed for its predecessor subproblems, i.e., those subproblems whose solution affects this one in some way?

8) What type of language and/or computer characteristics are envisioned as being either necessary or convenient to have in order to implement the solution?

9) If computer and language are predetermined, then do they possess the necessary characteristics to implement the solution plan? If not, can the characteristics be created in the language?

b. For each set of proposed solution plans for each subproblem, pick out the best solution plan based on the answers to the above questions so that it will be compatible with its predecessors and successors, which are also to be the best solution plans based on the above questions.

c. For each set of subproblems assemble the best set of solution plans.

3. If there is more than one set of subproblems which represent the given problem, then which solution plan set is the best total solution plan for the given problem?
   a. For each set of solution plans answer the following questions:
      1) Which set of solution plans requires the least amount of storage for both data representations and process representations?
      2) Which set of solution plans has the least number of steps?
      3) Which set of solution plans has the least number of step executions?
      4) Which set of solution plans exhibits the greatest amount of independence among the individual solution plans (independence here means no connectivity between solution plans except through defined inputs and outputs), i.e., modularity?
      5) Which set of solution plans seems to be the most straightforward?
      6) Does each set of solution plans correctly solve the given problem with its defined inputs, outputs, and mapping between the inputs and outputs?
      7) If the language and computer are predetermined, then can each set of solution plans be implemented using existing characteristics of the language and computer? If so, how difficult does the implementation

appear to be? If not, can the necessary characteristics be created; and if they can, how difficult does their creation and subsequent implementation appear to be?

8) If a choice of languages is available, which language(s) best fit each set of solution plans, i.e., which language(s) possess the necessary characteristics to allow implementation of the total solution plan either directly or with a minimum of created characteristics?

b. Determine the one total solution plan to be used on the answers to the above questions. There should be at least one such total solution plan to the given problem.

c. Determine which language (or languages) is to be used for the program by evaluating the answers to the following questions:

1) Which language possesses the characteristics (i.e., processing features, data structures, program structures) which match the needs of the proposed total solution plan the closest?

2) Which language possesses the characteristics from which the necessary features for the solution plan can be created with the least amount of difficulty?

Step 4 - Establishment of Algorithms

The Establishment of Algorithms step in the transformational process requires that the proposed solution plan developed during the

Solution Plan step be expressed algorithmically. Algorithmic expression, after Knuth [1969, pp. 4-6], means that the solution plan is to be expressed as a finite set of rules which gives a sequence of operations that will solve the given problem. The set of rules should exhibit the following features:

1) finiteness - the algorithm must terminate after a finite number of steps

2) definiteness - each step in the algorithm is to be precisely defined so that all required actions are unambiguous and rigorous with respect to each case to which it is applied

3) input - there is to be zero or more quantities from a specific set of objects which are given to the algorithm before it begins

4) output - there is to be one or more quantities resulting from the action of the algorithm which have a specified relation to the inputs

5) effectiveness - all operations in the algorithm must be so basic that they can be, in principle, done exactly and in a finite length of time by a person using pencil and paper

Regardless of the manner in which the proposed solution plan has been expressed as the result of the Solution Plan step, it must be expressed algorithmically so that it can be transformed into a program. In order to do this, each solution plan for each subproblem needs to be established as an algorithm so that the entire solution plan for the given problem will be in the required form.

The goal in this step is to establish a complete algorithmically expressed solution plan for the given problem. Therefore, this step expects to receive a complete solution plan for the given problem, where the expression of the solution plan as given in the Solution Plan step may range from a general description of the solution processes to a precise description which may be almost algorithmically expressed.

In case there is some question about the kinds of solution processes which can be expressed algorithmically, it does not matter whether the processes are in fact algorithms in that they always produce the desired answers or are heuristics in which the desired answers may or may not be produced, as long as the result of the proposed solution process is acceptable to the problem solver. It is expected that the proposed solution plan resulting from the Solution Plan step may be some type of heuristic process. This kind of process, if it is to be transformed into a program, must be expressed as an algorithm whether or not it produces the desired answer every time; as long as the heuristic process provides some means of expressing the lack of an answer, it can be given the algorithmic expression necessary for translating it into a computer program.

The Establishment of Algorithms step is a natural extension of the Solution Plan step in that there may be no discernable break between them in actual practice; however, there is a difference between being able to create a solution plan for a problem and being able to give an algorithmic expression of that plan. Thus the Solution Plan step and the Establishment of Algorithms step have been treated separately.

Specifications for the Establishment of Algorithms

1. What is the algorithmic expression for the proposed solution plan established during the Solution Plan step?
    a. For every solution plan corresponding to the subproblems representing the given problem, establish it as an algorithm, i.e., as a finite set of rules giving a sequence of operations where the following features are present:

1) finiteness

2) definiteness

3) input

4) output

5) effectiveness

Besides developing the algorithm, include the following information about it:

1) specify the inputs as to number, type, range of values, etc.

2) specify the outputs as to number, type, range of values, etc.

3) specify the mapping of the inputs and the outputs

b. Combine all algorithms for the solution plans for all the subproblems so that the complete solution plan for the given problem is an algorithm which exhibits the features listed above. Include information specifying the inputs, the outputs, and the mapping of the inputs and the outputs.

2. Does the complete algorithm provide a correct solution to the given problem?

a. Determine for all algorithms corresponding to the subproblems if each such algorithm expresses a solution to its corresponding subproblem by

1) showing that it produces the correct outputs for all expected inputs; do this by tracing through the algorithm using critical inputs (maximums, minimums, etc.)

2) showing what happens when it receives unexpected inputs; do this by tracing through the algorithm using values outside the range expected by the algorithm

b. Determine for the complete algorithm that it provides a solution to the given problem by

1) establishing that all inputs and outputs for each algorithm corresponding to a subproblem are correctly positioned in the subproblem structure

2) showing that it produces the correct outputs for all expected inputs

3) showing what happens when it receives unexpected inputs

3. What are the performance characteristics of the complete algorithm?

a. For each algorithm corresponding to a subproblem

1) indicate where the error-prone points are in the algorithmic process

2) establish how many times the steps in the algorithm will be executed

3) establish that the estimated execution time and storage requirements for the algorithm are within reasonable limits

4) establish under what conditions the algorithm will break down

5) establish under what conditions the algorithm becomes too costly

b. For the complete algorithm

   1) indicate where the error-prone points are

   2) establish the approximate amount of storage needed for the algorithm proper and its data structures

   3) establish an estimate of the time needed to execute the entire process

   4) indicate under what conditions it will break down

   5) establish under what conditions its performance becomes too costly

c. If any part of the complete algorithm appears that it will not perform at the desired level, then that part should be changed before proceeding further.

d. At those points in the algorithm which are susceptible to errors, e.g., ill-defined input, establish error traps as part of the algorithm.

Step 5 - Algorithmic Description

The Algorithmic Description step in the transformational process is concerned with giving the complete algorithm (established during the Establishment of Algorithms step) a description suitable for use in developing the program structure and the program code for the desired program. This description should, of course, accurately reflect the processes defined in the algorithm by preserving the operations defined and the paths through the steps in the algorithm. The description should also be in a notation which can be readily used to reflect the logical organization of the algorithm and, ultimately, the organization of the program itself. The description may be some combination of

1) flow diagrams

2) verbal descriptions, e.g., after Knuth [1969, pp. 2-4]

3) decision tables

Such a description will be called a flowchart in further discussions.

The degree of detail required by this step is difficult to define precisely; however, it can be broadly described as being that amount necessary for the programmer to organize and code the program. This is not to imply that the degree of detail required in the flowchart will be of the same level throughout. In fact, it will probably be a multi-level chart in which the highest level will encompass the entire program structure in a broad sense, where the subprograms are only seen as "black boxes." Lower level flowcharts will give greater detail for black boxes until the degree of detail is that required.

After the flowchart has been created, it should be carefully surveyed to insure that it agrees at every step with the algorithm it describes; the flowchart itself should be examined to see that it, with the included subprogram organization, gives a solution to the given problem. Those areas which were identified earlier as being error prone should be marked for incorporation of special testing processes. It is at this time, when the flowchart is tentatively in final form, that all work produced so far should be reviewed in an effort to locate slips and logical errors; once the coding of the program begins, the location and correction of errors becomes more difficult.

The Algorithmic Description step expects to take the complete algorithm produced during the Establishment of Algorithms step and produce a flowchart which, while reflecting the defined processes in the algorithm,

will be useful as a guide to the organization and coding of the program to be produced. The objective of this step is to represent the total program organization as a flowchart of some type which will provide a guide for program construction. It should be noted that flowcharting as an aid to programming is not restricted to this one step. The description of this step should not be taken as an implication that flowcharting is only used at this point; it is useful throughout the programming process and should be used whenever it is needed.

Specifications for Algorithmic Description

1. What representation is to be given to the complete algorithm?
    a. Does the complete algorithm with its sub-algorithm structuring (corresponding to the subproblem structure) possess the organizational characteristics desired for implementation of the sub-algorithms as subprograms in the program to be produced?
        1) Group together those sub-algorithms which form a logical function or set of related logical functions as a subprogram
        2) Group together those sub-algorithms which are concerned with one particular data structure or with one set of related data structures as a subprogram
        3) Produce a high level flowchart which gives the relationship among the subprograms and the main program giving the following information (subprograms are to be represented as black boxes):

a) a description of the main program with its defined inputs and outputs

b) the flow of control among the subprograms giving the inputs and outputs for the subprograms (i.e., parameters) along with their general descriptions

b. Does the high level flowchart reflect the intent of the complete algorithm?

1) Trace through the flowchart using the defined inputs in order to determine that the flow of control through the subprogram structure does produce the required output.

2) For those parts of the flowchart which do not represent subprograms, i.e., those parts of the main program which carry out some function other than subprogram calls, determine that the detail is sufficient for translation into code.

2. What representation is to be given to the subprograms?

a. Produce a flowchart to the desired level of detail for each subprogram describing

1) its inputs

2) its outputs

3) its relationship to the main program, i.e., where it can be called from and where it returns control to

4) its relationship to other subprograms by establishing

a) which ones it calls

b) which ones call it

c) the communication between calling and called as to inputs and outputs

5) the inner workings of the algorithm which it represents; the subprograms it calls may be at the black box level

b. Determine that the flowchart for each subprogram reflects the intent of the represented algorithm by tracing through the flowchart with inputs defined for it to determine that it produces the proper outputs; also, use undefined inputs to determine its performance under undefined conditions.

3. What parts of the flowchart are error prone?

a. Locate and mark each place which was found earlier in the Establishment of Algorithms step.

b. Locate any new places which have become apparent during the flowcharting process.

c. For each place so marked describe the potential error conditions by giving

1) values which could cause such conditions

2) reasons such values could exist

d. For each place so marked detail a process (or routine) that will test for such errors which can

1) be inserted into the program as a

a) permanent error trap

b) temporary error trap for use during testing only

2) be used as a test routine (include test data, if necessary) during testing

4. Are there any errors in the work produced so far?

Check the product of each step in the transformational process to make sure that

a. each product is consistent with its predecessor products from earlier steps

b. that the given problem is the one which is being solved

Step 6 - Program Coding

Program Coding is the step in which the flowchart is translated into program statements. This translation process involves

1) recognizing programming structures in the flowchart, e.g., loops, data structures, etc.

2) creating at least one corresponding language structure from the language concepts, e.g., variables, statements, etc.

3) choosing the best language structure

4) coding it

Programming structures are those concepts which can be generalized over many languages, although no one language can represent all such programming structures. For example, a loop is a programming structure, as is a subprogram, a variable, and a conditional. An iterative process and a recursive process are also programming structures. There are many ways of creating these structures in different programming languages, and certain programming structures can be represented in more than one way in any one language. Therefore, before coding can actually begin on a program,

the programmer must first recognize the programming structures in the flowchart. He must also be able to represent the programming structures in the language by combining the language concepts in specific ways. If there is more than one way, he needs to choose the one which fits best with the rest of the program code. Then, he must code the entire program.

It is advantageous to start the coding at the lowest level subprograms, i.e., those programs which are called by other subprograms or the main program but that do not call any themselves, and then proceed upward through the levels of subprograms until the main program is reached. This is in order that the running and debugging of the program can proceed efficiently. Each section of code, corresponding to a subprogram or other logical division, should be carefully checked to see that every part, no matter how small, corresponds exactly to the flowchart. Once the entire program is coded it should be thoroughly checked to insure that all the subprograms and the main program will communicate properly. Those places marked on the flowchart as being error prone should be marked on the coded program in anticipation of the incorporation of testing code to be produced. Those test routines and test data should then be coded as well.

Specifications for Program Coding

1. What code represents the flowchart?
    a. Determine the programming structures in the flowchart,
        e.g., type of processes - iterative, recursive
        type of subprograms - procedure, function
        type of data structures - simple, arrays
        type of values - characters, numeric

b. Create one or more language structures to correspond with the programming structures by combining language concepts such as variables, constants, assignment statements, etc.

c. Choose the best language structures by determining which ones

   1) represent the corresponding programming structures best

   2) make the most effective use of the language and computer

   3) mesh best with the other language structures to be coded

d. Code the program by

   1) starting with the lowest level subprogram(s)

   2) working up through the levels of subprograms until the main program is encountered

2. Does the code produced actually represent the flowchart solution?

   a. Check each step in the code to see that it corresponds to the flowchart by matching

      1) overall processes, such as subprograms, loops, etc.

      2) data structures

      3) data values

      4) total program organization

   b. Check the coded program to see that it produces the correct output for the inputs received by

1) tracing through each subprogram with its inputs and determining that the outputs are correct

2) tracing through each subprogram with undefined inputs and determining its behavior under undefined conditions

3) tracing through the entire coded program with the defined inputs to determine that the required solution and auxiliary outputs are produced, especially noting the communication between subprograms and main program

4) tracing through the program with undefined inputs to determine its behavior

3. Where are the error-prone areas in the coded program?
   a. Mark each place in the code which corresponds to an error-prone area marked on the flowchart.
   b. Mark any new places which have become apparent during coding.
   c. Briefly describe the reasons that they are error prone and outline a method for handling them which can be inserted in the code for testing purposes either as error traps or as part of test routines.
   d. Code those routines that are needed for initial testing purposes.
4. What should be done to clarify the coded program?
   a. Insert appropriate commentary for the main program and the subprograms which describes the purpose of each; such commentary should be set apart from the code in some way.

b. Use an indentation scheme which will set apart underlying programming constructs.

c. Organize the subprograms so that they are easily accessible for reference purposes, e.g., put related subprograms on the same page or on consecutive pages.

Step 7 - Program Testing

Program Testing is the step in which the coded program is run to determine if it does in fact produce the correct output. Program Testing must be planned before the program is run in order that testing routines can be devised which will provide adequate tests. Therefore, trouble areas are anticipated starting early in the programming process. Such anticipation is explicitly expressed in the Establishment of Algorithms step, and it is in the flowcharting and coding steps when test routines are developed along with program development as a whole.

Program Testing can be thought of as consisting of several phases. (Although the first phase should be almost complete by the time this step is reached, it is a part of the testing process and is, therefore, included as part of this step.) The phases of testing are

1) planning, in which test routines with test data are devised which will test each subprogram and the main programs at those places which are potentially error prone

2) running, in which the tests are carried out in an organized manner as independently as possible

3) devising new tests, whenever bugs are found which require program modification

When it has been determined that the program has been adequately tested, all test routines are removed and the program is tested as it is intended to be run to insure that no errors were introduced or covered up by the test routines. Upon satisfactory completion of this last testing, the program is ready for use and can be completely documented.

The Program Testing step requires the presence of both the coded program and the test routines which have been developed in earlier steps. Program Testing is also closely related to another step — Program Debugging. Testing of programs leads (usually) to the discovery of errors which must be corrected. Debugging and correcting lead to more testing, so it is apparent that a type of cyclic relationship exists between the two steps.

Specifications for Program Testing

1. Are the existing test routines adequate for testing the coded program?
    a. Survey the test routines to see that they cover all error-prone areas marked on the coded program.
    b. If there are trouble spots which do not seem adequately tested by the existing routines, revise them or create new ones.
    c. Note especially the following test areas:
        1) verification of program input
        2) verification of input to subprograms
        3) verification of output from subprograms

2. Do the results of the test runs indicate that the program gives the desired output?
   a. Run each subprogram and the main program as independently as possible from all other subprograms, each with its own test routines by
      1) starting at the lowest level subprograms, i.e., those which do not call other subprograms
      2) working up through the subprograms as the lower level subprograms are satisfactorily tested and can be used to test the higher level subprograms
      3) using dummy subprograms for those which require the results from subprograms which have not yet been completely tested
   b. When all subprograms have been tested satisfactorily as has the main program, combine them and run the program with its test routines.
   c. When satisfactory results are produced with the test routines for the entire program, remove them and test the program to see that their presence did not cover up any errors and that their removal did not introduce any errors. The testing is complete when a satisfactory result, i.e., correct output, is obtained.
   d. For each error discovered, proceed with the debugging as outlined in the Debugging step.

3. What do the results of the Debugging step require in the way of new tests, if any?
   a. If the debugging results have necessitated major changes in the program and the previous tests do not test the new code, devise a test which will.
   b. If the debugging results are covered by the present tests or have been compensated for by new tests, rerun the program as outlined above.
4. Do the results of the test runs indicate that the program runs as efficiently as expected for
   a. Each subprogram
   b. The entire program?

   If not, how can it be improved?

Step 8 - Debugging

This step in the transformational process is concerned mainly with locating and correcting errors, or bugs, in the program which are discovered during the Program Testing step. The discovery that errors exist, an activity of the Program Testing step, can be and usually is separate from the location and correction of such errors. There are four types of errors which can exist in a program; these are

1) syntactic
2) semantic
3) pragmatic
4) logic

Syntactic errors are those which violate some language rule(s) for some statement in the program, e.g., misplaced comma, misspelled variable, etc.

Semantic errors are those which violate some language rules governing a combination of statements, e.g., if the number of subscripts in a variable differs from the way in which the variable has been declared. Pragmatic errors are those in which the conceptualization of the meaning of a language structure by the programmer is different from the defined meaning of the language structure, e.g., input/output operations for arrays may cause an error if the programmer thinks that a certain operation handles the data elements by rows but the operation in fact handles them by columns. Logic errors are those errors in which the conceptualization of the solution plan is incorrect, e.g., a loop which does not execute the correct number of times due to a miscalculation by the programmer.

Syntactic errors are usually obvious due to diagnostic messages from the computer system. Exceptions include those errors which are syntactically correct but are not correct in the context of the program. For example, a keypunch error in which a plus sign is punched instead of a minus sign in an expression will produce an incorrect result during the execution of the program but will not necessarily be incorrect syntactically. This latter type of error is usually discovered because the program produces incorrect results.

Semantic errors, like syntactic errors, are usually found due to diagnostic messages. Again, the exceptions to this are usually discovered because the program produces incorrect results.

Pragmatic errors and logic errors are both difficult to find. Both of these types of errors can be syntactically and semantically correct and therefore produce no diagnostics. Again, it is usually only through incorrect results that the presence of these errors is discovered.

There are basically, then, two ways in which errors can be discovered. One is by the diagnostic messages given during compilation and during execution. The other is by discovering incorrect results. Errors pointed out by diagnostic messages are usually the easiest to find and correct. Those errors which are discovered as a result of observing that incorrect results have been produced are usually more difficult to find, although their correction may be relatively easy. In order to locate where such an error occurs, traces through the program, either by hand or by computer printouts, are usually necessary. Such tracing can be quite costly in terms of both human time and computer time. It is for this reason that testing should be carried out in stages as described earlier in an effort to eliminate extended error tracing.

The correction of errors seems to be of two basic types. One is a straightforward correction of a small error, e.g., a keypunch error, in which only a few statements are involved and the logic of the program is correct — this includes most syntactic and semantic error corrections. The other type is more serious and involves a revision of several statements or more in the program code; this usually reflects a pragmatic or logic error. In corrections where extensive revision is necessary, extreme care must be taken to insure that new errors are not introduced while the attempt is being made to correct the first error. Such corrections may in fact require a complete reevaluation starting with Problem Analysis in order that a logic error can be corrected throughout the entire program organization. Whenever any correction is made, the program should be retested to insure that the corrections are correct.

The Debugging step, then, is primarily an error location and correction process. Whenever errors are discovered during Program Testing, as a result of either diagnostic messages or incorrect results, debugging is necessary; and whenever corrections are made, testing is again necessary. Thus, the Program Testing and Debugging steps are interdependent.

Specifications for Debugging

1. Where is the error located?
    a. If there is a diagnostic message and it is
        1) a compile time error message, then
            a) determine what errors can cause such a message
            b) locate the error by looking for occurrences of the possible errors
        2) a run time error message, then
            a) if the message indicates the approximate location, proceed as in a) above
            b) if the message does not indicate the location then determine what errors can cause such an error message, and trace through the program either by hand or by inserting trace printouts in order to locate the error
    b. If there was no message, then what results are incorrect?
        1) Where were the results calculated and at what point were they output? The point of output can give an approximate location of the error. The point where calculations were initiated can also give an approximate location of the error.