On the Design and Development of Program Families

DAVID L. PARNAS

TECHNICAL LIBRARY
SINGER COMPANY
SIMULATION PRODUCTS DIVISION
BINGHAMTON, NEW YORK 13902

Abstract—**Program families are defined (analogously to hardware families) as sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members. The assumption that, if one is to develop a set of similar programs over a period of time, one should consider the set as a whole while developing the first three approaches to the development, is discussed. A conventional approach called "sequential development" is compared to "stepwise refinement" and "specification of information hiding modules." A more detailed comparison of the two methods is then made. By means of several examples it is demonstrated that the two methods are based on the same concepts but bring complementary advantages.**

Index Terms—**Information hiding modules, module specifications, program families, software design methodology, software engineering, stepwise refinement.**

Introduction

We consider a set of programs to constitute a *family*, whenever it is worthwhile to study programs from the set by *first* studying the common properties of the set and *then* determining the special properties of the individual family members. A typical family of programs is the set of versions of an operating system distributed by a manufacturer. While there are many significant differences between the versions, it usually pays to learn the common properties of all the versions before studying the details of any one. Program families are analogous to the hardware families promulgated by several manufacturers. Although the various models in a hardware family might not have a single component in common, almost everyone reads the common "principles of operations" manual before studying the special characteristics of a specific model. Traditional programming methods were intended for the development of a single program. In this paper, we propose to examine explicitly the process of developing a program family and to compare various programming techniques in terms of their suitability for designing such sets of programs.

Motivation for Interest in Families

Variations in application demands, variations in hardware configurations, and the ever-present opportunity to improve a program mean that software will *inevitably* exist in many versions. The differences between these versions are unavoidable and purposeful. In addition, experience has shown that we cannot always design all algorithms before implementation of the system. These algorithms are invariably improved experimentally after the system is complete. This need for the existence of many experimental versions of a system is yet another reason for interest in "multiversion" programs.

It is well known that the production and maintenance of multiversion programs is an expensive problem for software distributors. Often separate manuals and separate maintenance groups are needed. Converting a program from one version to another is a nontrivial (and hence expensive) task.

This paper discusses two relatively new programming methods which are intended explicitly for the development of program families. We are motivated by the assumption that if a designer/programmer pays conscious attention to the family rather than a sequence of individual programs, the overall cost of development and maintenance of the programs will be reduced.[1] The goal of this paper is to compare the methods, providing some insight about the advantages and disadvantages of each.

Classical Method of Producing Program Families

The classical method of developing programs is best described as *sequential completion*. A particular member of the family is developed completely to the "working" stage. The next member(s) of the family is (are) developed by modification of these working programs. A schematic representation of this process is shown by Fig. 1. In this figure a node is represented as a circle, if it is an intermediate representation on the way to producing a program, but not a working program itself. An X represents a complete (usable) family member. An arc from one node to another indicates that a program (or intermediate representation of a program) associated with the first node was modified to produce that associated with the second.

Each arc of this graph represents a design decision. In most cases each decision reduces the set of possible programs under consideration. However, when one starts from a working program, one generally goes through a reverse step, in which the set of possible programs is again increased (i.e., some details are not decided). Nodes 5 and 6 are instances of this.

When a family of programs is produced according to the above model, one member of the family can be considered to be an ancestor of other family members. It is quite usual for

Manuscript received November 3, 1975.

The author is with the Research Group on Operating Systems I, Fachbereich Informatik, Technische Hochschule Darmstadt, Darmstadt, West Germany.

[1] Some preliminary experiments support this assumption [1], [2], but the validity of our assumption has not yet been proved in practice. Readers who do not want to read about programming techniques based on this unproved assumption should stop reading here.

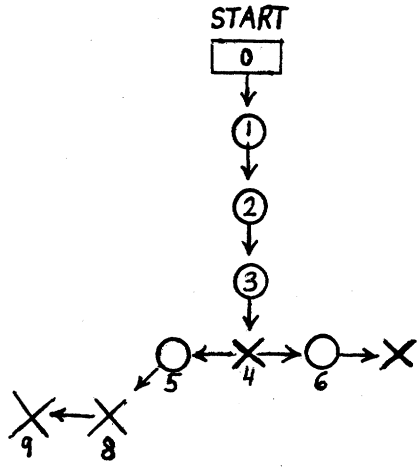

Fig. 1. Representation of development by sequential completion. Note: nodes 5 and 6 represent incomplete programs obtained by removing code from program 4 in preparation for producing programs 1, 8, and 9. *Symbols:* □ is the set of initial possibilities; ○ is the incomplete program; X is the working program.

descendants of a given program to share some of its ancestor's characteristics which are not appropriate to the purpose of the descendants. In bringing the earlier version to completion, certain decisions were made which would not have been made if the descendant program had been developed independently. These decisions remain in the descendant program only because their removal would entail a great deal of reprogramming. As a result, later versions of the program have performance deficiencies, because they were derived by modifying programs designed to function in a different environment or with a different load.

New Techniques

Fig. 2 shows the common basic concept of newer methods. Using these methods one never modifies a completed program to get a new family member; one always begins with one of the intermediate stages and continues from that point with design decisions, ignoring the decisions made after that point in the development of the previous versions. Where in the classical method one can say that one version of the program is the ancestor of another, here we find that the two versions have a common ancestor [3].

The various versions need not be developed sequentially. If the development of one branch of the tree does not use information from another branch, the two subfamilies could be developed in parallel. A second important note is that in these methods the order in which decisions are made has more significance than in the classical method. Recall that all decisions made above a branch point are shared by all family members below that point. In our motivation of the family concept we emphasized the value of having much in common among the family members. By deciding as much as possible before a branch point, we increase the "similarity" of the systems. Because we know that certain differences must exist between the programs, the aim of the new design methods is to allow the decisions, which can be shared by a whole family, to be made before those decisions, which differentiate family members. As Fig. 2 illustrates, it is meaningful to talk of subfamilies which share more decisions than are shared by the whole family.

If the root of the tree represents the situation before any decisions are made, then two programs, which have only the root as common ancestor, have nothing in common.

We should note that representing this process by a tree is an oversimplification. Certain design decisions can be made without consideration of others (the decision processes can be viewed as commutative operators). It is possible to use design decisions in several branches. For example, a number of quite different operating systems *could* make use of the same deadlock prevention algorithm, even if it was not one of the decisions made in a common ancestor.

Representing the Intermediate Stages

In the classical method of producing program families, the intermediate stages were not well defined and the incomplete designs were not precisely represented. This was both the cause and the result of the fact that communication between versions was in the form of completed programs. If either of the two methods discussed here is to work effectively, it is necessary that we have precise representations of the intermediate stages (especially those that might be used as branch points). Both methods emphasize precision in the descriptions of partially designed programs. They differ in the way that the partial designs are represented. We should note that it is not the final version of the program, which is our real product (one seldom uses a program without modification); in the new methods it is the well-developed but still incomplete representation that is offered as a contribution to the work of others.

Programming by Stepwise Refinement

The method of "stepwise refinement"[2] was first formally introduced by Dijkstra [3] and has since been further discussed by a variety of contributors [4]-[6]. In the literature the major emphasis has been on the production of correct programs, but the side effect is that the method encourages the production of program families. One of the early examples was the development of a program for generation of prime numbers in which the next to the last program still permitted the use of two quite different algorithms for generating primes. This incomplete program defined a family of programs which included at least two significantly different members.

In "stepwise refinement" the intermediate stages are represented by programs, which are complete except for the implementation of certain operators and operand types. The programs are written as if the operators and operands were "built in" the language. The implementation of these operators in the actual language is postponed to the later stages. Where the (implicit or explicit) definition of the operators is sufficiently abstract to permit a variety of implementations, the early versions of the program define a family in which there is a mem-

[2]The reader should note that although stepwise refinement is often identified with "goto less programming," the use and abuse of the goto is irrelevant in this paper.

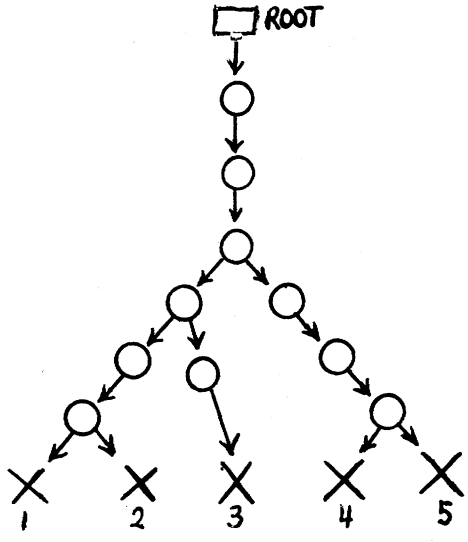

Fig. 2. Representation of program development using "abstract decisions." *Symbols:* □ is the set of initial possibilities; ○ is the incomplete program; X is the working program.

ber for each possible implementation of the unimplemented operators and operands. For example, a program might be written with a declaration of a data type stack and operators push and pop. Only in later versions would the stack representation and procedures to execute push and pop be introduced. We illustrate the technique of stepwise refinement with two examples, which will be used in a later comparison.

Example 1), Dijkstra's Prime Program: Dijkstra [3] has described the development of a program to print numbers. The first step appears as follows:

```
begin variable table p;
        fill table p with first thousand prime numbers;
        print table p;
end
```

In this program Dijkstra has assumed an operand type "table" and two operators. The representation of the table, the method of calculating the primes, and the printing format are all left undecided. In fact, the only binding decisions (common characteristics of the whole family of programs) are that *all* the primes will be developed before *any* are printed, and that we will always want the first thousand primes. Dijkstra then debates between implementing table or elaborating "fill table." Eventually he decides that "table" should be implemented, and all members of the remaining family share the same table implementation. A branch of the family with an alternative table implementation is mentioned, but not developed. Later members of the family are developed by considering various possible methods of computing the prime numbers.

Example 2), Wulf's KWIC Index Program: Wulf [5] presents a proposed stepwise refinement development of a KWIC index production program as follows:

Step 1: PRINTKWIC

We may think of this as being an instruction in a language (or machine), in which the notion of generating a KWIC index is primitive. Since this operation is not primitive in most practical languages, we proceed to define it:

```
Step 2: PRINTKWIC: generate and save all
                   interesting circular
                   shifts

                   alphabetize the saved
                   lines

                   print alphabetized lines
```

Again we may think of each of these lines as being an instruction in an appropriate language; and again, since they are not primitive in most existing languages, we must define them; for example:

```
Step 3a: generate and save all interesting
         circular shifts:

            for each line in the input do
               begin
               generate and save all inter-
                 esting shifts of "this
                 line"
               end
etc.
```

For purposes of later comparison, we note the decisions that must be shared by the remaining members of the family:

1) all shifts will be stored;
2) all circular shifts will be generated and stored before alphabetization begins;
3) alphabetical ordering will be completed before printing is started;
4) all shifts of the one line will be developed before any of the shifts for another line;
5) "uninteresting" shifts will be eliminated at the time that the shifts are generated.

In the best-known examples of programming by stepwise refinement the definitions of the operators have been informal. All of the published examples have been designed as tutorial examples, and the operators are kept "classical" so that one's intuitive understanding of them suffices for the correct understanding of the program development. The only exception known to the author is [11].[3] Formal definition of the operators can be included by application of the predicate insertion technique first introduced by Floyd for the purpose of program verification. As Dijkstra has suggested, we can think of the operators as "predicate transformers" (rules which describe how a predicate which describes the state of the program variables after application of the operator can be transformed into a predicate describing the state of the program variables before the operator is executed [7]).

Technique of Module Specification

Another technique for the design of program families has been described in [8], [9]. This method is distinguished from the method of stepwise refinement in that the intermediate representations are *not* incomplete programs. Instead, they are "specifications" of the externally visible collective be-

[3] In this example the method failed to produce a correct program because the intuitive understanding of the operators was too vague.

havior of program groups called modules.[4] These intermediate representations are not written in a programming language, and they never become part of the final system.

To illustrate this method we compare the development of the KWIC program described in [8], [9] with the development by stepwise refinement discussed earlier in this paper.

In the method of "module specification" the design decisions which *cannot* be common properties of the family are identified and a module (a group of programs) is designed to hide each design decision. For our example, the following design decisions were identified:

1) the internal representation of the data to be processed;
2) the representation of the circular shifts of those lines and the time at which the shifts would be computed;
3) the method of alphabetization, which would be used, and the time at which the alphabetization would be carried out;
4) the input formats;
5) the output formats;
6) the internal representation of the individual words (a part of decision 1).

To hide the representation of the data in memory, a module was provided which allows its users to simply write CHAR (line, word, c) in order to access a certain character. Data were "stored" in this module by calling SETCHAR (line, word, c, d). Other functions in the module would report the number of lines, the number of words in a given line, and the number of characters in a word. By the use of this group of programs the rest of the program could be written in a way that was completely independent of the actual representation.

A module quite similar in appearance to the one described above hid the representation of the circular shifts, the time at which they were computed, even whether or not they were ever stored. (Some members of the program family reduced storage requirements by computing the character at a given point in the list of shifts whenever it was requested.) All of these implementations shared the same external interface.

Still another pair of programs hid the time and method of alphabetization. This (2 program) module provided a function ITH (i) which would give the index in the second module for the i-th line in the alphabetic sequence.

The decisions listed above are those which are not made, i.e., postponed. The decisions which were made are more difficult to identify. The design has placed restrictions on the way that program parts may refer to each other and has, in that way, reduced the space of possible programs.

The above description is intended as a brief review for those who already have some familiarity with the two methods. Those who are new to the ideas should refer to the original articles before reading further.[5]

[4] Naur has called a similar concept "action clusters" [10].

[5] For symmetry we remark that while stepwise refinement was developed primarily to assist in the production of correct programs and has a pleasant side effect in the production of program families, module specification was developed for the production of program families but helps with "correctness" as discussed in [14].

Comparison Based on the KWIC Example

To understand the differences in the techniques the reader should look at the list of decisions which define the family of KWIC programs whose development was started by Wulf. All of the decisions which are shared by the members of Wulf's family are hidden in individual modules by the second method and can therefore differentiate family members. Those decisions about sequencing of events are specified early in Wulf's development but have been postponed in the second method.

Lest one think that in the second method no decisions about implementation have been made, we list below some of the common properties of programs produced using the second method.

1) All programs will have access to the original character string during the process of computing the KWIC index.

2) Common words such as THE, AND, etc., would not be eliminated until the output stage (if ever).

3) The output module will get its information one character at a time.

The astute reader will have noted that these decisions are not necessarily good ones. Nonetheless, decisions have been made which allow work on the modules to begin and progress to completion without further interaction between the programmers. In this method the aim *of the early work* is not to make decisions about a program but to make it possible to postpone (and therefore easily change) decisions about the program. Later work should proceed more quickly and easily as a result [1].

In the stepwise refinement method we progressed quickly toward a relatively narrow family (limited variations in the family). With modules we have prepared the way for the development of a relatively broad family.

Comparative Remarks Based on Dijkstra's Prime Program

We now take a second look at the Dijkstra development of the prime number program.

In his development Dijkstra is moved to make an early decision about the implementation of TABLE in order to go further. All members of the family developed subsequently share that implementation. Should he decide to go back and reconsider that decision, he would have to reconsider all of the decisions made after that point. The method of module specification would have allowed him to postpone the table implementation to a later stage (i.e., to hide the decision) and thereby achieve a broader family.

Comparative Remarks Based on an Operating System Problem

We consider the problem of core allocation in an operating system. We assume that we have a list of free core areas and data that should be brought to core storage. Writing a program that will find a free spot, and allocate the space to the program needing it, is trivial. Unfortunately there are many such programs, and we cannot be certain which of them we want. The programs can differ in at least two important ways,

policy and implementation of the mechanism. By "policy" we mean simply the rule for choosing a place, if there are several usable places; by "implementation of the mechanism" we mean such questions as, how shall we represent the list of free spaces, what operations must we perform to add a free space to the list, to remove a free space? Should the list be kept in a special order? What is the search procedure? etc.

The decisions discussed above are important in that they can have a major impact on the performance of a system. On the other hand, we cannot pick a "best" solution; there is no best solution!

On the policy side there have been numerous debates between such policies as "first fit"–allocate the first usable space in the list, "best fit"–find the smallest space that will fit, "favor one end of core," "modified best fit"–look for a piece that fits well but does not leave a hopelessly small fragment, etc. It is clear to most who have studied the problem that the "best" policy depends on the nature of the demand, i.e., the distribution of the requested sizes, the expected length of time that an area will be retained, and so on.

Choosing an implementation is even more complicated because it depends in part on the policy choice. Keeping a list ordered by size of fragment is valuable if we are going to seek a "best fit" but worse than useless for a policy which tends to put things as low in core as possible.

The following "structured programming" development of such an algorithm illustrates the construction of an abstract program which has the properties of all of those that we are interested in and does not yet prejudice our choice.

```
stage 1:
   bestyet := null;
while not all spaces considered do
      begin
        find next item from list of free spaces (candidate)
        best yet := bestof (bestyet,candidate)
      end
          if bestyet = null then erroraction
      allocate (best yet); remove (best yet)
```

Strictly following the principles of writing well-structured programs we should now verify that the above is correct or write down the conditions under which we can be certain that it is correct.

Correctness Assumptions:

1) "bestyet" is a variable capable of indicating a free space; null is a possible value of this variable indicating no space.

2) "not all spaces considered" is a predicate which will be *true* as long as it is possible that a "better" space is still to be found but will be *false* when all possible items have been considered.

3) "candidate" is a variable of the same type as bestyet.

4) "find next item from list of free spaces" will assign to its parameter a value indicating one of the items on the free space list. If there are n such items on the list, n calls of the procedure will deliver each of the n items once.

5) No items will be removed from or added to the list during the execution of the program.

6) "bestof" is a procedure which takes two variables of the type of bestyet and returns (as a value of the same type) the better of the two possible spaces according to some unspecified criterium. If neither place is suitable, the value is "null," which is always unsuitable.

7) "error action" is what the program is supposed to do if no suitable place can be found.

8) "remove" is a procedure which removes the space indicated by its parameter from the list of free spaces. A later search will not find this space.

9) "allocate" is a procedure which gives the space indicated by its parameter to the requesting program.

10) Once we have begun to execute this program, no other execution of it will begin until this one is complete (mutual exclusion).

11) The only other program which might change the data structures involved is one that would add a space to the free space list. Mutual exclusion may also be needed here.

Design Decisions in Stage 1

Although this first program appears quite innocuous, it does represent some real design decisions which are best understood by considering programs which do *not* share the properties of the above abstract program.

1) We have decided to produce a program in which one is not allowed to add to the free space list *during* a search for a free space.

2) We have not allowed a program in which two searches will be conducted simultaneously.

3) We are considering only programs where a candidate is not removed from the free space list while it is being considered. Perfectly reasonable programs could be written in which the "bestyet" was not on the list and was reinserted in the list when a better space was discovered.

4) We have chosen not to use a program in which a check for possible allocation is made before searching the list. Some reasonable programs would have a check for the empty list, or even a check for the size of the largest available space before the loop so that no time would be spent searching for an optimum fit when no fit at all was possible. In our program, an assignment to "bestyet," an evaluation of the termination condition, plus an evaluation of "bestyet=null" will take place every time the program is called.

The programs omitted from the family of programs which share the abstract program of stage 1 are not significant omissions. If they were, we would not have chosen to eliminate them at such an early stage in our design. We have discussed them only so that the reader will see that writing the program of stage 1 has not been an empty exercise.

We now consider a subfamily of the family of programs defined in stage 1. In this subfamily we will decide to represent the list by a two-dimensional array in which each row represents an item in the free space list. We assume further that the first free space is kept in row 1, that the last is in row N, and that all rows between 1 and N represent valid free spaces. We make no assumptions about the information kept in each row to describe the free space nor the order of rows in the array. This allows us to write the following:

```
stage 2:
        bestyet := 0;
        candidate := 0;
   while candidate ≠ N do
      begin
         candidate := candidate +1;
         bestyet    := bestof (bestyet,candidate)
      end
   if bestyet = 0 then erroraction;

   allocate (bestyet)
   remove (bestyet).
```

We have been able to allow the variables "bestyet" and "candidate" to be integers to implement the test for "not all spaces considered" as an integer test on the value of "candidate" because of our assumptions. Our assumptions do not yet permit us to elaborate the operations on the table rows or to implement our policy decision in "best of." We cannot even implement "remove," because we do not know if we are going to allocate all of the space found or allocate only that part needed and leave the rest on the free space list.

Stage 3

We now skip several stages in a "proper" structured programming development in order to show one of the possible "concrete" family members. In this program we have decided that the entries in each row of the array will give the first and last locations of each free space and that when we allocate a space we will allocate the whole space so as to avoid having to keep track of an ever increasing set of small fragments. We also assume a policy of "best fit" which means that we pick the smallest of the suitable free spaces.

```
                bestyet := 0;
                candidate := 0;
                OLDT := ∞

 while candidate ≠ N do

      begin

         candidate := candidate +1

         T := (end (candidate)-start (candidate))
         if T ≥ request ∧ T < OLDT then begin
                  bestyet := candidate
                  OLDT := T end;
      end;
         if bestyet = 0 then erroraction;
      allocate (bestyet)
         N := N-1;
       for I := bestyet step 1 until N do begin

         END[I] := END[I+1];
         START[I] := START[I+1];
       end;
```

To understand the value of structured programming in producing programming families, we now have to consider what would happen if, instead of the program developed in stage 3, we wanted a program in which 1) we did *not* allocate the *smallest suitable* space but only that part of it that was needed and 2) we represented the free spaces by giving the *start address* and the *length* rather than *start and end* addresses. We consider making this change in two situations.

Situation 1: We wrote the program shown in stage 3 in the classical way, i.e., we wrote that program directly without writing down the intermediate stages.

Situation 2: We used the structured programming development as shown above.

In situation 1 we would have to modify the programs shown in the section in stage 3. We would have nothing else. As you can see, it would take some effort to identify which lines in the program could remain and which could or should be changed. Even on this rather simple example it would require a fairly careful study of the program to determine which changes should be made unless the person making the changes was very familiar with the program (e.g., unless he personally had just written it).

In situation 2, however, we have the option of returning to the program labeled "stage 2." All the assumptions made in stage 2 are still valid and the program itself is still valid, only incomplete. Completing the program shown in stage 2 in order to produce the new nonabstract program is as straightforward as the original modification of stage 2 to get stage 3. It can be done by someone new. In this situation the new final program is obtained not by modifying the old working program but by modifying the closest common ancestor.

If the organization in charge of maintaining the system wishes to keep both versions in active use, they can use the stage 2 documentation as valid documentation for both versions of the program and even consider some changes for both versions by studying stage 2.

This example was intended to demonstrate why structured programming is such a valuable tool for those who wish to maintain and develop families of programs such as operating systems. The reader must keep in mind that this is a small and simple example, the benefits would be even greater for larger programs developed in this way.

Although we have shown an advantage for development of program families by using structured programming we have also revealed a fundamental problem. Progress at each stage was made by making design decisions. Going back to stage 2 was possible in our case because we had in stage 2 all of those design decisions which we wanted to keep and none of those which we wanted to discard. Unless we were able to predict in advance exactly which decisions we would change and which we would keep, we are not likely to be so lucky in practice. In fact, even with the ability to see into the future, there might not be any decision making sequence which would allow us to backtrack without discarding the results of decisions which will remain unchanged. The results of perfectly valid design decisions may have to be recoded, because the code that implements those decisions was designed to interact with the code that is being changed.

It is to get around these difficulties that the division into "information hiding" modules can be introduced. Rather than continually refine step by step a single program, as is done in stepwise refinement, we break the program up into independent parts and develop each of them in ignorance of the implementation of the other. In contrast to classical programming methods, these parts are not the subprograms which are called from a main program; they are collections of subprograms.

In our example we would have a free space list module, allocation module, and a selection criterium module. The free space list module would consist of

1) the code which implemented the variable bestyet and any other variable that could represent a place in a list as well as the representation of the constant null;
2) the program "not all spaces considered";
3) the program "find next item from the list of free spaces";
4) the program "remove";
5) a program to add items to the free space list (this program is not called in the above program, but must be called elsewhere in the system and would be considered a part of the free space list module);
6) programs to give the essential characteristics of a space on the list (e.g., start and end address).

The selection criterium module would consist of

1) bestof;
2) some other programs which will be called elsewhere, such as programs to choose a victim (a space to be removed from its owner and made available).

The allocation module consists of "allocate" and other programs not discussed above. Each of these modules would have to contain an initialization section which would be called from the main program so that the additional temporary variables introduced in implementing the programs would not be visible in the main program. For some implementations of a module the initialization section would be empty, but its call would be written in the main program so that the main program would not have to be changed if the new implementation included variables which had to be initialized.

This division into modules and independent implementation will only result in a working program if the external characteristics of each module were sufficiently well specified so that the code could be written without looking at the implementation of other modules [1], [9]. This is clearly an extra effort which is not needed if only the stepwise refinement method is used. In return for this effort one would gain the ability to reverse the decision about table representation made in stage 2 without even considering the code written to implement the policy introduced in stage 3. One also gains the ability to develop the two parts of the program without any communication between the groups developing each one. This can lead to a shorter development time and the ability to develop several versions of the system simultaneously.

How the Module Specifications Define a Family

Members of a family of programs defined by a set of module specifications can vary in three principal ways.

1) Implementation methods used within the modules. Any combination of sets of programs which meets the module specifications is a member of the program family. Subfamilies may be defined either by dividing each of the main modules into submodules in alternative ways, or by using the method of structured programming to describe a family of implementations for the module.

2) Variation in the external parameters. The module specifications can be written in terms of parameters so that a family of specifications results. Programs may differ in the values of those parameters and still be considered to be members of the program family.

3) Use of subsets. In many situations one application will require only a subset of the functions provided by a system. We may consider programs which consist of a subset of the programs described by a set of module specifications to be members of a family as well. This is especially important in the development of families of operating systems, where some installations will require only a subset of the system provided for another. The set of possible subsets is defined by the "uses" relation between the individual programs [16].

Which Method to Use

By now it should be clear that the two methods are neither equivalent nor contradictory. Rather they are complementary. They are both based on the same basic ideas (see historical note which follows): 1) precise representations of the intermediate stage in a program design, and 2) postponement of certain decisions, while continuing to make progress towards a completed program.

Stepwise refinement (as practiced in the literature) encourages one to make decisions about sequencing early, because the intermediate representations are all programs. Postponement of sequencing decisions until run time requires the introduction of processes [13]. The method of module specification is not usually convenient for the expressing of sequencing decisions. (In our KWIC index project sequencing had to be described by writing a brief "structured" "Main Program," which was one of several possible ways that the modules could have been used to produce a KWIC index. It was written last!)

Stepwise refinement has the significant advantage that it does not add to the total amount of effort required to design the first complete family member. By keeping complexity in control, it usually reduces the total amount of effort. In contrast, the module specifications represent a very significant amount of extra effort. Experience has shown that the effort involved in writing the set of specifications can be greater than the effort that it would take to write one complete program. The method permits the production of a broader family and the completion of various parts of the system independently, but at a significant cost. It usually pays to apply the method only when one expects the eventual implementation of a wide selection of possible family members. In contrast, the method of stepwise refinement is always profitable.

Relation of the Question of Program Families to Program Generators

A common step taken by industrial maintainers of multiversion programmers is the construction of system generation programs. These programs are given a great deal of data describing the hardware configuration and software needs of the users. Built into the generator is a description of a large family of programs and the generator causes one member of the family to materialize and be loaded on the target hardware.

The methods described in this paper are not intended to re-

place system generators. Since these methods are applied in the design stage and generators are useful when a specific family member must be produced. Stepwise refinement and the method of module specification can simplify the work to be done by a system generation program.

System generators would be completely unnecessary if we wished to build a program which at run time could "simulate" any member of the family. Such a program would be relatively inefficient. By removing much of this variability at the time that the program is generated, increases in productive capacity are made possible.

Often a family of programs includes small members in which certain variables are fixed and larger members in which these factors may vary. For example, an operating system family may include some small members where the number of processes is fixed and other members where dynamic creation and deletion is possible. The programs developed for the larger members of the family can be used as part of the "generator," which produces a smaller member.

Concluding Remarks

Another way of comparing the two methods is to answer the following often-heard questions.

1) When should we teach structured programming or stepwise refinement to our students?

2) When should we teach about modules and specifications?

To the first question we can respond with another question: "When should we teach unstructured programming?" The second question, however, requires a "straight answer": module design specifications should only be taught to students who have learned to program well and have decided to proceed further and learn methods appropriate to the production of software packages [12].

One of the difficulties in applying the recent concepts of structured programming is that there are no criteria by which one may evaluate the structure of a system on an objective basis. Aspiring practitioners must go to a famous artist and ask for an evaluation. The "master" may then indicate whether or not he considers the system "tasteful."

The concept of program families provides one way of considering program structure more objectively. For any precise description of a program family (either an incomplete refinement of a program or a set of specifications or a combination of both) one may ask which programs have been excluded and which still remain.

One may consider a program development to be good, if the early decisions exclude only uninteresting, undesired, or unnecessary programs. The decisions which remove desired programs would be either postponed until a later stage or confined to a well delimited subset of the code. Objective criticism of a program's structure would be based upon the fact that a decision or assumption which was likely to change has influenced too much of the code either because it was made too early in the development or because it was not confined to an information hiding module.

Clearly this is not the only criterion which one may use in evaluating program structures. Clarity (e.g., ease of understanding, ease of verification) is another quite relevant consideration. Although there is some reason to suspect that the two measures are not completely unrelated, there are no reasons to assume that they will agree. For one thing, the "ease" measures mentioned above are functions of the understander or verifier, the set of programs being excluded by a design decision can be interpreted objectively. Of course, the question of which decisions are likely to require changing for some family members is again a question which requires judgment and experience. It is, however, a somewhat more concrete and more easily discussed question than ease of comprehension.

Historical Note

In closing this comparison, I want to make a comment on the origin and history of some of the ideas found in this paper. I recently reread one of the papers in which Dijkstra introduced the ideas of structured programming [3]. This paper is unusual in that it seems better each time you read it. The root of *both* methods of producing program families and the concept of family itself is in this original work by Dijkstra. The concept of the division into modules is somewhat differently formulated, but it is present in the concept of the design of the abstract machines, the notion of information hiding is implicit (in the discussion of the thickness of the ropes tying the pearls together). Module specification is not discussed. (Naur introduced a concept quite similar to that of the module when he discussed action clusters [10], but the concept of information hiding was not made specific and the example does not correspond exactly to what this principle would suggest.) For various reasons the concept of division into modules and the hiding of information seems to have attracted less attention, and later works by other authors [4], [5] have emphasized only the stepwise refinement of programs, ignoring the order of the steps or the question of the thickness of the ropes.

Acknowledgment

I am grateful for opportunities to discuss the subject with members of I.F.I.P. Working Group 2.3 on Programming Methodology. These discussions have helped me to clarify the points in this paper. I am also grateful to W. Bartussek of the Technische Hochschule Darmstadt, for his thoughtful comments on an earlier version of this paper, to Dr. H. Mills of the IBM Federal Systems Division who found a rather subtle error in a recent draft, and to Dr. L. Belady of the IBM T. J. Watson Research Laboratory who made a number of helpful comments.

References

[1] D. L. Parnas, "Some conclusions from an experiment in software engineering techniques," in *1972 Fall Joint Computer Conf., AFIPS Conf. Proc.*, vol. 41. Montvale, NJ: AFIPS Press, 1972, pp. 325-329.

[2] H. Mills, "Mathematical foundations of structured programming," IBM Federal Systems Div., No. FSC72-6012, pp. 1-62, Feb. 1972.

[3] E. W. Dijkstra, "Structured programming," in *Software Engineer-*

ing Techniques, J. N. Buxton and B. Randell, Ed. Brussels, Belgium: NATO Scientific Affairs Division, 1970, pp. 84–87.
[4] N. Wirth, "Program development by stepwise refinement," *Commun. ACM*, vol. 14, pp. 221–227, Apr. 1971.
[5] W. A. Wulf, "The GOTO controversy: A case against the GOTO," *SIGPLAN Notices*, vol. 7, pp. 63–69, Nov. 1972.
[6] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. ACM*, vol. 17, pp. 549–557, Oct. 1974.
[7] E. W. Dijkstra, "On the axiomatic definition of semantics," *EWD 367*, privately circulated.
[8] D. L. Parnas, "On the criteria used in decomposing systems into modules," *Commun. ACM*, vol. 15, pp. 1053–1058, Dec. 1972.
[9] —, "A technique for software module specification with examples," *Commun. ACM* (Programming Techniques Dept.), pp. 330–336, May 1972.
[10] P. Naur, "Programming by action clusters," *BIT*, vol. 9, pp. 250–258, 1969.
[11] P. Henderson and R. Snowdon, "An experiment in structured programming," *BIT*, vol. 12, pp. 38–53, 1972.
[12] D. L. Parnas, "A course on software engineering techniques," in *Proc. ACM SIGCSE*, 2nd Tech. Symp., Mar. 24–25, 1972.
[13] E. W. Dijkstra, "Co-operating sequential processes," *Programming Languages*, F. Genuys, Ed. New York: Academic Press, 1968, pp. 43–112.
[14] W. R. Price, "Implications of a virtual memory mechanism for implementing protection in a family of operating systems," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, 1973.
[15] B. Randell and F. W. Zurcher, "Iterative multi-level modelling—A methodology for computer system design," in *Proc. IFIP Congr.*, 1968.
[16] D. L. Parnas, "On a 'buzzword' hierarchical structure," in *Proc. IFIP Congr.*, 1974, pp. 336–339.

David L. Parnas received the B.S. and M.S. degrees in electrical engineering, and the Ph.D. degree in systems and communications sciences, from the Carnegie Institute of Technology, Pittsburgh, PA, in 1961, 1964, and 1965, respectively.

He has held the position of Assistant Professor of Computer Science, University of Maryland, College Park, and was Assistant and Associate Professor of Computer Science at Carnegie-Mellon University, Pittsburgh, PA. Since June of 1973 he has been Professor and Head of one of the two Research Groups on Operating Systems at the Technische Hochschule Darmstadt, Darmstadt, West Germany. He is also a consultant for the U.S. Naval Research Laboratory, Washington, D.C. His areas of research have been design methods for computer systems, process synchronization in operating systems, security mechanisms in operating systems, simulation techniques, and design automation.

Higher Order Software—A Methodology for Defining Software

MARGARET HAMILTON AND SAYDEAN ZELDIN

Abstract—**The key to software reliability is to design, develop, and manage software with a formalized methodology which can be used by computer scientists and applications engineers to describe and communicate interfaces between systems. These interfaces include: software to software; software to other systems; software to management; as well as discipline to discipline within the complete software development process. The formal methodology of Higher Order Software (HOS), specifically aimed toward large-scale multiprogrammed/multiprocessor systems, is dedicated to systems reliability. With six axioms as the basis, a given system and all of its interfaces is defined as if it were one complete and consistent computable system. Some of the derived theorems provide for: reconfiguration of real-time multiprogrammed processes, communication between functions, and prevention of data and timing conflicts.**

The first step in defining a system with a formal methodology is to apply a formalized set of rules. We have found that enforcing such rules, especially on a large project with many organizations, is very difficult. In fact, it is almost impossible without the aid of automated tools to describe the design process and its verification. We envision a scheme in which the definition of a given system can be described with an HOS specification language which, by its very nature, enforces the axioms with the use of each construct. A system defined in HOS can be analyzed automatically for axiomatic consistency by the Design Analyzer without program execution, and by the Structuring Executive Analyzer on a real-time basis. The result is that a software system can be developed efficiently with reliable interfaces. This is significant since interface testing in a large system accounts for approximately 75 percent of the verification effort.[1]

Manuscript received May 1, 1975; revised October 15, 1975. This paper was prepared under Contract NASA9-13809 with the Lyndon B. Johnson Space Center of the National Aeronautics and Space Administration, and under The Charles Stark Draper Laboratory, Inc., Internal Research and Development Funds. The publication of this paper does not constitute approval by the National Aeronautics and Space Administration of the findings or conclusions contained herein. It is published only for the exchange and stimulation of ideas. The Charles Stark Draper Laboratory, Inc., 68 Albany St., Cambridge, MA 02142 already holds the copyright to the contents contained in this manuscript.

The authors are with the Computer Science Division, The Charles Stark Draper Laboratory, Inc., Cambridge, MA 02142.

[1] Seventy-three percent of all problems found during the APOLLO integration effort were interface problems [2]; and verification accounts for 50 percent of the total software development effort [3]–[5].