Considerations and techniques are proposed that reduce the complexity of programs by dividing them into functional modules. This can make it possible to create complex systems from simple, independent, reusable modules. Debugging and modifying programs, reconfiguring I/O devices, and managing large programming projects can all be greatly simplified. And, as the module library grows, increasingly sophisticated programs can be implemented using less and less new code.

Structured design

by W. P. Stevens, G. J. Myers, and L. L. Constantine

Structured design is a set of proposed general program design considerations and techniques for making coding, debugging, and modification easier, faster, and less expensive by reducing complexity.[1] The major ideas are the result of nearly ten years of research by Mr. Constantine.[2] His results are presented here, but the authors do not intend to present the theory and derivation of the results in this paper. These ideas have been called *composite design* by Mr. Myers.[3-5] The authors believe these program *design* techniques are compatible with, and enhance, the *documentation* techniques of HIPO[6] and the *coding* techniques of structured programming.[7]

These cost-saving techniques always need to be balanced with other constraints on the system. But the ability to produce simple, changeable programs will become increasingly important as the cost of the programmer's time continues to rise.

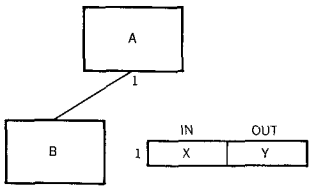
General considerations of structured design

Simplicity is the primary measurement recommended for evaluating alternative designs relative to reduced debugging and modification time. Simplicity can be enhanced by dividing the system into separate pieces in such a way that pieces can be considered, implemented, fixed, and changed with minimal consideration or effect on the other pieces of the system. Observability (the ability to easily perceive how and why actions occur) is another use-

ful consideration that can help in designing programs that can be changed easily. Consideration of the effect of reasonable changes is also valuable for evaluating alternative designs.

Mr. Constantine has observed that programs that were the easiest to implement and change were those composed of simple, independent modules. The reason for this is that problem solving is faster and easier when the problem can be subdivided into pieces which can be considered separately. Problem solving is hardest when all aspects of the problem must be considered simultaneously.

Figure 1 A structure chart



The term *module* is used to refer to a set of one or more contiguous program statements having a name by which other parts of the system can invoke it and preferably having its own distinct set of variable names. Examples of modules are PL/I procedures, FORTRAN mainlines and subprograms, and, in general, subroutines of all types. Considerations are always with relation to the program statements *as coded*, since it is the programmer's ability to understand and change the *source* program that is under consideration.

While conceptually it is useful to discuss dividing whole programs into smaller pieces, the techniques presented here are for designing simple, independent modules originally. It turns out to be difficult to divide an existing program into separate pieces without increasing the complexity because of the amount of overlapped code and other interrelationships that usually exist.

Graphical notation is a useful tool for structured design. Figure 1 illustrates a notation called a *structure chart*,[8] in which:

1. There are two modules, A and B.
2. Module A *invokes* module B. B is *subordinate* to A.
3. B receives an input parameter X (its name in module A) and returns a parameter Y (its name in module A). (It is useful to distinguish which calling parameters represent data passed *to* the called program and which are for data to be *returned* to the caller.)

Coupling and communication

To evaluate alternatives for dividing programs into modules, it becomes useful to examine and evaluate types of "connections" between modules. A connection is a reference to some label or address defined (or also defined) elsewhere.

The fewer and simpler the connections between modules, the easier it is to understand each module without reference to other

Table 1 Contributing factors

	Interface complexity	*Type of connection*	*Type of communication*
low	simple, obvious	to module by name	data
COUPLING			control
high	complicated, obscure	to internal elements	hybrid

modules. Minimizing connections between modules also minimizes the paths along which changes and errors can propagate into other parts of the system, thus eliminating disastrous "ripple" effects, where changes in one part cause errors in another, necessitating additional changes elsewhere, giving rise to new errors, etc. The widely used technique of using common data areas (or global variables or modules without their own distinct set of variable names) can result in an enormous number of connections between the modules of a program. The complexity of a system is affected not only by the number of connections but by the degree to which each connection couples (associates) two modules, making them interdependent rather than independent. Coupling is the measure of the strength of association established by a connection from one module to another. Strong coupling complicates a system since a module is harder to understand, change, or correct by itself if it is highly interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules.

The degree of coupling established by a particular connection is a function of several factors, and thus it is difficult to establish a simple index of coupling. Coupling depends (1) on how complicated the connection is, (2) on whether the connection refers to the module itself or something inside it, and (3) on what is being sent or received.

Coupling increases with increasing complexity or obscurity of the interface. Coupling is lower when the connection is to the normal module interface than when the connection is to an internal component. Coupling is lower with data connections than with control connections, which are in turn lower than hybrid connections (modification of one module's code by another module). The contribution of all these factors is summarized in Table 1.

interface complexity

When two or more modules interface with the same area of storage, data region, or device, they share a common environment. Examples of common environments are:

- A set of data elements with the EXTERNAL attribute that is

copied into PL/I modules via an INCLUDE statement or that is found listed in each of a number of modules.
- Data elements defined in COMMON statements in FORTRAN modules.
- A centrally located "control block" or set of control blocks.
- A common overlay region of memory.
- Global variable names defined over an entire program or section.

The most important structural characteristic of a common environment is that it couples every module sharing it to every other such module without regard to their functional relationship or its absence. For example, only the two modules XVECTOR and VELOC might actually make use of data element X in an "included" common environment of PL/1, yet changing the length of X impacts *every* module making any use of the common environment, and thus necessitates recompilation.

Every element in the common environment, whether used by particular modules or not, constitutes a separate path along which errors and changes can propagate. Each element in the common environment adds to the complexity of the total system to be comprehended by an amount representing all possible pairs of modules sharing that environment. Changes to, and new uses of, the common area potentially impact all modules in unpredictable ways. Data references may become unplanned, uncontrolled, and even unknown.

A module interfacing with a common environment for some of its input or output data is, on the average, more difficult to use in varying contexts or from a variety of places or in different programs than is a module with communication restricted to parameters in calling sequences. It is somewhat clumsier to establish a new and unique data context on each call of a module when data passage is via a common environment. Without analysis of the entire set of sharing modules or careful saving and restoration of values, a new use is likely to interfere with other uses of the common environment and propagate errors into other modules. As to future growth of a given system, once the commitment is made to communication via a common environment, any new module will have to be plugged into the common environment, compounding the total complexity even more. On this point, Belady and Lehman,[9] observe that "a well-structured system, one in which communication is via passed parameters through defined interfaces, is likely to be more growable and require less effort to maintain than one making extensive use of global or shared variables."

The impact of common environments on system complexity may be quantified. Among M objects there are $M\ (M-1)$ or-

dered pairs of objects. (Ordered pairs are of interest because A and B sharing a common environment complicates both, A being coupled to B and B being coupled to A.) Thus a common environment of N elements shared by M modules results in $NM(M-1)$ first order (one level) relationships or paths along which changes and errors can propagate. This means 150 such paths in a FORTRAN program of only three modules sharing the COMMON area with just 25 variables in it.

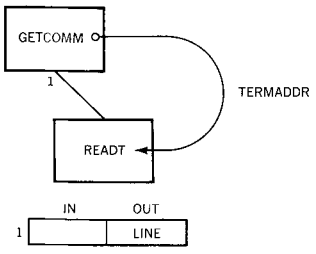
It is possible to minimize these disadvantages of common environments by limiting access to the smallest possible subset of modules. If the total set of potentially shared elements is subdivided into groups, all of which are *required* by some subset of modules, then both the size of each common environment and the scope of modules among which it is shared is reduced. Using "named" rather than "blank" COMMON in FORTRAN is one means of accomplishing this end.

The complexity of an interface is a matter of how much information is needed to state or to understand the connection. Thus, obvious relationships result in lower coupling than obscure or inferred ones. The more syntactic units (such as parameters) in the statement of a connection, the higher the coupling. Thus, extraneous elements irrelevant to the programmer's and the modules' immediate task increase coupling unnecessarily.

type of connection

Connections that address or refer to a module as a whole by its name (leaving its contents unknown and irrelevant) yield lower coupling than connections referring to the internal elements of another module. In the latter case, as for example the use of a variable by direct reference from within some other module, the entire content of that module may have to be taken into account to correct an error or make a change so that it does not make an impact in some unexpected way. Modules that can be used easily without knowing anything about their insides make for simpler systems.

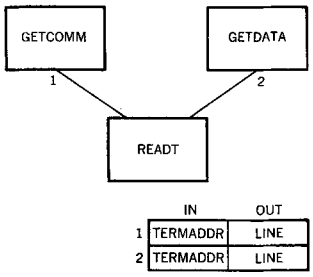
Figure 2 Module connections



Consider the case depicted in Figure 2. GETCOMM is a module whose function is getting the next command from a terminal. In performing this function, GETCOMM calls the module READT, whose function is to read a line from the terminal. READT requires the address of the terminal. It gets this via an externally declared data element in GETCOMM, called TERMADDR. READT passes the line back to GETCOMM as an argument called LINE. Note the arrow extending from *inside* GETCOMM to *inside* READT. An arrow of this type is the notation for references to internal data elements of another module.

Now, suppose we wish to add a module called GETDATA, whose function is to get the next data line (i.e., not a command) from a

Figure 3 Improved module connections

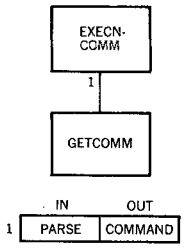


(possibly) different terminal. It would be desirable to use module READT as a subroutine of GETDATA. But if GETDATA modifies TERMADDR in GETCOMM before calling READT, it will cause GETCOMM to fail since it will "get" from the wrong terminal. Even if GETDATA restores TERMADDR after use, the error can still occur if GETDATA and GETCOMM can ever be invoked "simultaneously" in a multiprogramming environment. READT would have been more usable if TERMADDR had been made an input argument to READT instead of an externally declared data item as shown in Figure 3. This simple example shows how references to internal elements of other modules can have an adverse effect on program modification, both in terms of cost and potential bugs.

type of communication

Modules must at least pass data or they cannot functionally be a part of a single system. Thus connections that pass data are a necessary minimum. (Not so the communication of control. In principle, the presence or absence of requisite input data is sufficient to define the circumstances under which a module should be activated, that is, receive control. Thus the explicit passing of control by one module to another constitutes an additional, theoretically inessential form of coupling. In practice, systems that are *purely* data-coupled require special language and operating system support but have numerous attractions, not the least of which is they can be fundamentally simpler than any equivalent system with control coupling.[10])

Figure 4 Control-coupled modules



Beyond the practical, innocuous, minimum control coupling of normal subroutine calls is the practice of passing an "element of control" such as a switch, flag, or signal from one module to another. Such a connection affects the execution of another module and not merely the data it performs its task upon by involving one module in the internal processing of some other module. Control arguments are an additional complication to the essential data arguments required for performance of some task, and an alternative structure that eliminates the complication always exists.

Consider the modules in Figure 4 that are control-coupled by the switch PARSE through which EXECNCOMM instructs GETCOMM whether to return a parsed or unparsed command. Separating the two distinct functions of GETCOMM results in a structure that is simpler as shown in Figure 5.

The new EXECNCOMM is no more complicated; where once it set a switch and called, now it has two alternate calls. The sum of GETPCOMM and GETUCOMM is (functionally) less complicated than GETCOMM was (by the amount of the switch testing). And the two small modules are likely to be easier to comprehend than the one large one. Admittedly, the immediate gains here

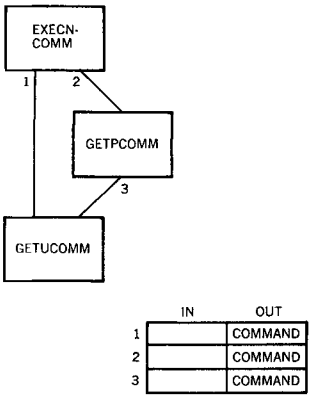
may appear marginal, but they rise with time and the number of alternatives in the switch and the number of levels over which it is passed. Control coupling, where a called module "tells" its caller what to do, is a more severe form of coupling.

Modification of one module's code by another module may be thought of as a hybrid of data and control elements since the code is dealt with as data by the modifying module, while it acts as control to the modified module. The target module is very dependent in its behavior on the modifying module, and the latter is intimately involved in the other's internal functioning.

Cohesiveness

Coupling is reduced when the relationships among elements *not* in the same module are minimized. There are two ways of achieving this – minimizing the relationships among modules and maximizing relationships among elements in the same module. In practice, both ways are used.

Figure 5 Simplified coupling



The second method is the subject of this section. "Element" in this sense means any form of a "piece" of the module, such as a statement, a segment, or a "subfunction". Binding is the measure of the cohesiveness of a module. The objective here is to reduce coupling by striving for high binding. The scale of cohesiveness, from lowest to highest, follows:

1. Coincidental.
2. Logical.
3. Temporal.
4. Communicational.
5. Sequential.
6. Functional.

The scale is not linear. Functional binding is much stronger than all the rest, and the first two are much weaker than all the rest. Also, higher-level binding classifications often include all the characteristics of one or more classifications below it *plus* additional relationships. The binding between two elements is the highest classification that applies. We will define each type of binding, give an example, and try to indicate why it is found at its particular position on the scale.

coincidental binding

When there is no meaningful relationship among the elements in a module, we have coincidental binding. Coincidental binding might result from either of the following situations: (1) An existing program is "modularized" by splitting it apart into modules. (2) Modules are created to consolidate "duplicate coding" in other modules.

As an example of the difficulty that can result from coincidental binding, suppose the following sequence of instructions appeared several times in a module or in several modules and was put into a separate module called X:

```
A = B + C
GET CARD
PUT OUTPUT
IF B = 4, THEN E = 0
```

Module X would probably be coincidentally bound since these four instructions have no apparent relationships among one another. Suppose in the future we have a need in one of the modules originally containing these instructions to say GET TAPERECORD instead of GET CARD. We now have a problem. If we modify the instruction in module X, it is unusable to all of the other callers of X. It may even be difficult to *find* all of the other callers of X in order to make any other compatible change.

It is only fair to admit that, independent of a module's cohesiveness, there are instances when any module can be modified in such a fashion to make it unusable to all its callers. However, the *probability* of this happening is very high if the module is coincidentally bound.

logical binding

Logical binding, next on the scale, implies some logical relationship between the elements of a module. Examples are a module that performs all input and output operations for the program or a module that edits all data.

The logically bound, EDIT ALL DATA module is often implemented as follows. Assume the data elements to be edited are master file records, updates, deletions, and additions. Parameters passed to the module would include the data and a special parameter indicating the type of data. The first instruction in the module is probably a four-way branch, going to four sections of code – edit master record, edit update record, edit addition record, and edit deletion record.

Often, these four functions are also intertwined in some way in the module. If the deletion record changes and requires a change to the edit deletion record function, we will have a problem if this function is intertwined with the other three. If the edits are truly independent, then the system could be simplified by putting each edit in a separate module and eliminating the need to decide which edit to do for each execution. In short, logical binding usually results in tricky or shared code, which is difficult to modify, and in the passing of unnecessary parameters.

temporal binding

Temporal binding is the same as logical binding, except the elements are also related in time. That is, the temporally bound elements are executed in the same time period.

The best examples of modules in this class are the traditional "initialization", "termination", "housekeeping", and "clean-up" modules. Elements in an initialization module are logically bound because initialization represents a logical class of functions. In addition, these elements are related in time (i.e., at initialization time).

Modules with temporal binding tend to exhibit the disadvantages of logically bound modules. However, temporally bound modules are higher on the scale since they tend to be simpler for the reason that *all* of the elements are executable at one time (i.e., no parameters and logic to determine which element to execute).

communicational binding

A module with communicational binding has elements that are related by a reference to the same set of input and/or output data. For example, "print and punch the output file" is communicationally bound. Communicational binding is higher on the scale than temporal binding since the elements in a module with communicational binding have the stronger "bond" of referring to the same data.

sequential binding

When the output data from an element is the input for the next element, the module is sequentially bound. Sequential binding can result from flowcharting the problem to be solved and then defining modules to represent one or more blocks in the flowchart. For example, "read next transaction and update master file" is sequentially bound.

Sequential binding, although high on the scale because of a close relationship to the problem structure, is still far from the maximum—functional binding. The reason is that the procedural processes in a program are usually distinct from the *functions* in a program. Hence, a sequentially bound module can contain several functions or just part of a function. This usually results in higher coupling and modules that are less likely to be usable from other parts of the system.

functional binding

Functional binding is the strongest type of binding. In a functionally bound module, all of the elements are related to the performance of a single function.

A question that often arises at this point is what is a function? In mathematics, $Y = F(X)$ is read "Y is a function F of X." The function F defines a transformation or mapping of the independent (or input) variable X into the dependent (or return) variable Y. Hence, a function describes a transformation from some

input data to some return data. In terms of programming, we broaden this definition to allow functions with no input data and functions with no return data.

In practice, the above definition does not clearly describe a functionally bound module. One hint is that if the elements of the module all contribute to accomplishing a single goal, then it is probably functionally bound. Examples of functionally bound modules are "Compute Square Root" (input and return parameters) "Obtain Random Number" (no input parameter), and "Write Record to Output File" (no return parameter).

A useful technique in determining whether a module is functionally bound is writing a sentence describing the function (purpose) of the module, and then examining the sentence. The following tests can be made:

1. If the sentence *has* to be a compound sentence, contain a comma, or contain more than one verb, the module is probably performing more than one function; therefore, it probably has sequential or communicational binding.

2. If the sentence contains words relating to time, such as "first", "next", "then", "after", "when", "start", etc., then the module probably has sequential or temporal binding.

3. If the predicate of the sentence doesn't contain a single specific object following the verb, the module is probably logically bound. For example, Edit All Data has logical binding; Edit Source Statement may have functional binding.

4. Words such as "initialize", "clean-up", etc. imply temporal binding.

Functionally bound modules *can* always be described by way of their elements using a compound sentence. But if the above language is unavoidable while still completely describing the module's function, then the module is probably not functionally bound.

One unresolved problem is deciding how far to divide functionally bound subfunctions. The division has probably gone far enough if each module contains no subset of elements that could be useful alone, and if each module is small enough that its entire implementation can be grasped all at once, i.e., seldom longer than one or two pages of source code.

Observe that a module can include more than one type of binding. The binding between two elements is the highest that can be

applied. The binding of a module is lowered by every element pair that does not exhibit functional binding.

Predictable modules

A predictable, or well-behaved, module is one that, when given the identical inputs, operates identically each time it is called. Also, a well-behaved module operates independently of its environment.

To show that dependable (free from errors) modules can still be unpredictable, consider an oscillator module that returns zero and one alternately and dependably when it is called. It might be used to facilitate double buffering. Should it have multiple users, each would be required to call it an even number of times before relinquishing control. Should any of the users have an error that prevented an even number of calls, all other users will fail. The operation of the module given the same inputs is not constant, resulting in the module not being predictable even though error-free. Modules that keep track of their own state are usually not predictable, even when error-free.

This characteristic of predictability that can be designed into modules is what we might loosely call "black-boxness." That is, the user can understand what the module does and use it without knowing what is inside it. Module "black-boxness" can even be enhanced by merely adding comments that make the module's function and use clear. Also, a descriptive name and a well-defined and visible interface enhances a module's usability and thus makes it more of a black box.

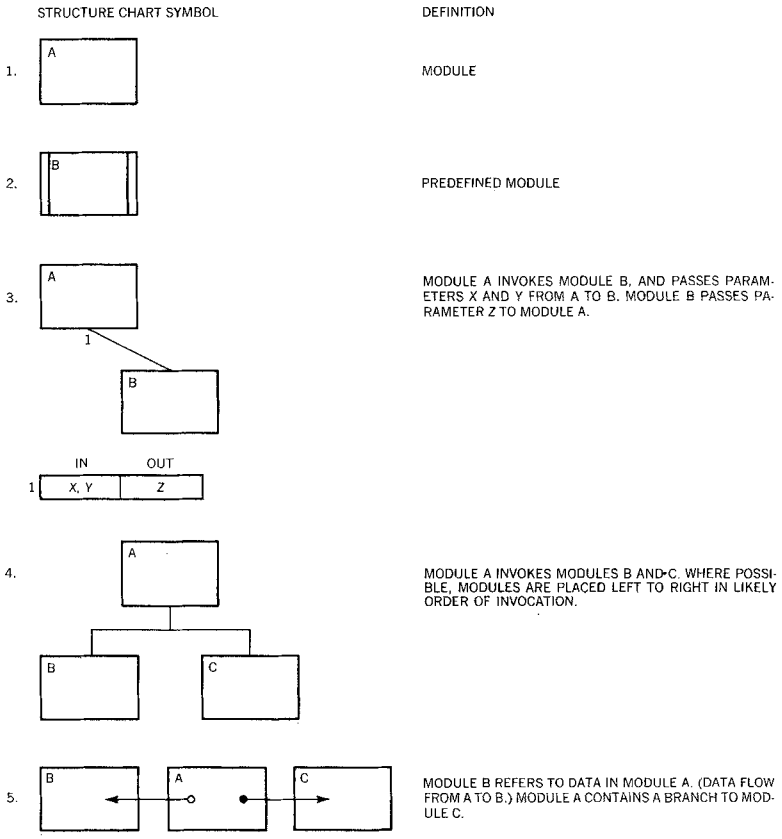
Tradeoffs to structured design

The overhead involved in writing many simple modules is in the execution time and memory space used by a particular language to effect the call. The designer should realize the adverse effect on maintenance and debugging that may result from striving just for minimum execution time and/or memory. He should also remember that programmer cost, is, or is rapidly becoming, the major cost of a programming system and that much of the maintenance will be in the future when the trend will be even more prominent. However, depending on the actual overhead of the language being used, it is very possible that a structured design can result in less execution and/or memory overhead rather than more due to the following considerations:

For memory overhead

1. Optional (error) modules may never be called into memory.

Figure 6 Definitions of symbols used in structure charts



THE MORE COMPREHENSIVE "PROPOSED STANDARD GRAPHICS FOR PROGRAM STRUCTURE," PREFERRED BY MR. CONSTANTINE AND WIDELY USED OVER THE PAST SIX YEARS BY HIS CLASSES AND CLIENTS, USES SEPARATE ARROWS FOR EACH CONNECTION, SUCH AS FOR THE CALLS FROM A TO B AND FROM A TO C, TO REFLECT STRUCTURAL PROPERTIES OF THE PROGRAM. THE CHARTING SHOWN HERE WAS ADOPTED FOR COMPATIBILITY WITH THE HIERARCHY CHART OF HIPO.

2. Structured design reduces duplicate code and the coding necessary for implementing control switches, thus reducing the amount of programmer-generated code.
3. Overlay structuring can be based on actual operating characteristics obtained by running and observing the program.
4. Having many single-function modules allows more flexible, and precise, grouping, possibly resulting in less memory needed at any one time under overlay or virtual storage constraints.

For execution overhead

1. Some modules may only execute a few times.
2. Optional (error) functions may never be called, resulting in zero overhead.
3. Code for control switches is reduced or eliminated, reducing the total amount of code to be executed.

4. Heavily used linkage can be recompiled and calls replaced by branches.
5. "Includes" or "performs" can be used in place of calls. (However, the complexity of the system will increase by at least the extra consideration necessary to prevent duplicating data names and by the difficulty of creating the equivalent of call parameters for a well-defined interface.)
6. One way to get fast execution is to determine which parts of the system will be most used so all optimizing time can be spent on those parts. Implementing an initially structured design allows the testing of a working program for those critical modules (and yields a working program prior to any time spent optimizing). Those modules can then be optimized separately and reintegrated without introducing multitudes of errors into the rest of the program.

Structured design techniques

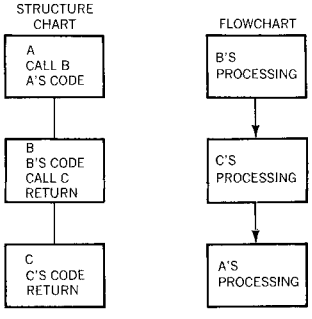
It is possible to divide the design process into general program design and detailed design as follows. General program design is deciding *what* functions are needed for the program (or programming system). Detailed design is *how* to implement the functions. The considerations above and techniques that follow result in an identification of the functions, calling parameters, and the call relationships for a structure of functionally bound, simply connected modules. The information thus generated makes it easier for each module to then be separately designed, implemented, and tested.

structure charts

The objective of general program design is to determine what functions, calling parameters, and call relationships are needed. Since flowcharts depict *when* (in what order and under what conditions) blocks are executed, flowcharts unnecessarily complicate the general program design phase. A more useful notation is the structure chart, as described earlier and as shown in Figure 6.

To contrast a structure chart and a flowchart, consider the following for the same three modules in Figure 7 – A which calls B which calls C (coding has been added to the structure chart to enable the proper flowchart to be determined; B's code will be executed first, then C's, then A's). To design A's interfaces properly, it is necessary to know that A is responsible for invoking B, but this is hard to determine from the flowchart. In addition, the structure chart can show the module connections and calling parameters that are central to the consideration and techniques being presented here.

Figure 7 Structure chart compared to flowchart



The other major difference that drastically simplifies the nota-

Figure 8 Basic form of low-cost implementation

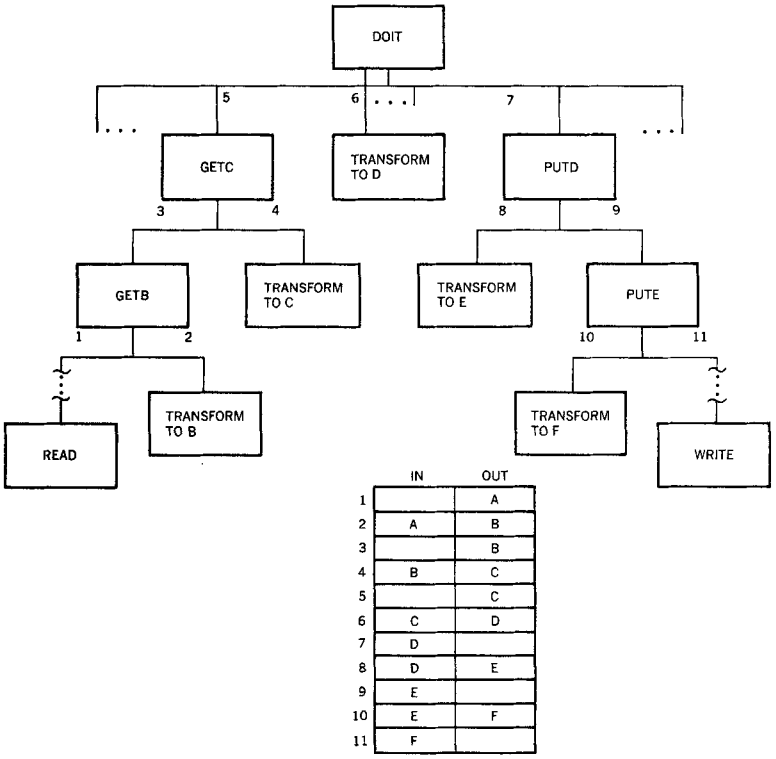
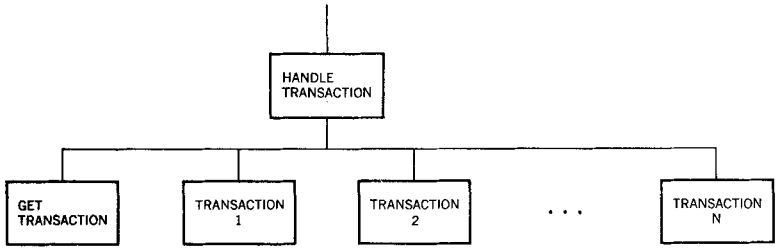


Figure 9 Transaction structure

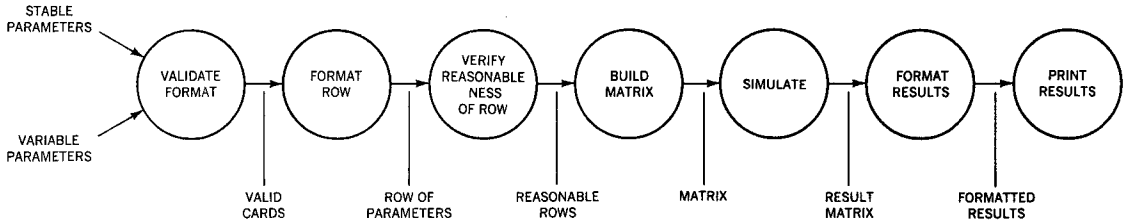


tion and analysis during general program design is the absence in structure charts of the decision block. Conditional calls can be so noted, but "decision designing" can be deferred until detailed module design. This is an example of where the *design* process is made simpler by having to consider only part of the design problem. Structure charts are also small enough to be worked on all at once by the designers, helping to prevent suboptimizing parts of the program at the expense of the entire problem.

common structures

A shortcut for arriving at simple structures is to know the general form of the result. Mr. Constantine observed that programs of the general structure in Figure 8 resulted in the lowest-cost

Figure 10 Rough structure of simulation system



implementations. It implements the input-process-output type of program, which applies to most programs, even if the "input" or "output" is to secondary storage or to memory.

In practice, the sink leg is often shorter than the source one. Also, source modules may produce output (e.g., error messages) and sink modules may request input (e.g., execution-time format commands.)

Another structure useful for implementing parts of a design is the transaction structure depicted in Figure 9. A "transaction" here is any event, record, or input, etc. for which various actions should result. For example, a command processor has this structure. The structure may occur alone or as one or more of the source (or even sink) modules of an input-process-output structure. Analysis of the transaction modules follows that of a transform module, which is explained later.

designing the structure

The following procedure can be used to arrive at the input-process-output general structure shown previously.

Step One. The first step is to sketch (or mentally consider) a functional picture of the problem. As an example, consider a simulation system. The rough structure of this problem is shown in Figure 10.

Step Two. Identify the external conceptual streams of data. An *external* stream of data is one that is external to the system. A *conceptual* stream of data is a stream of related data that is independent of any physical I/O device. For instance, we may have several conceptual streams coming from one I/O device or one stream coming from several I/O devices. In our simulation system, the external conceptual streams are the input parameters, and the formatted simulation the result.

Step Three. Identify the *major* external conceptual stream of data (both input and output) in the problem. Then, using the diagram of the problem structure, determine, for this stream, the points of "highest abstraction" as in Figure 11.

Figure 11 Determining points of highest abstraction

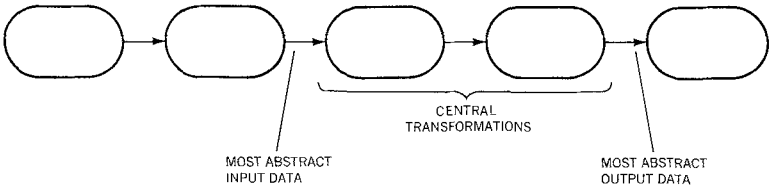
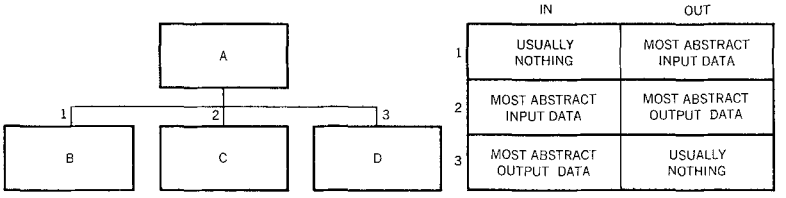


Figure 12 The top level



The "point of highest abstraction" for an input stream of data is the point in the problem structure where that data is farthest removed from its physical input form yet can still be viewed as coming in. Hence, in the simulation system, the most abstract form of the input transaction stream might be the built matrix. Similarly, identify the point where the data stream can first be viewed as going out—in the example, possibly the result matrix.

Admittedly, this is a subjective step. However, experience has shown that designers trained in the technique seldom differ by more than one or two blocks in their answers to the above.

Step Four. Design the structure in Figure 12 from the previous information with a source module for each conceptual input stream which exists at the point of most abstract input data; do sink modules similarly. Often only single source and sink branches are necessary. The parameters passed are dependent on the problem, but the general pattern is shown in Figure 12.

Describe the function of each module with a short, concise, and specific phrase. Describe what transformations occur when that module is called, not how the module is implemented. Evaluate the phrase relative to functional binding.

When module A is called, the program or system executes. Hence, the function of module A is equivalent to the problem being solved. If the problem is "write a FORTRAN compiler," then the function of module A is "compile FORTRAN program."

Module B's function involves obtaining the major stream of data. An example of a "typical module B" is "get next valid source statement in Polish form."

Module C's purpose is to transform the major input stream into the major output stream. Its function should be a nonprocedural description of this transformation. Examples are "convert Polish form statement to machine language statement" or "using keyword list, search abstract file for matching abstracts."

Module D's purpose is disposing of the major output stream. Examples are "produce report" or "display results of simulation."

Step Five. For each source module, identify the last transformation necessary to produce the form being returned by that module. Then identify the form of the input just prior to the last transformation. For sink modules, identify the first process necessary to get closer to the desired output and the resulting output form. This results in the portions of the structure shown in Figure 13.

Repeat Step Five on the new source and sink modules until the original source and final sink modules are reached. The modules may be analyzed in any order, but each module should be done completely before doing any of its subordinates. There are, unfortunately, no detailed guidelines available for dividing the transform modules. Use binding and coupling considerations, size (about one page of source), and usefulness (are there subfunctions that could be useful elsewhere now or in the future) as guidelines on how far to divide.

During this phase, err on the side of dividing too finely. It is always easy to recombine later in the design, but duplicate func-

Figure 13 Lower levels

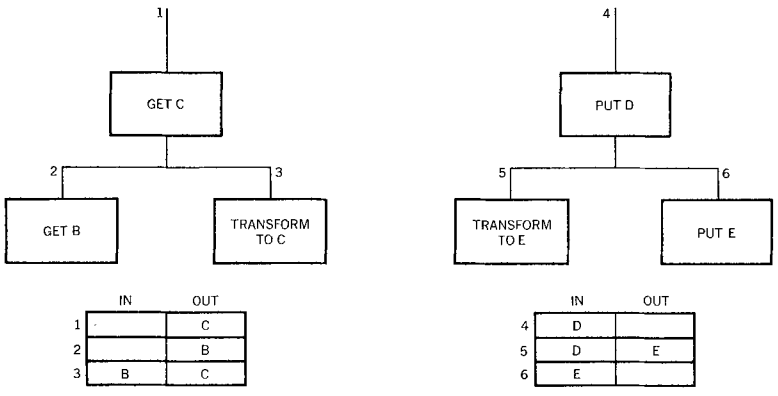
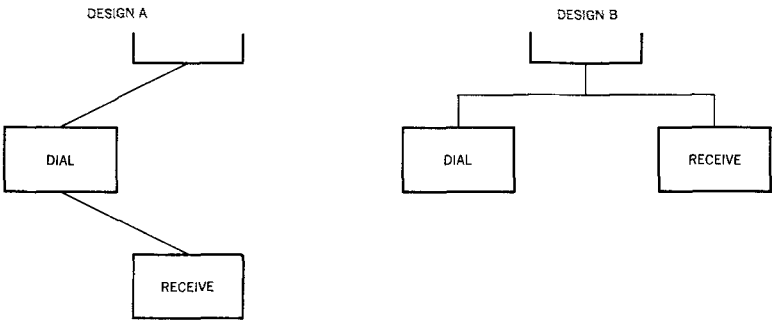


Figure 14 Design form should follow function



tions may not be identified if the dividing is too conservative at this point.

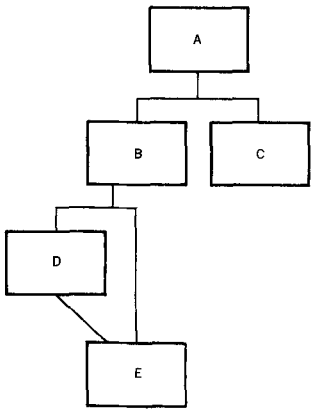
Design guidelines

The following concepts are useful for achieving simple designs and for improving the "first-pass" structures.

match program to problem

One of the most useful techniques for reducing the effect of changes on the program is to make the structure of the design match the structure of the problem, that is, form should follow function. For example, consider a module that dials a telephone and a module that receives data. If receiving immediately follows dialing, one might arrive at design A as shown in Figure 14. Consider, however, whether receiving is part of dialing. Since it is not (usually), have DIAL's caller invoke RECEIVE as in design B.

Figure 15 Scope of control



If, in this example, design A were used, consider the effect of a new requirement to transmit immediately after dialing. The DIAL module receives first and cannot be used, or a switch must be passed, or another DIAL module has to be added.

To the extent that the design structure does match the problem structure, changes to single parts of the problem result in changes to single modules.

scopes of effect and control

The *scope of control* of a module is that module plus all modules that are ultimately subordinate to that module. In the example of Figure 15, the scope of control of B is B, D, and E. The *scope of effect* of a decision is the set of all modules that contain some code whose execution is based upon the outcome of the decision. The system is simpler when the scope of effect of a decision is in the scope of control of the module containing the decision. The following example illustrates why.

If the execution of some code in A is dependent on the outcome of decision X in module B, then either B will have to return a flag to A or the decision will have to be repeated in A. The former approach results in added coding to implement the flag, and the latter results in some of B's function (decision X) in module A. Duplicates of decision X result in difficulties coordinating changes to both copies whenever decision X must be changed.

The scope of effect can be brought within the scope of control either by moving the decision element "up" in the structure, or by taking those modules that are in the scope of effect but not in the scope of control and moving them so that they fall within the scope of control.

module size

Size can be used as a signal to look for *potential* problems. Look carefully at modules with less than five or more than 100 executable source statements. Modules with a small number of statements may not perform an entire function, hence, may not have functional binding. Very small modules can be eliminated by placing their statements in the calling modules. Large modules may include more than one function. A second problem with large modules is understandability and readability. There is evidence to the fact that a group of about 30 statements is the upper limit of what can be mastered on the first reading of a module listing.[11]

error and end-of-file

Often, part of a module's function is to notify its caller when it cannot perform its function. This is accomplished with a return error parameter (preferably binary only). A module that handles streams of data must be able to signal end-of-file (EOF), preferably also with a binary parameter. These parameters should not, however, tell the caller what to do about the error or EOF. Nevertheless, the system can be made simpler if modules can be designed without the need for error flags.

intialization

Similarly, many modules require some initialization to be done. An initialize module will suffer from low binding but sometimes is the simplest solution. It may, however, be possible to eliminate the need for initializing without compromising "black-boxness" (the same inputs *always* produce the same outputs). For example, a read module that detects a return error of file-not-opened from the access method and recovers by opening the file and rereading eliminates the need for initialization without maintaining an internal state.

selecting modules

Eliminate duplicate functions but not duplicate code. When a function changes, it is a great advantage to only have to change it in one place. But if a module's need for its own copy of a random collection of code changes slightly, it will not be necessary to change several other modules as well.

Figure 16 Outline of problem structure

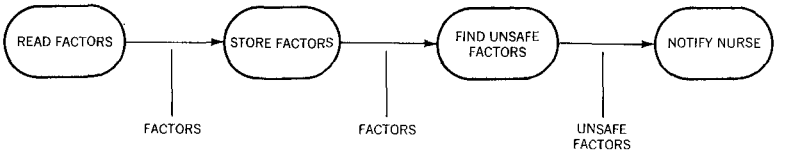
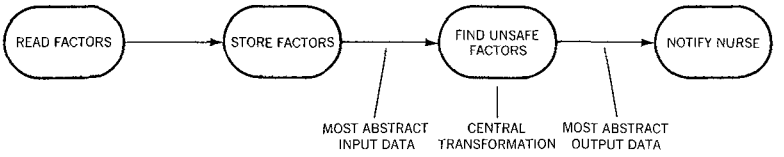


Figure 17 Points of highest abstraction



If a module seems almost, but not quite, useful from a second place in the system, try to identify and isolate the useful subfunction. The remainder of the module might be incorporated in its original caller.

Check modules that have many callers or that call many other modules. While not always a problem, it may indicate missing levels or modules.

isolate specifications

Isolate all dependencies on a particular data-type, record-layout, index-structure, etc. in one or a minimum of modules. This minimizes the recoding necessary should that particular specification change.

reduce parameters

Look for ways to reduce the number of parameters passed between modules. Count every item passed as a separate parameter for this objective (independent of how it will be implemented). Do not pass whole records from module to module, but pass only the field or fields necessary for each module to accomplish its function. Otherwise, all modules will have to change if one field expands, rather than only those which directly used that field. Passing only the data being processed by the program system with necessary error and EOF parameters is the ultimate objective. Check binary switches for indications of scope-of-effect/scope-of-control inversions.

Have the designers work together and with the complete structure chart. If branches of the chart are worked on separately, common modules may be missed and incompatibilities result from design decisions made while only considering one branch.

Figure 18 Structure of the top level

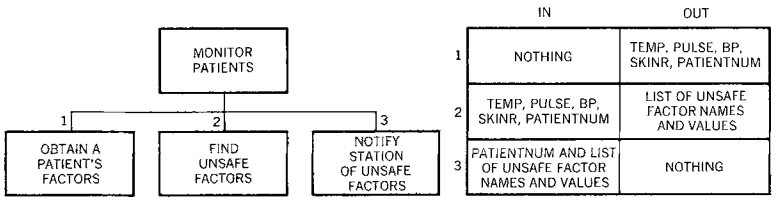
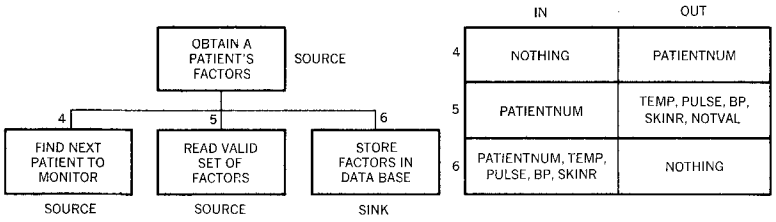


Figure 19 Structure of next level



An example

The following example illustrates the use of structured design:

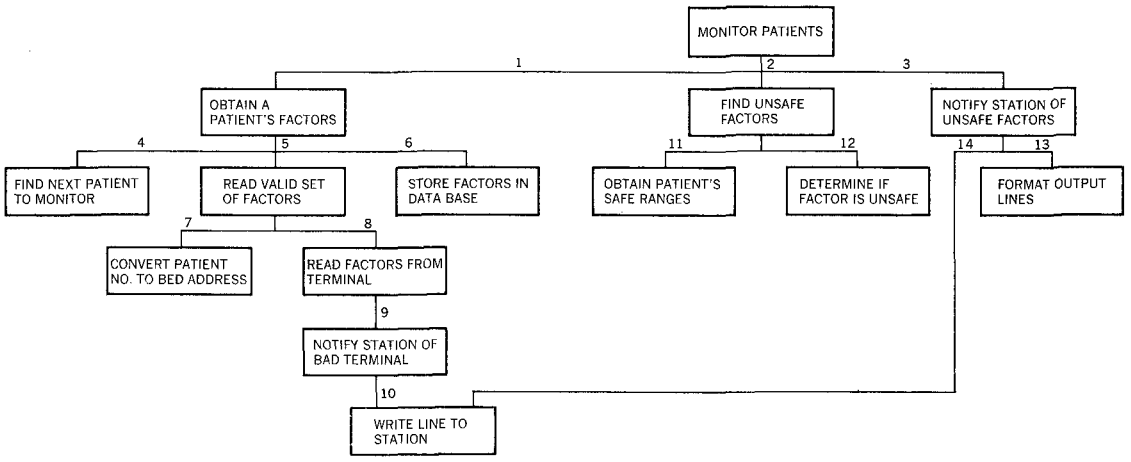
A patient-monitoring program is required for a hospital. Each patient is monitored by an analog device which measures factors such as pulse, temperature, blood pressure, and skin resistance. The program reads these factors on a periodic basis (specified for each patient) and stores these factors in a data base. For each patient, safe ranges for each factor are specified (e.g., patient X's valid temperature range is 98 to 99.5 degrees Fahrenheit). If a factor falls outside of a patient's safe range, or if an analog device fails, the nurse's station is notified.

In a real-life case, the problem statement would contain much more detail. However, this one is of sufficient detail to allow us to design the structure of the program.

The first step is to outline the structure of the problem as shown in Figure 16. In the second step, we identify the external conceptual streams of data. In this case, two streams are present, factors from the analog device and warnings to the nurse. These also represent the major input and output streams.

Figure 17 indicates the point of highest abstraction of the input stream, which is the point at which a patient's factors are in the form to store in the data base. The point of highest abstraction of the output stream is a list of unsafe factors (if any). We can now begin to design the program's structure as in Figure 18.

Figure 20 Complete structure chart



	IN	OUT
1	———	TEMP, PULSE, BP, SKINR, PATIENTNUM
2	TEMP, PULSE, BP, SKINR, PATIENTNUM	LIST OF UNSAFE FACTOR NAMES AND VALUES
3	PATIENTNUM & LIST OF UNSAFE FACTOR NAMES & VALUES	———
4	———	PATIENTNUM
5	PATIENTNUM	TEMP, PULSE, BP, SKINR, NOTVAL
6	PATIENTNUM, TEMP, PULSE, BP, SKINR	———
7	PATIENTNUM	BEDNUM
8	BEDNUM	TEMP, PULSE, BP, SKINR, NOTVAL
9	BEDNUM	———
10,14	LINE	———
11	PATIENTNUM	TEMPR, PULSER, BPR, SKINRR
12	FACTOR, RANGE	UNSAFE
13	LIST OF UNSAFE FACTOR NAMES AND VALUES	LIST OF LINES

In analyzing the module "OBTAIN A PATIENT'S FACTORS," we can deduce from the problem statement that this function has three parts: (1) Determine which patient to monitor next (based on their specified periodic intervals). (2) Read the analog device. (3) Record the factors in the data base. Hence, we arrive at the structure in Figure 19. (NOTVAL is set if a valid set of factors was not available.)

Further analysis of "READ VALID SET OF FACTORS", "FIND UNSAFE FACTORS" and "NOTIFY STATION OF UNSAFE FACTORS" yields the results shown in the complete structure chart in Figure 20.

Note that the module "READ FACTORS FROM TERMINAL" contains a decision asking "did we successfully read from the terminal?" If the read was not successful, we have to notify the nurse's station and then find the next patient to process as depicted in Figure 21.

Modules in the scope of effect of this decision are marked with an X. Note that the scope of effect is *not* a subset of the scope

Figure 21 Structure as designed

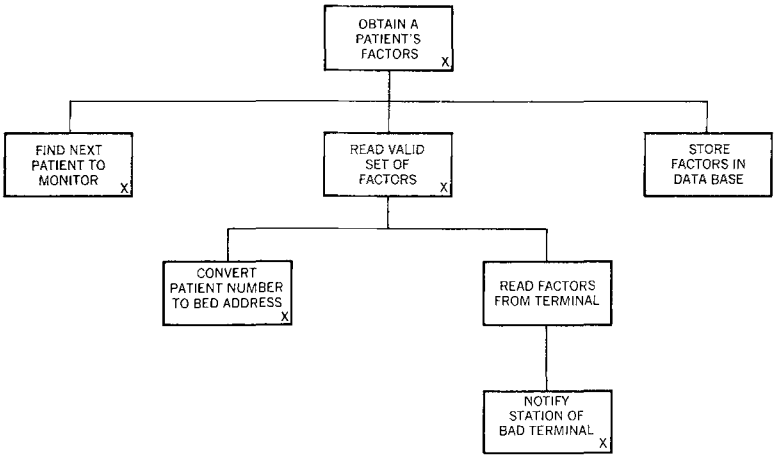
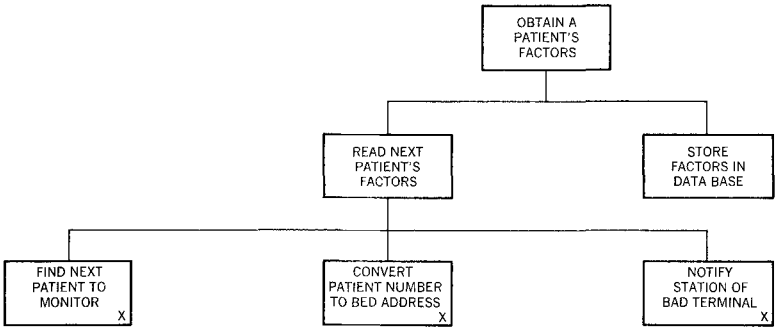


Figure 22 Scope of effect within scope of control



of control. To correct this problem, we have to take two steps. First, we will move the decision up to "READ VALID SET OF FACTORS." We do this by merging "READ FACTORS FROM TERMINAL" into its calling module. We now make "FIND NEXT PATIENT TO MONITOR" a subordinate of "READ VALID SET OF FACTORS." Hence, we have the structure in Figure 22. Thus, by slightly altering the structure and the function of a few modules, we have completely eliminated the problem.

Concluding remarks

The HIPO Hierarchy chart is being used as an aid during general systems design. The considerations and techniques presented here are useful for evaluating alternatives for those portions of the system that will be programmed on a computer. The charting technique used here depicts more details about the interfaces than the HIPO Hierarchy chart. This facilitates consideration during general program design of each individual connection and

its associated passed parameters. The resulting design can be documented with the HIPO charts. (If the designer decides to have more than one function in any module, the structure chart should show them in the same block. However, the HIPO Hierarchy chart would still show all the functions in separate blocks.) The output of the general program design is the input for the detailed module design. The HIPO input-process-output chart is useful for describing and designing each module.

Structured design considerations could be used to review program designs in a walk-through environment.[12] These concepts are also useful for evaluating alternative ways to comply with the requirement of structured programming for one-page segments.[7]

Structured design reduces the effort needed to fix and modify programs. If all programs were written in a form where there was one module, for example, which retrieved a record from the master file given the key, then changing operating systems, file access techniques, file blocking, or I/O devices would be greatly simplified. And if *all* programs in the installation retrieved from a given file with the same module, then one properly rewritten module would have *all* the installation's programs working with the new constraints for that file.

However, there are other advantages. Original errors are reduced when the problem at hand is simpler. Each module is self-contained and to some extent may be programmed independently of the others in location, programmer, time, and language. Modules can be tested before all programming is done by supplying simple "stub" modules that merely return preformatted results rather than calculating them. Modules critical to memory or execution overhead can be optimized separately and reintegrated with little or no impact. An entry or return trace-module becomes very feasible, yielding a very useful debugging tool.

Independent of all the advantages previously mentioned, structured design would *still* be valuable to solve the following problem alone. Programming can be considered as an art where each programmer usually starts with a blank canvas—techniques, yes, but still a blank canvas. Previous coding is often not used because previous modules usually contain, for example, *at least* GET and EDIT. If the EDIT is not the one needed, the GET will have to be recoded also.

Programming can be brought closer to a science where current work is built on the results of earlier work. Once a module is written to get a record from the master file given a key, it can be used by all users of the file and need not be rewritten into each

succeeding program. Once a module has been written to do a table search, anyone can use it. And, as the module library grows, less and less new code needs to be written to implement increasingly sophisticated systems.

Structured design concepts are not new. The whole assembly-line idea is one of isolating simple functions in a way that still produces a complete, complex result. Circuits are designed by connecting isolatable, functional stages together, not by designing one big, interrelated circuit. Page numbering is being increasingly sectionalized (e.g., 4–101) to minimize the "connections" between written sections, so that expanding one section does not require renumbering other sections. Automobile manufacturers, who have the most to gain from shared system elements, finally abandoned even the coupling of the windshield wipers to the engine vacuum due to effects of the engine load on the performance of the wiping function. Most other industries know well the advantage of isolating functions.

It is becoming increasingly important to the data-processing industry to be able to produce more programming systems and produce them with fewer errors, at a faster rate, and in a way that modifications can be accomplished easily and quickly. Structured design considerations can help achieve this goal.

CITED REFERENCES AND FOOTNOTES

1. This method has not been submitted to any formal IBM test. Potential users should evaluate its usefulness in their own environment prior to implementation.
2. L. L. Constantine, *Fundamentals of Program Design*, in preparation for publication by Prentice-Hall, Englewood Cliffs, New Jersey.
3. G. J. Myers, *Composite Design: The Design of Modular Programs*, Technical Report TR00.2406, IBM, Poughkeepsie, New York (January 29, 1973).
4. G. J. Myers, "Characteristics of composite design," *Datamation* **19**, No. 9, 100–102 (September 1973).
5. G. J. Myers, *Reliable Software through Composite Design*, to be published Fall of 1974 by Mason and Lipscomb Publishers, New York, New York.
6. HIPO – Hierarchical Input-Process-Output documentation technique. Audio education package, Form No. SR20-9413, available through any IBM Branch Office.
7. F. T. Baker, "Chief programmer team management of production programming," *IBM Systems Journal* **11**, No. 1, 56–73 (1972).
8. The use of the HIPO Hierarchy charting format is further illustrated in Figure 6, and its use in this paper was initiated by R. Ballow of the IBM Programming Productivity Techniques Department.
9. L. A. Belady and M. M. Lehman, *Programming System Dynamics or the Metadynamics of Systems in Maintenance and Growth*", RC 3546, IBM Thomas J. Watson Research Center, Yorktown Heights, New York (1971).
10. L. L. Constantine, "Control of sequence and parallelism in modular programs," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **32**, 409 (1968).
11. G. M. Weinberg, *PL/I Programming: A Manual of Style*, McGraw-Hill, New York, New York (1970).
12. *Improved Programming Technologies: Management Overview*, IBM Corporation, Data Processing Division, White Plains, New York (August 1973).