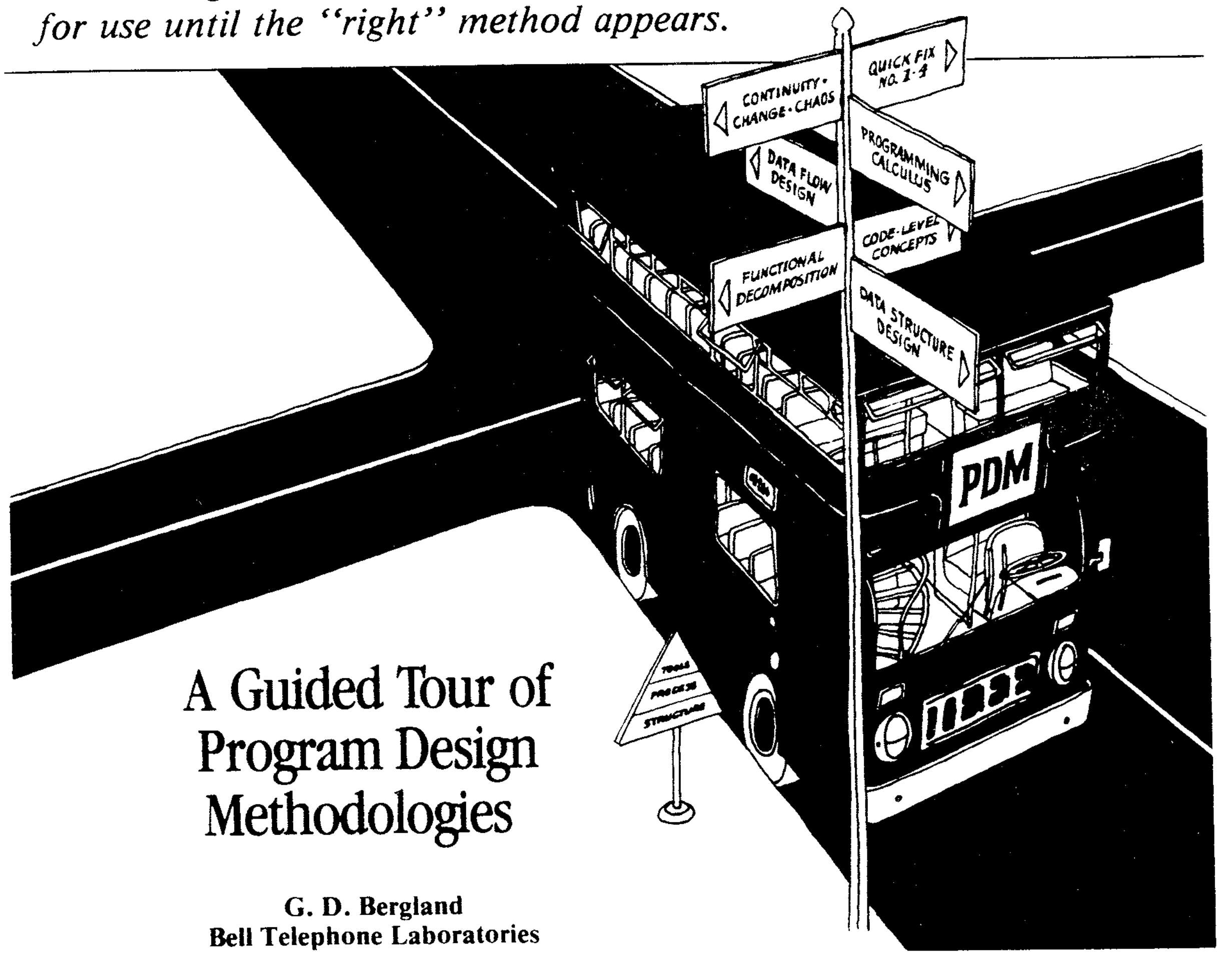
After describing, applying, comparing, and evaluating four major methodologies, this guided tour concludes with an interim procedure for use until the "right" method appears.



A Guided Tour of Program Design Methodologies

G. D. Bergland
Bell Telephone Laboratories

Much as a building architect specifies the structure and construction of a building (see Figure 1), the software architect must specify the structure and construction of a program. This guided tour examines some of the concepts, techniques, and methodologies that can aid in this task.

During this guided tour, the software problem and the attempts at its solution are briefly described. Software engineering techniques are classified into three groups: those that primarily impact the program structure, the development process, and the development support tools. Structural analysis concepts are described that have their major impact at the code level, the module level, and the system level. Then, four of the major program design methodologies that have been reported in the literature are developed and compared. Functional decomposition, data flow design, data structure design, and programming calculus are described, characterized, and applied to a specific example.

While no one design methodology can be shown to be "correct" for all types of problems, these four methodologies can cover a variety of applications. Finally, an interim approach for large software design problems is suggested that may be useful until an accepted "correct" methodology comes along.

Motivation. The major motivation for looking at program design methodologies is the desire to reduce the cost of producing and maintaining software. Developing programs that are reliable enough to support nonstop computer systems can sometimes be a secondary motivation, but these applications seem to be in the minority.

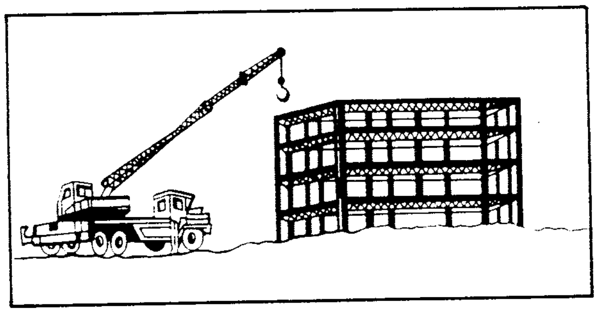


Figure 1. A good design implies a good structure.

0018-9162/81/1000-0013$00.75 © 1981 IEEE

A host of claims in technical journals, conference proceedings, and short-course advertisements herald the virtues of new design methodologies—claims such as 1.9 to 1 increase in productivity, 39 percent savings in programming costs, 80 percent reduction in bugs, etc. Their inconsistencies suggest that without a good set of metrics, you can prove anything you want. Alternatively, software development is so inefficient that almost anything can improve it.

If one were to take these claims at face value, it would seem that at least some of the problems of producing inexpensive, reliable software have been solved. Unfortunately, the benefits of structured programming, software engineering techniques, or whatever have remained either nebulous or illusive to many people. Even though structured programming has been with us for more than a decade, we are still far from having all the answers or, for that matter, even all of the questions. While it is clear that progress has been made, there is still much to be done.

Historical perspective. During the 1950's, programming was in its golden age. The approach was to take a small group of highly qualified people and solve a problem by writing largely undocumented code maintained by the people who wrote it. The result was inflexible and inextensible code, but it was adequate to the demands of the time. Buxton[1] called this "cottage industry" programming.

The software crisis hit in the 1960's. The problems got two orders of magnitude harder, and we were introduced to the problems of having many people work on large programs that were continually changing. This was the beginning of "heavy industry" programming.[1]

The structured programming of the 1970's was primarily an attempt to address the problems of heavy industry programming. This quest for a better way was started in response to rapidly rising costs[2]—more than one percent of the gross national product was being spent on software in the US—and the feeling that change was technically feasible. Thus started a variety of approaches that are collectively known as structured programming.

Techniques hierarchy. Software engineering has been defined by Parnas as multiperson construction of multiversion programs.[3] As such, there is much emphasis on the development process, its attendant support tools, and the basic structure of a program. Many of the concepts which people tend to apply first—like teams, design reviews, and program librarians—primarily involve changing the development process. Vyssotsky characterized these techniques as dealing with programmer "crowd control."[4] While they can be implemented relatively quickly, their major benefits can only be realized in the context of a well-structured program. Those techniques dealing with program structure form the foundation on which the other techniques should be applied[5] (see Figure 2). Admittedly, the support tools and the development process strongly influence the structure of the program; however, the tools should be adapted to support the desired structure, not vice versa.

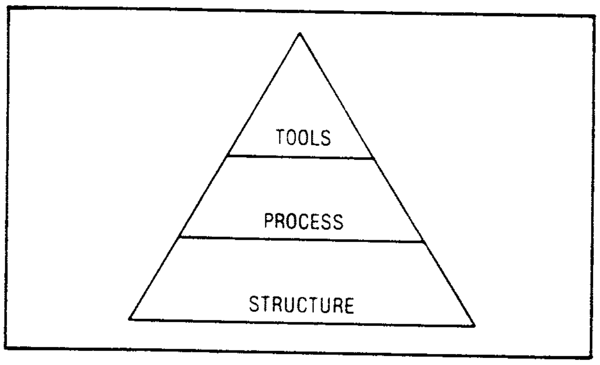


Figure 2. Software engineering techniques hierarchy.

While there are many design methodologies around, only a few of them have been extensively tested. Four methodologies used or discussed more than most are

- functional decomposition,
- data flow design,
- data structure design, and
- programming calculus.

I believe that the quality of the program structure resulting from a design methodology is the single most important determinant of the life-cycle costs for the resulting software system. Thus, before discussing the methodologies in detail, it seems worthwhile to discuss some of the concepts that play a role in evaluating the structure of a program.

Structural analysis concepts

While most structural analysis concepts apply at more than one level of a software system, it is convenient in this discussion to separate them into three categories. Concepts are discussed that have their major impact at the code level, the module level, and the software system level.

Code-level concepts. Concepts having their major impact at this level include abstraction, communication, clarity, and control flow constructs.

Abstraction. Abstraction is defined as the consideration of a quality apart from a particular instance. In programming, the application of abstraction ranks as one of the most important advances that has occurred in the last 20 years. It is the basis for high-level languages, virtual machines, virtual I/O devices, data abstractions, plus both top-down and bottom-up design.

The whole concept of bottom-up design consists of building up layers of abstract machines that get more and more powerful until only one instruction is needed to solve the problem. While people usually stop far short of defining that one superpowerful instruction, they do significantly enhance the environment in which they have to program. Device drivers, operating system primitives, I/O routines, and user-defined macros are built on the concept of abstraction. All of them raise the level at which the programmer thinks and programs.

Often, the objective is to abstract many of the complicated interactions that can occur when many users or user programs are sharing the same machine. In other instances, a virtual machine is created to hide the idiosyncrasies of a particular machine from the user so that the resulting program will be more portable.

When the modular programming era began in the 1960's, many people hoped that hundreds of reusable building-block programs could be abstracted and added to their programming libraries so that they could finally begin to "build on the work of others." Unfortunately, as Weinberg noted, "Program libraries are unique; everyone wants to put something in but no one wants to take anything out."[6]

Communication. A program communicates with both people and machines. The effect of comments can be profound. A ten-year-old program I've seen that has the comment "subtle" in it is still left alone at all costs. Although the person who wrote the program is long gone, he has left a legacy of problems that will last as long as the program.

Another program contained the comment, "They made me do it!" This comment was undoubtedly an apology for corrupting the structure of the program to provide an expedient fix to a pressing problem. Clearly, the program is a little harder to understand and modify now. This type of change is not unusual. It's the apology that's unusual.

In the long run, changes tend to obscure the structure of a program, thus making the processes of error correction and feature addition difficult and dangerous. A well-written and well-maintained program is meant to communicate its structure to the programmer as well as to give instructions to the machine. I believe that the life-cycle cost of operating a program usually depends far more on how well it communicates with people than on how fast it initially runs.

Clarity. It has been said that a person who writes English clearly can write a program clearly. In studying English, we are taught first to read and then to write. This seems to work well. In programming, however, we are usually taught only to write. I think we miss something by not learning to read programs first. At one time it was even considered fashionable to write unreadable programs. It got so bad that one language, famous for its "one-liners," was dubbed a "write-only" language.

The structure* of an article, paper, or book is very important in clearly communicating ideas. The structure of a program is equally important in communicating both the algorithm and the context of a problem solution. This structure should be apparent when one reads a program.

Clarity of program structure was obviously not the primary concern of the person who wrote the program represented in Figure 3. This program has been running for more than 10 years. Fortunately, few changes or feature enhancements have been required. In an attempt to understand how the program worked, this diagram was drawn by the last person who had to change it.

The "structuredness" of this program—and, for that matter, of any program—is not well-defined. There is still no generally accepted metric for characterizing the merit of a program structure. The unavailability of such a metric results in some strange phenomena. For example, do you know someone who writes complicated and unintelligible code, who spends long hours and late nights on it, finally getting it "done" just before the deadline—all the while letting everyone know how difficult his task is and what a hero he is for having gotten it done just in time?

In contrast, consider the neat, well-organized programmer who takes care to plan ahead, do a proper design, document her work, and get done well ahead of the deadline, with no one even aware that she was involved. How often have you thought "Boy, John has certainly earned his wings with that difficult program, while Jane hasn't had a chance to prove herself."

In the best of all worlds, the criterion of clarity could be applied quantitively. Lacking that, we'll have to stick with peer pressure applied in design reviews and code walkthroughs to ensure the clarity of the final product.

Control flow constructs. The concept of limiting the number and type of control flow constructs, to more clearly express algorithms, is now pretty generally accepted. The notation recommended by Michael Jackson[7] is shown in Figure 4.

The *sequence* and *selection* constructs shown in this figure can be generalized in the obvious way to a sequence or selection of *N* items for arbitrary *N*. The *iteration* stands for "zero or more" program executions. The advantage of

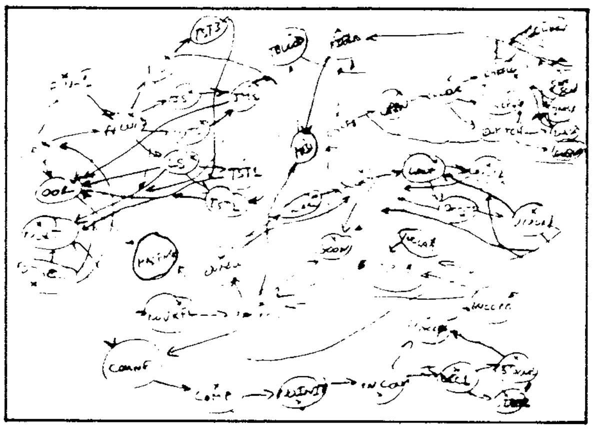


Figure 3. A clearly presented program structure?

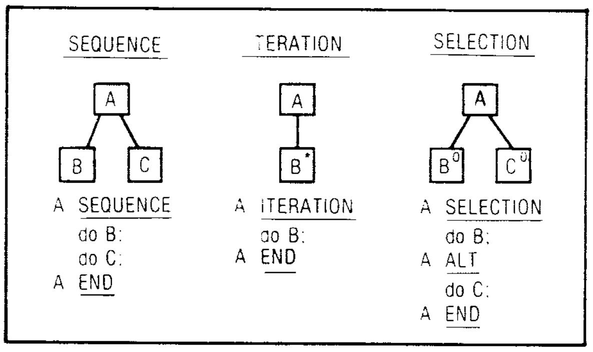


Figure 4. The three basic control flow constructs.

*An adequate but not inspired definition of "structure" is "the arrangement or interrelation of parts as dominated by the general character of the whole."

these constructs is that they tend to provide a map of the program structure rather than an itinerary of tasks.

The transformation of a program that uses these graphical constructs to structure text (also shown in Figure 4) is remarkably straightforward, as is the later transformation to a specific programming language.

Module-level concepts. Cohesion, coupling, complexity, correctness, and correspondence are concepts with major impact at the module level.

Cohesion. Cohesion is the "glue" that holds a module together. It can also be thought of as the type of association among the component elements of a module. Generally, one wants the highest level of cohesion possible. While no quantitative measure of cohesion exists, a qualitative set of levels for cohesion has been suggested by Constantine[8] and modified by Myers.[9] The levels proposed by Constantine are shown in Figure 5.

Coincidental cohesion is Constantine's lowest level of cohesion. Here, the component parts of a module are there only by coincidence. No significant relationship exists among them.

Logical cohesion is present when a module performs one of a set of logically related functions. An example would be a module composed of 10 different types of print routines. The routines do not work together or pass work to each other but logically perform the same function of printing.

Temporal cohesion is present when a module performs a set of functions related in time. An initialization module performs a set of operations at the beginning of a program. The only connection between these operations is that they are all performed at essentially the same time.

Procedural cohesion occurs when a module consists of functions related to the procedural processes in a program. Functions that can be well represented together on a flowchart are often grouped together in a module with procedural strength. Conversely, when a program is designed by using a flowchart, the resulting module often has procedural cohesion.

Communicational cohesion results when functions that operate on common data are grouped together. A data abstraction, or data cluster,[10] is a good example of a module with communicational cohesion.

Sequential cohesion often results when a module represents a portion of a data flow diagram. Typically, the modules so formed accept data fron one module, modify or transform it, and then pass it on to another module.

Functional cohesion results when every function within the module contributes directly to performing one single function. The module often transforms a single input into a single output. An example often cited is square root. This is the highest level of cohesion in the hierarchy. As such, it is desirable whenever it can be achieved.

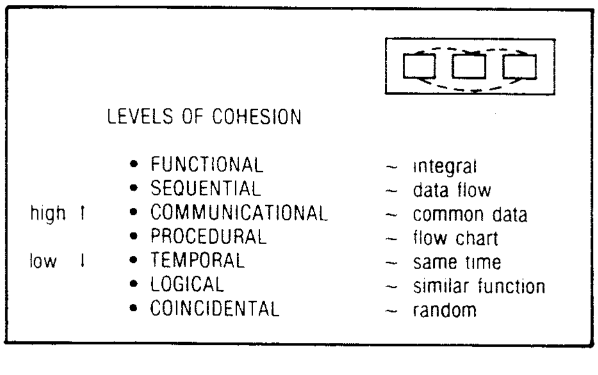
A program of any reasonable size will usually contain modules of several different levels of cohesion. Many modules simultaneously exhibit characteristics of a multiplicity of levels. Where possible, functional, sequential, and communicational strength modules should be given preference over modules with lower levels of cohesion. On a scale of 0 to 10, Yourdon[8] rates coincidental, logical, and temporal cohesion as 0, 1, and 3, respectively. Procedural cohesion would score 5. Communicational, sequential, and functional cohesion would score 7, 9, and 10, respectively.

While levels of cohesion can be useful guides in evaluating the structure of a program, they don't provide a clear-cut method for attaining high levels of cohesion. Furthermore, levels of cohesion do not allow us to say that program A is right and program B is wrong. They do, however, represent a definite step forward. Before levels of cohesion were introduced, there was no recognized basis for comparison. Now, at least one can say that structure A is probably better than structure B.

Coupling. Coupling is a measure of the strength of interconnection (i.e., the communication bandwidth) between modules. In Figure 6, two program structures are represented that would result in significantly different degrees of coupling.

High coupling among program modules results when a problem is partitioned in an arbitrary way such as cutting off sections of a flowchart. This method of chopping up a large program often complicates the total job because of the resultant tight coupling between the pieces. This latter type of partitioning leads to "mosaic" modularity.[11]

The other extreme in structuring a program is to consider only pure tree structures. These structures give rise to the concept of hierarchical modularity and provide many advantages for abstraction, testing, and later mod-


LEVELS OF COHESION

high ↑	• FUNCTIONAL	– integral
	• SEQUENTIAL	– data flow
	• COMMUNICATIONAL	– common data
	• PROCEDURAL	– flow chart
low ↓	• TEMPORAL	– same time
	• LOGICAL	– similar function
	• COINCIDENTAL	– random

Figure 5. Levels of cohesion.

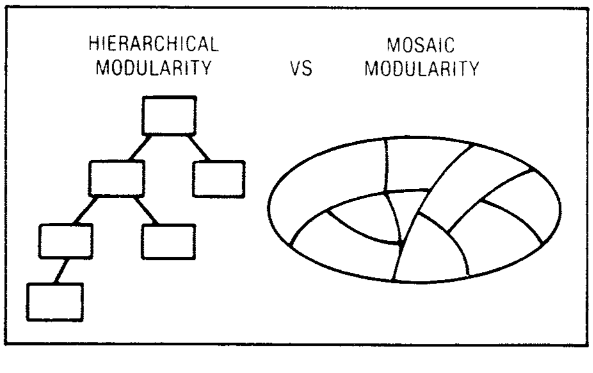


Figure 6. Partitioning method affects level of coupling.

ification. Jackson would accuse you of "arboricide" (the killing of trees) whenever you deviate from a pure hierarchical tree structure. Brooks has said, 'I am persuaded that top-down design (incorporating hierarchy, modularity, and stepwise refinement) is the most important new programming formalization of the decade."[12] And Dijkstra has said that "the sooner we learn to limit ourselves to hierarchical program constructs the faster we will progress."[13]

Modular programs can be characterized as

- implementing a single independent function,
- performing a single logical task,
- having a single entry and exit point,
- being separately testable, and
- being entirely constructed of modules.

When these rules are followed, a set of nested modules result that can be connected in a hierarchy to form large programs. In an attitude survey,[14] users perceived that modular programs were easier to maintain and change, easier to test, and more reliable. The major perceived disadvantage was the feeling that the final program was less efficient than it could have been.

When modularity is used without hierarchy, one can only implement independent functions that can be executed in sequence, which corresponds to drawing circles around portions of a flowchart. Although this approach tends to work on small programs, it can seldom be applied to complex programs without seriously compromising module independence, connectivity, and testability. Only when the concepts of modular programming are combined with the concepts of hierarchical program structure can one implement arbitrarily complex functions and still maintain module integrity.

Modularity can be applied without hierarchy in cases that lend themselves naturally to the efficient use of a very high level language. Very high level language statements are examples of functions that can be implemented relatively independent of each other but still be strung together sequentially in a useful form. Unfortunately for most applications, the design of a convenient and efficient very high level language is difficult.

Hierarchical modularity forms an extremely attractive foundation for most of the other software engineering techniques. While some of these techniques can be used without having a hierarchical program structure, the primary benefit can only be gained when the techniques are used as a unit and build on each other. Specifically, a hierarchical modular program structure enhances top-down development, programming teams, modular programming, design walkthroughs, and other techniques that deal with improving the development process.

Complexity. The control of program complexity is the underlying objective of most of the software engineering techniques. The concept of "divide and conquer" is important as an answer to complexity, provided it is done correctly. When a program can be divided into two independent parts, complexity is reduced dramatically, as shown in Figure 7.

Consider program A, where you have access to only the input and the output.[9] A noble goal would be to completely test this program by executing each unique path. In the example shown, there are approximately 250 billion unique paths through this module. If you were capable of performing one test each millisecond, it would take you eight years to completely test all of the unique paths. If, however, you had knowledge of what was inside the program and recognized that it could be partitioned into two independent modules B and C, which have low connectivity and coupling, your testing job could be reduced. To test both of these modules separately requires that you only test the one million unique paths through each module. At one millisecond per test, these tests would take a total of only 17 minutes.

In this particular example, from a testing viewpoint, it is clearly worth trying to partition the problem so that small, independently testable modules can be dealt with instead of just the input and output of a large program. Unfortunately, partitioning most programs into independently testable iodules requires much more work than simply drawing small circles around portions of the flowchart.

It should also be clear from this example that the testing problem is best solved during the design stage. It is impossible to exhaustively test any program of significant size. Testing is experimental evidence.[7] It does not verify correctness. It simply raises your confidence.

Correctness. A "correct" program is one that accurately implements the specification. A "correct" program often has limited value since the specifications are in error. Again, correctness cannot be verified by testing. Searching for errors is like searching for mermaids. Just because you haven't seen one doesn't mean they don't exist.

It is also unfortunate that, for most problems, mathematical proofs of correctness are as difficult to produce as a correct program. Some people can write and prove their program simultaneously. For most of us, however, day-to-day proving of programs is still a long way off.

The most promising approach for the near future may lie in finding a constructive proof of correctness. We will really have something if a design methodology can be

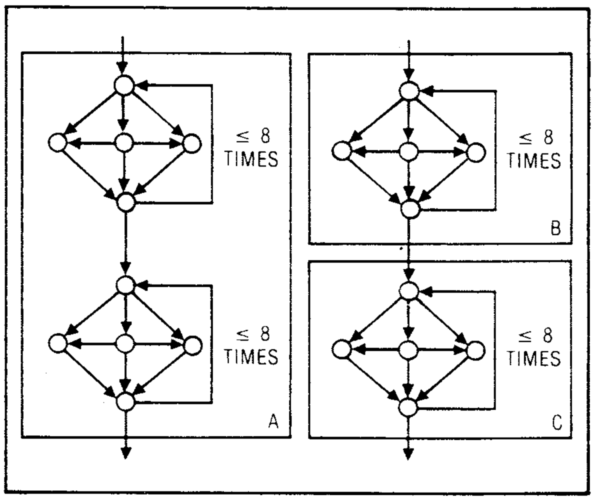


Figure 7. Control of complexity.

found that leads one through the design process step-by-step and guarantees the correctness of the final program if each of the steps has been done correctly. While I don't remember that 321 × 25 is 8025, I do remember that 5 × 1 = 5, and 5 × 2 = 10, and 5 × 3 = 15, etc. Knowing these values and the steps of multiplication, I can rest assured that 8025 is indeed the correct answer. If only a program design process existed that was as foolproof and easy to apply.

Without such a process, we must live with a limitless capacity for producing error. Weinberg once pointed out that errors can be produced in arbitrarily large numbers for an arbitrarily low cost.

Correspondence. In Jackson's view, perhaps the most critical factor in determining the life-cycle cost of a program is the degree to which it faithfully models the problem environment[7]—that is, the degree to which the program model corresponds to the real world. All too often, a small local change in the problem environment results in a large diffuse change in the program.

The world is always bigger than the program specification says it is, but a specification can always be extended if it corresponds to reality. Since users tend to be gradualists, the changes in a realistic problem model will tend to be gradual. If the program structure is formed around the static instead of the dynamic properties of the problem, it should prove to be more resilient to changes.

While a program's model of the world cannot be complete, it must at least be useful and true. If these criteria are met, many maintenance and feature enhancement problems will be avoided in the future.

System-level concepts. Concepts of major impact at this level include consistency, connectivity, continuity, change, chaos, optimization and packaging.

Consistency. An important objective of a good design methodology is that it should produce a consistent program structure independent of whoever is applying it. Three different programs—created by using the same design methodology to model the same problem environment—should have the same basic structure. Unless consistent designs can be achieved, there can never be a true right or wrong structure for a given problem solution.

One problem with most design methodologies is that there is no one consistently obtainable solution. Instead, designers seem to pull unique solutions out of the air. Consequently, there is never discussion of a solution being right or wrong—only discussion of my style versus your style.

Connectivity. The harmful effects of high connectivity on system modifiability can best be illustrated by using an analogy.[15]

Consider a system composed of 100 light bulbs. Each light in the system can either be on or off. Connections are made between the light bulbs so that if the light is on, it has a 50 percent chance of going off in the next second. If the light bulb is off, it has a 50 percent chance of going on in the next second—provided one of the lights to which it is connected in on. If none of the lights connected to it is on, the light stays off. Sooner or later, this system of light bulbs will reach an equilibrium state in which all of the lights go off and stay off.

The average length of time required for this system to reach equilibrium is solely a function of the interconnection pattern of the lights. In the most trivial interconnection pattern, all of the lights operate independently. None of them is connected to any of its neighbors. Here, the average time for the system to reach equilibrium is approximately the time required for any given light to go off—about two seconds. Thus, the system can be expected to reach equilibrium in a matter of seconds.

At the other extreme, consider the case when each light in the array is fully connected to all other lights in the array—that is, assume that there is a connectivity matrix for the lights similar to the program connectivity matrix shown on the left side of Figure 8, where *N* is equal to 100. The array on the left side of the figure then describes the connectivity matrix of the lights and shows that every light is connected to every other light. In this case, the length of time required for the system to reach equilibrium is 10^{22} years. This is a very long time when you consider that the current age of the universe is only 10^{10} years.

Now, consider one final interconnection pattern in which the set of 100 lights is partitioned into 10 sets of 10 lights each, with no connections between the sets, but with full interconnection within each set. In this case, the time required for the system of lights to reach equilibrium is about 17 minutes. This example dramatically shows the effect of connectivity. In terms of the concepts presented earlier, this example corresponds to high cohesion within each module and low coupling between modules.

Much as proper physical partitioning can dramatically reduce the time required for the system of lights to reach equilibrium, proper functional partitioning can dramati-

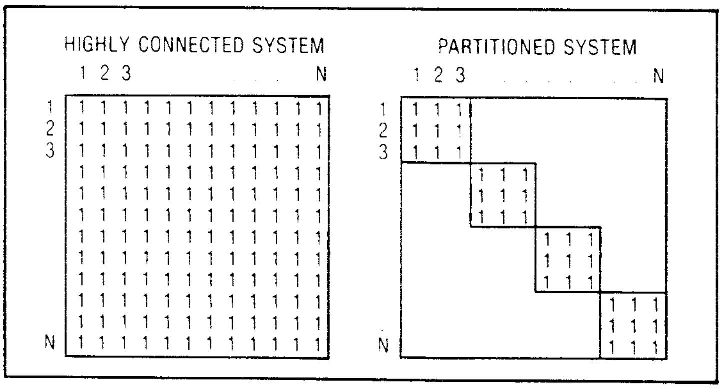


Figure 8. Low connectivity implies low maintenance costs.

LAW OF CONTINUING CHANGE: A SYSTEM THAT IS USED UNDERGOES CONTINUING CHANGE UNTIL IT IS JUDGED MORE COST-EFFECTIVE TO FREEZE AND RECREATE IT.

LAW OF INCREASING UNSTRUCTUREDNESS: THE ENTROPY (DISORDER) OF A SYSTEM INCREASES WITH TIME UNLESS SPECIFIC WORK IS EXECUTED TO MAINTAIN OR REDUCE IT.

Figure 9. Continuity/change/chaos.

cally reduce the time required for a program that is being debugged to reach stability.

Continuity/change/chaos. As noted by Belady and Lehman,[16] a large program often seems to live a life of its own, independent of the noble intentions of those trying to control it. Two important observations are summarized in the Law of Continuing Change and the Law of Increasing Unstructuredness shown in Figure 9.

These laws dramatize the key role played by the program structure during the life cycle of software systems. The natural order of things is to produce disorder. If the program structure is unclear from the beginning, things will only get worse later. These two laws coupled with a poor program structure have produced many of the maintenance-cost horror stories.

Optimization and packaging. All too often, people confuse packaging and design. Design is the process of partitioning a problem and its solution into significant pieces. Optimization and packaging consist of clustering pieces of a problem solution into computer load modules that run within system space and time requirements without unduly compromising the integrity of the original design.[8]

At least three different types of modules must be considered in programming—functional modules, data modules, and physical modules. Packaging is concerned with placing functional modules and data modules into physical modules. In packaging a program, several of these pieces of the program may be put together as one load module or may even be written together as one program.

It is in the packaging phase of a design that optimization should be considered for the first time. This phase is done at the end, and great care should be taken to preserve the program structure that you have worked so hard to create. In Jackson's words, "It is easy to make a program that is right, faster. It is difficult to make a program that is fast, right." Once an optimization has been cast in code, it's like concrete. It is very difficult to undo.

Functional decomposition

Functional decomposition is simply the divide-and-conquer technique applied to programming, as shown in Figure 10. Various forms of functional decomposition have been popularized by a host of people including Dijkstra, Wirth, Parnas, Liskov, Mills, and Baker.[17-22]

By viewing the stepwise decomposition of the problem and the simultaneous development and refinement of the program as a gradual progression to levels of greater and greater detail, we can characterize functional decomposition as a *top-down* approach to problem-solving. Conversely, we can form and layer groups of instruction sequences together into "action clusters," starting at the atomic machine instruction level and working our way up to the complete solution. This approach leads to a *bottom-up* method.[21]

Often, the preferred strategy is to shift back and forth between top-down functional decomposition and the bottom-up definition of a virtual machine environment.

Design strategy. The design process can be divided into the following steps:[19]

(1) Clearly state the intended function.
(2) Divide, connect, and check the intended function by reexpressing it as an equivalent structure of properly connected subfunctions, each solving part of the problem.
(3) Divide, connect, and check each subfunction far enough to feel comfortable.

In following this procedure, the key to successful program design is rewriting followed by more rewriting. Every effort should be made at each step to conceive and evaluate alternate designs.

A useful mind set is to pretend that you are programming on a machine that has a language powerful enough to solve your problem in only a handful of commands. In your level 1 decomposition, you simply write down that handful of commands and you have a complete program. In your level 2 decomposition, you try to refine each of your level 1 instructions into a set of less powerful instructions. By continuing to successively refine each instruction, one level at a time, you eventually get to a program that can be executed on your own real computer. In carrying out this process, you will have decomposed the problem into its constituent functions by "stepwise refinement."

There are several problems involved in applying this technique. First, the method specifies that a functional decomposition be performed, but it does not say what you are decomposing with respect to. One can decompose with respect to time order, data flow, logical groupings, access to a common resource, control flow, or some other criterion. If you decompose with respect to time, you get modules like initialize, process, and terminate, and you have a structure with temporal cohesion. If you cluster functions that access a shared data base, you have made a start toward defining abstract data types and will get communicational cohesion. If you decompose using a data flow chart, you may end up with sequential cohesion. If you decompose around a flowchart, you will often end up with logical cohesion. The choice of "what to decompose with respect to" has a major effect on the "goodness" of the resulting program and is therefore the subject of much controversy.

The major advantage of functional decomposition is its general applicability. It has also been used by more people longer than any of the other methods discussed. The dis-

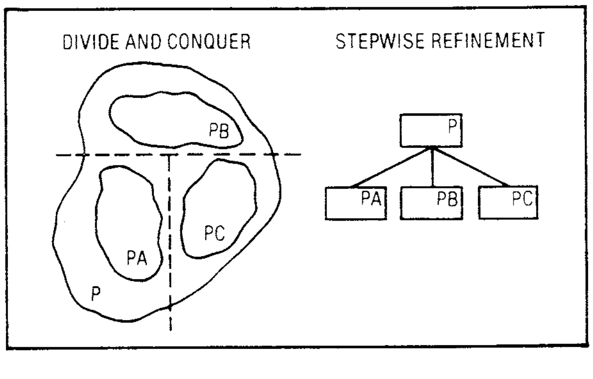
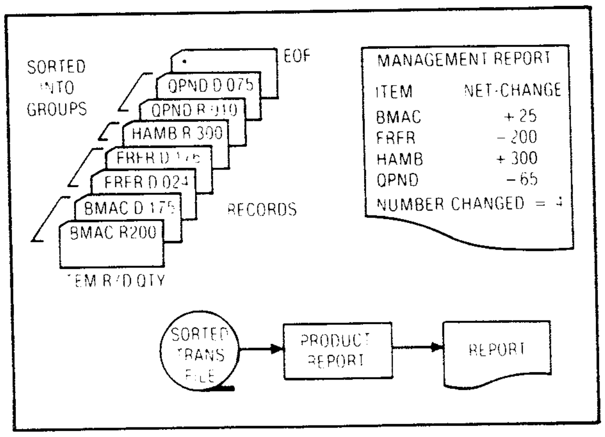


Figure 10. Functional decomposition.

advantages are its unpredictability and variability. The chance of two programmers independently solving a given problem in the same way are practically nil. Thus, each new person exposed to a program starts by saying, "This isn't the way I would have done it, but. . ." To some extent, this problem can be reduced by using functional decomposition in combination with some other technique that determines what each function should be composed with respect to.

McDonald's example. The McDonald's functional decomposition solution, presented below, is patterned (with permission) after a story called "Getting it Wrong" that has been related by Michael Jackson on numerous occasions in his short courses and seminars. The McDonald's frozen-food warehouse problem is, of course, entirely fictitious.

Problem specification. McDonald's frozen-food warehouse receives and distributes food items. Each shipment received or distributed is recorded on a punched card that contains the name of the item, the type of shipment (R for received, D for distributed), and the quantity of each item affected. These transaction cards are sorted by another program and appear grouped in alphabetical order by item name. A management report, showing the net change in inventory of each item, is produced once a week. The input file and output report formats are shown in Figure 11.



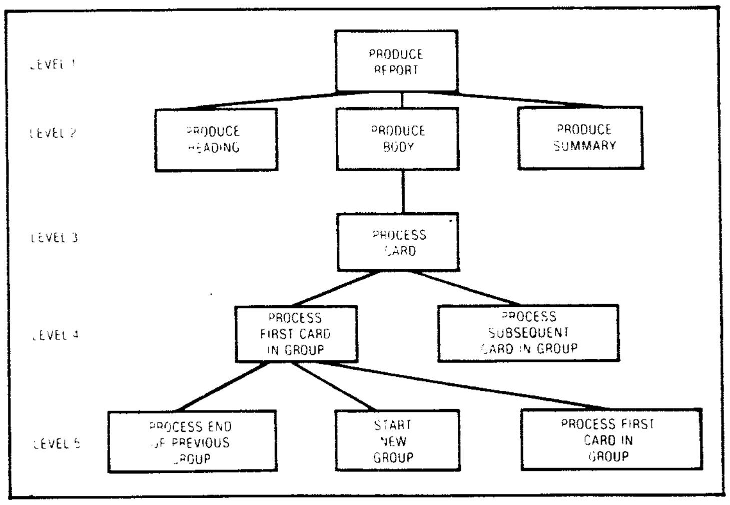


Figure 12. A five-level functional decomposition.

Design phase. The hero who originally designed this program is named Ivan. Ivan is a very "with it" fellow. He swore off GO TOs years ago. His code is structured like the Eiffel Tower. He can whip out a neatly indented structured program in nothing flat. In doing his design, Ivan was careful to do the five-level, hierarchical, functional decomposition shown in Figure 12.

In this figure, PRODUCE REPORT is shown to be a sequence of PRODUCE HEADING followed by PRODUCE BODY followed by PRODUCE SUMMARY. PRODUCE BODY is shown to be an iteration of PRODUCE CARD that is a selection between PROCESS FIRST CARD IN GROUP and PROCESS SUBSEQUENT CARD IN GROUP.

Clearly, recognizing the first card of each item group is important. Once this card is found, everything else falls into place.

At this point, it may be worth examining the structure of Ivan's program. The obvious question is, "Is this a good decomposition?" The obvious reply is, "Good with respect to what?"

If we apply the concept of cohesion to this structure, it might seem that the level 1 PRODUCE REPORT module has temporal cohesion since the PRODUCE HEADING module is something like an initialization module and the PRODUCE SUMMARY module is something like a terminate module. On the other hand, one could argue that the heading, body, and summary are such integral parts of the report that this is really functional cohesion.

Likewise, the PROCESS CARD module seems to have been partitioned along temporal lines as well. On the other hand, PROCESS CARD seems to be an integral part of the PRODUCE BODY module, so maybe PRODUCE BODY is also functionally cohesive.

As you can tell by the preceding examination, while levels of cohesion may constitute an improvement over having no basis for comparison, they are still difficult to apply consistently. In cases where more than one type of cohesion seem to be present, the rule[8] is to assume that it is really the higher of the two levels.

Ivan wrote the level 1, 2, and 3 programs shown in Figure 13. The level 4 decomposition shown in Figure 14 started to look like a finished product. The level 5 decomposition shown in Figure 15 was the finished product.

In his normal thorough manner, Ivan volume-tested his program and turned it over to the user. It was perfect except for one small glitch. You see, the systems programmers were still playing with the compiler and obviously hadn't fixed all the errors. As a result, the program worked fine, but some garbage appeared on the first line of the output immediately after the headings. It was believed, of course, that this would disappear as soon as they fixed that %&" **? > compiler.

Friendly user phase. Now, Ronald McUser is a friendly sort of person. Since he was in a hurry to use the program, he did not complain about the small glitch that would, of course, disappear very soon. Ronald's boss, Big Mac, however, failed to see the humor of it all. The compiler had been fixed for three months now, and the management report that went to the board of directors still had that garbage in it.

Finally, one day, Ronald could stand it no longer. The program *had* to be fixed. When Ronald arrived, Ivan was in his cubicle, listening to an audio cassette of Dijkstra's Turing lecture. He, of course, didn't learn anything; that's the way he had always done his designs. Ronald showed Ivan a printout.

After a few MMMs and AAAHHHHs and AAAHHHAAAs, he saw the problem. This first time through the program, there was no previous group. Thus, the output was just random data. The solution to any first-time-through problem is, of course, obvious. Add a first-time switch (see Figure 16).

Maintenance phase. Six months later, our hero was in McDonald's "think room" when Ronald McUser came in and said, "I put 80 transactions in last week and nothing came out!" Our hero looked at the printout and saw that, indeed, only the heading had come out.

Ivan knew immediately what the problem was. It must be a hardware problem. After all, his program had been running nearly a year now with only one small complaint. Ivan spent most of the night running hardware diagnostics until Steve Saintly, a keypunch operator, wandered by and said, "That nationwide special on Big Macs last week was all we handled. Everything else had to wait."

A little later, Sally Saintly came by and said, "Isn't it about time you included those new Zebra sodas in the management report?"

By this time, Ivan was very discouraged with his diagnostics, so he followed last week's inputs through the code. "Horrors! There was only one item group processed last week—Big Macs!"

As Ivan soon discovered, the last item group was never processed. Since only one item group was processed all week, nothing was output. Up until this time, only Zebra sodas had been skipped. Since they were not a big winner, it seems that no one had even cared that they had been left off. In fact, everyone assumed they were being left off on purpose. Ivan's solution is shown in Figure 17.

```
LEVEL 1     P: PRODUCE REPORT;
               STOP;

LEVEL 2     P: PRODUCE HEADING;
               PRODUCE BODY;
               PRODUCE SUMMARY;
               STOP;

LEVEL 3     P: PRODUCE HEADING;
                 READ STF;
                 DO WHILE (NOT EOF-STF);
                     PROCESS CARD;
                     READ STF;
                 END;
               PRODUCE SUMMARY;
               STOP;
```

Figure 13. Steps in functional decomposition.

```
P: PRODUCE HEADING;

      READ STF;
      DO WHILE (NOT EOF—STF);
          IF FIRST CARD IN GROUP THEN
             PROCESS FIRST CARD IN GROUP;
             ELSE PROCESS SUBSEQUENT CARD IN GROUP;
          READ STF;
      END;

   PRODUCE SUMMARY;
   STOP;
```

Figure 14. Level 4 functional decomposition.

```
P: PRODUCE HEADING;

      READSTF;
      DO WHILE (NOT EOF—STF);
        IF FIRST CARD IN GROUP THEN
           DO;
                  PROCESS END OF PREVIOUS GROUP;

                  PROCESS START OF NEW GROUP;
                  PROCESS CARD;

           END;
        ELSE DO: PROCESS CARD;
             END;
        READ STF;
      END;

   PRODUCE SUMMARY;
   STOP;
```

Figure 15. The final functional decomposition.

```
P: PRODUCE HEADING;
   SW1 = 0;
      READSTF;
      DO WHILE (NOT EOF—STF);
        IF FIRST CARD IN GROUP THEN
           DO; IF SW1 = 1 THEN
               DO: PROCESS END OF PREVIOUS GROUP;
               END. SW1 = 1
                  PROCESS START OF NEW GROUP;
                  PROCESS CARD;

           END;
        ELSE DO: PROCESS CARD;
             END;
        READ STF;
      END;

P: PRODUCE SUMMARY;
   STOP;
```

Figure 16. Quick fix no. 1.

Meanwhile, Sally Saintly asked, "Why didn't you see that during all the volume-testing you did? You tied up the machine for most of a day."

The answer again is obvious. In volume-testing, you put in thousands of inputs but don't look at the output.

Passing the baton. Six months later, Ivan was feeling pretty pleased with himself. He had just turned the program over to a new hire. There would still be some training, but everything should go well. After all, hadn't the program run for nearly a year and a half with only a couple of small problems? Suddenly, Ronald burst in, "I thought you fixed this first-line problem. Here it is again."

Ivan knew immediately what the problem was. The new program librarian they had forced him to use had put in an old version of the program without his first patch.

```
P: PRODUCE HEADING:
  SW1: = 0:
      READSTF:
      DO WHILE (NOT EOF—STF):
         IF FIRST CARD IN GROUP THEN
            DO: IF SW1 = 1 THEN
                  DO: PROCESS END OF PREVIOUS GROUP:
                  END: SW1: = 1:
                        PROCESS START OF NEW GROUP:
                        PROCESS CARD:

            END:
         ELSE DO: PROCESS CARD:
               END:
         READ STF:
      END:
                     PROCESS END OF LAST GROUP

PRODUCE SUMMARY:
STOP:
```

Figure 17. Quick fix no. 2.

After many heated comments plus a core dump, Ivan was still baffled. Finally, in desperation, he sat down to look at the input data and found out there wasn't any. Last week, a trucker's strike had shut down the warehouse. Nothing came in; nothing went out. They ran the program anyway. Good grief, who would have thought that they would run the program with no inputs!

The problem, as it turns out, was that the new PROCESS END OF LAST GROUP module needed protection just like the PROCESS END OF PREVIOUS GROUP module had before. Since that first-time switch had worked so nicely earlier, it was clearly the solution to apply again (see Figure 18).

We all know that Ivan's troubles are over now. Or are they? Two months later, Sally Saintly came in and said, "Where are the Zippo sandwiches? They were in for two months, but now they've suddenly disappeared from the report."

After complaining that the new hire was supposed to be maintaining that program now, Ivan looked at the input data and noticed that only one order per item had been issued during the whole run.

"What happened?" he exclaimed.

It seems that a new manager, Mary Starr, had started a new policy to try to get things better organized. She had asked each of the stores to place only one order a day instead of placing orders at random. She had also said that it would be nice if they could schedule things so that the warehouse had to be concerned only with receiving one particular item on one day and with distributing that item the next day. In addition, she wanted the management report program run once a day from now on. The effect on Ivan's program was that Zippo sandwiches was dropped.

Instead of moving the set for SW2, the safest thing to do—according to the principles of defensive programming—is to add an extra set. Since you don't know what you're doing, you never touch a previous fix—just add a new one (see Figure 19).

Now we can all rest assured that Ivan's program works, right?

```
P: PRODUCE HEADING:
  SW1: = 0:    SW2: = 0
      READSTF:
      DO WHILE (NOT EOF—STF):
         IF FIRST CARD IN GROUP THEN
            DO: IF SW1 = 1 THEN
                  DO: PROCESS END OF PREVIOUS GROUP:
                  END: SW1: = 1:
                        PROCESS START OF NEW GROUP:
                        PROCESS CARD:

            END:
         ELSE DO: PROCESS CARD: SW2: = 1
               END:
         READ STF:
      END: IF SW2: = 1 THEN
            DO: PROCESS END OF LAST GROUP:
            END:
PRODUCE SUMMARY:
STOP:
```

Figure 18. Quick fix no. 3

```
P: PRODUCE HEADING:
  SW1: = 0: SW2: = 0:
      READSTF:
      DO WHILE (NOT EOF—STF):
         IF FIRST CARD IN GROUP THEN
            DO: IF SW1 = 1 THEN
                  DO:   PROCESS END OF PREVIOUS GROUP:
                  END: SW1: = 1:
                        PROCESS START OF NEW GROUP:
                        PROCESS CARD:
                        SW2: = 1

            END:
         ELSE DO: PROCESS CARD: SW2: = 1.
               END:
         READ STF:
      END: IF SW2: = 1 THEN
            DO: PROCESS END OF LAST GROUP:
            END:
PRODUCE SUMMARY:
STOP:
```

Figure 19. Quick fix no. 4.

The effect of all of these changes on the program structure is shown in Figure 20.

What was Ivan's major sin? Simply that the program structure didn't correspond to the problem structure. In the original data, there existed something called an item group. There is no single component in Ivan's program structure that corresponds to an item group. Thus, the actions that should be performed once per group, before a group, or after a group have no natural home. They are spread all over the program, and we have to rely on first-time switches and the like to control them. Instead of components that start a new group, process a group, or process the end of a group, we have a mess.

In Jackson's words, when this correspondence is not present, the program is not poor, suboptimal, inefficient, or tricky. It's *wrong*. The problems we have seen are, in reality, nothing but self-inflicted wounds stemming from an incorrect program structure.

Data flow design

The data flow design method first proposed by Larry Constantine[8] has been advocated and extended by Ed Yourdon[8] and Glen Myers.[9] It has been called by several different names, including "transform-centered design" and "composite design." In its simplest form, it is nothing more than functional decomposition with respect to data flow. Each block of the structure chart is obtained by successive application of the engineering definition of a black box that transforms an input data stream into an output data stream. When these transforms are linked together appropriately, the computational process can be modeled and implemented much like an assembly line that merges streams of input parts and outputs streams of final products.[8,9]

Design strategy. The first step in using the data flow design method is to draw a data flow graph (see Figure 21). This graph is a model of the problem environment that is transformed into the program structure. An example of its use will be given later.

While the modules in functional decomposition often tend to be attached by a "uses" relationship, the bubbles in a data flow graph could be labeled "becomes." That is, data input A "becomes" data output B. Data B becomes C, C becomes D, etc. The only shortcoming of this decomposition is that it tends to produce a network of programs—not a hierarchy of programs. Yourdon and Constantine solve this problem by simply picking the data flow graph up in the middle and letting the input and output data streams "hang down" from the middle. At each level, a module in one of the input or output data streams can be factored into a "get" module, a "transform" module, and a "put" module. By appropriate linking, a hierarchy is formed. Thus, once the data flow graph is drawn, the hierarchical program structure chart can be derived in a relatively mechanical way.

Given the data flow graph of Figure 21, the modules of the structure chart are defined as GET A, GET B, and GET C. Also defined are the modules that transform A into B, B into C, C into D, and so on. The output module is illustrated by the PUT D module. The GET modules are called "afferent modules" and the PUT modules are called "efferent modules."[8] The TRANSFORM C TO D module is known as the "central transform."

The data flow design method can be broken into the following four basic steps:

(1) Model the program as a data flow graph.
(2) Identify afferent, efferent, and central transform elements.
(3) Factor the afferent, efferent, and central transform branches to form a hierarchical program structure.
(4) Refine and optimize.

This procedure is represented schematically in Figure 22.

Note that while the connections between modules in the data flow graph context were motivated by a "becomes" or "consumes/produces" relationship, the factoring procedure leads to modules that are connected by a "calls/is called by" relationship. Thus, the hierarchy formed is really being artificially imposed by the scheduling and has little to do with modeling the problem in

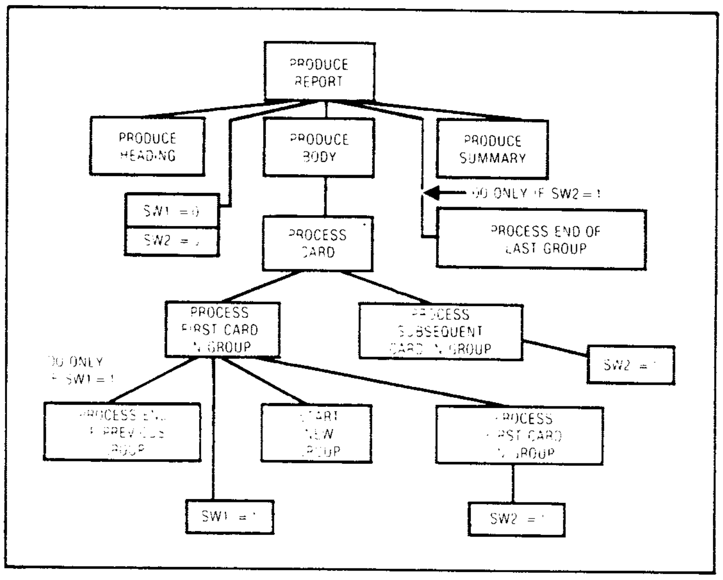


Figure 20. Functional decomposition with quick fixes.

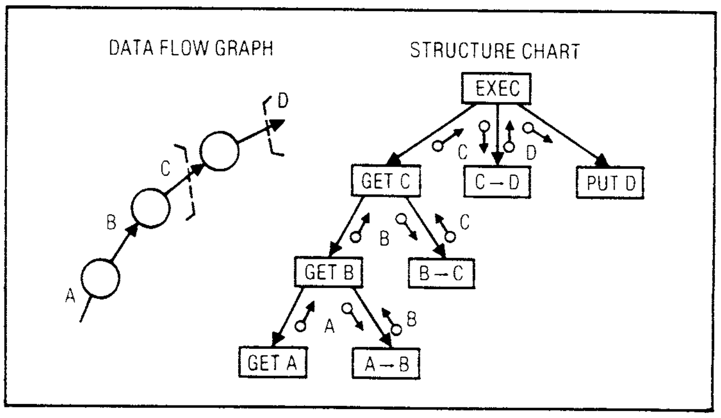


Figure 21. Data flow design method.

hierarchical fashion. This contrasts with both the "uses" relationship of decomposition with respect to function and the "is composed of" relationship that motivates a data structure design.

Also note that a lot of data passes between modules in the structure in assembly-line fashion. This results in sequential cohesion. By Constantine's measure of goodness, the data flow design method produces a very good program structure.

The act of concentrating the I/O functions in the afferent and efferent "ears" of the program structure may or may not produce a structure that models the problem environment accurately. This partitioning often seems artificial to me and would seem to violate the principle of correspondence.

The central transform is located between the data flow graph's most abstract input and most abstract output. While the graph shows it as simply transforming Cs into Ds, it often requires a sophisticated functional decomposition in its own right. That is, except for the "ears" which come from the data flow graph, one is forced right back to the art of functional decomposition.

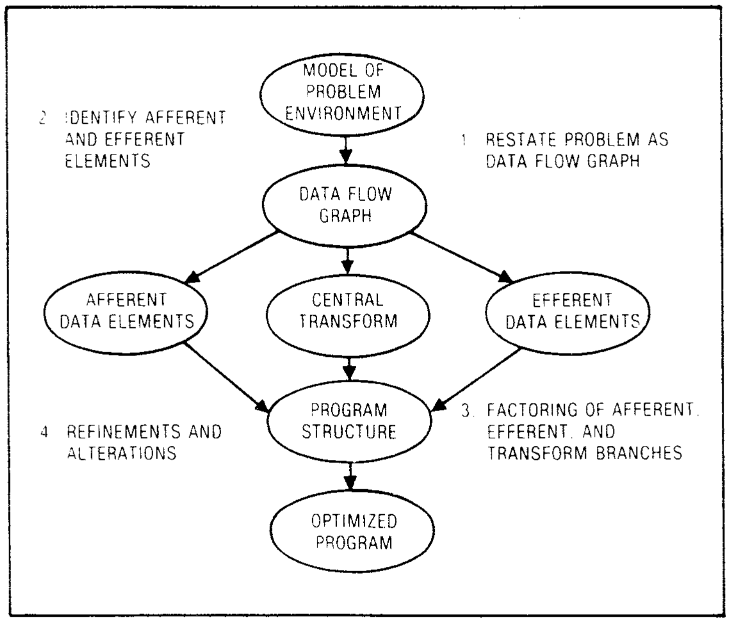

MODEL OF PROBLEM ENVIRONMENT
1 RESTATE PROBLEM AS DATA FLOW GRAPH
2 IDENTIFY AFFERENT AND EFFERENT ELEMENTS
DATA FLOW GRAPH
AFFERENT DATA ELEMENTS
CENTRAL TRANSFORM
EFFERENT DATA ELEMENTS
PROGRAM STRUCTURE
3 FACTORING OF AFFERENT, EFFERENT, AND TRANSFORM BRANCHES
4 REFINEMENTS AND ALTERATIONS
OPTIMIZED PROGRAM

Figure 22. Data flow design procedure.

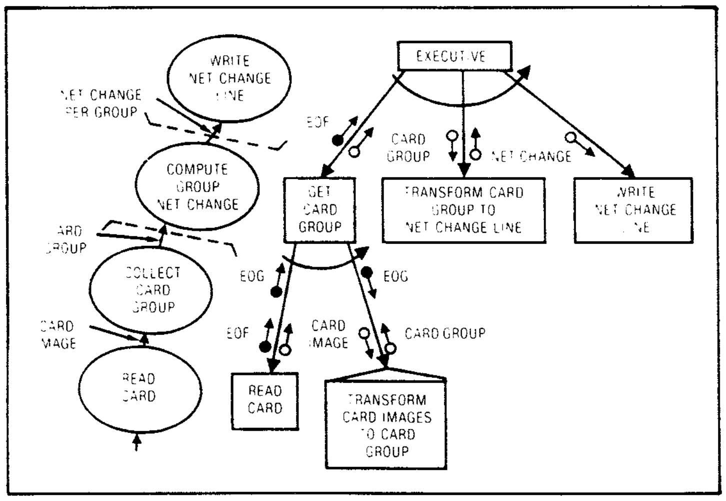


Figure 23. Data flow design structure.

McDonald's example. The heroine who designed the next program was Ivan's sister, Ivy. Ivy is a child of the late 1960's. When modular programming came in, she jumped right on the bandwagon. Her modules had only one entrance and one exit. She passed all her parameters in each call statement. Each of her modules performed only a single logical task, was independent, and could be separately tested. She read daily from the gospel according to Harlan Mills[12] and remembered nearly every error that she had ever made. She well remembers the day she designed this program.

Model problem. The data flow graph for the McDonald's example is shown on the left side of Figure 23. The data items being passed between bubbles are labeled, and the modules are named with an action verb and an object. Note that one or more *card images* "become" a *card group* after being processed by the COLLECT CARD GROUP function.

Afferent, efferent, and central transform elements. The most abstract input in the data flow graph is a *card group.* The most abstract output is the *net change* resulting from processing each card group. Thus, the COMPUTE GROUP NET CHANGE module is the central transform, the WRITE NET CHANGE LINE module is the efferent (or output) element, and the READ CARD and COLLECT CARD GROUP modules are the afferent (or input) elements.

Factor branches. Note that the concept of a card group emerges naturally out of the data flow diagram and leads to a reasonably straightforward program structure (see Figure 23). The TRANSFORM CARD IMAGES TO CARD GROUP module is shown with a roof that denotes lexical inclusion. That is, while TRANSFORM CARD IMAGES TO CARD GROUP is a functional module, it is not necessarily a physical module. In this example, it will be packaged together with the GET CARD GROUP module.

The curved arrows in Figure 23 denote iteration. That is, the GET CARD GROUP module calls its two subtending modules once per card image. The EXECUTIVE module calls its three subtending modules once per card group. Note that the READ CARD module is assumed to pass an "end of file" flag up the chain when it is detected. (The passing of control information is denoted by the small arrows with solid tails, while the passing of data is shown by the small arrows with open tails.) Ivy's program listing is shown in Figure 24.

Only three of the six modules are shown. The TRANSFORM CARD IMAGES TO CARD GROUP module was lexically included in the GET CARD GROUP module. The other two modules are not necessary for this particular discussion. Note that the READ CARD module leaves much to be desired.

Program testing. Ivy has learned to make good use of a program librarian. She wrote the program, gave the coding sheets to the librarian, and went back to her reading. The program librarian faithfully executed his duties and brought back a printout that said "CRITEM undefined."

Ivy had forgotten to make sure that all of her variables were initialized. Oh well, this should be easy. How about setting CRITEM: = XXXX at the beginning of the GET CARD GROUP module? That would be before CRITEM was used the first time, and when it has a symmetry with the way EOG (end of group) is initialized.

Unfortunately, something just didn't seem right about it. GET CARD GROUP is executed many times during the program, and CRITEM only needs to be initialized once.

Ivy decided to initialize CRITEM at the beginning, instead. While that means passing it as a parameter up two levels, it certainly sounded better than a bunch of first-time switches.

On the next run, Ivy's efforts were rewarded with some output—nothing fancy, but still some output.

"What happened to the heading?" exclaimed Ivy. "That data link must be dropping bits again."

"Where did you write out the heading?" asked the program librarian. Enter quick fix number 2, shown in Figure 25.

"Wait, what about that garbage in the front?"

The problem, of course, is that the program has no way of knowing when it's through with a group until it's already started processing the next group. Thus, the first card of a new group serves as a key to tell the rest of the program to send on the previous group. It can't do this, however, without distorting the structure.

"I think it's time to call out my secret weapon," said Ivy, "my UNREAD command." Enter quick fix number 3, shown in Figure 26.

Field debugging. This fix apparently worked fine for about two months. It was not exactly speedy, but it did work. Then, all of a sudden, a visit from on high. Big Mac himself came down and said, "Our people in Provo, Utah, have been trying to bring up your program, and it just doesn't work."

It turns out that in Provo they never found it necessary to buy a tape reader. Ivy's UNREAD operation didn't work on cards. That meant that Ivy had to find another way of UNREADing.

In this data flow design, the equivalent of an UNREAD is, at best, messy. It corrupts the structure badly, no matter how it's done. Modules end up storing internal states or values, and first-time switches abound.

In Ivy's case, she chose to read ahead by one, passing state information by SW1 and storing the NEW CARD value within module READ CARD. Other solutions are possible, of course, but it isn't clear that they are a whole lot better (see Figure 27). The effect of these changes on the program structure is shown dramatically in Figure 28.

Ivy has committed arboricide, to use Jackson's words. What was a nice clean tree structure now has two programs calling the same READ CARD module. I think

```
EXECUTIVE: EOF: =FALSE:

   DO WHILE(EOF=FALSE):
      GET__CARD__GROUP(CG.EOF                    ):
      TRANSFORM__CG__TO__NC__LINE(CG.NC):
      WRITE__NC__LINE (NC.EOF):
   END:                       STOP:

GET__CARD__GROUP(CG.EOF                   ): EOG: =FALSE: I: =0:
   DO WHILE (EOG=FALSE): I: =I+1:
      READ__CARD(CI.EOG.EOF                ):CG(I) = CI:
   END:                RETURN:

READ__CARD(CI.EOG.EOF                 ):
   NEW__CARD: =READ STF:
      IF NEW__CARD__ITEM≠CRITEM THEN
         DO: EOG. =TRUE: CRITEM=NEW__CARD__ITEM:
                       END:
      ELSE    CI: =NEW CARD:
      IF CI=EOF__STF THEN EOF: =TRUE: RETURN:
```

Figure 24. Data flow design program.

```
EXECUTIVE:  EOF: =FALSE:
   CRITEM: =XXXX:
   DO WHILE(EOF=FALSE):
      GET__CARD__GROUP(CG.EOF.CRITEM      ):
      TRANSFORM__CG__TO__NC__LINE(CG.NC):
      WRITE__NC__LINE (NC.EOF):
   END:                       STOP:

GET__CARD__GROUP(CG.EOF.CRITEM         ): EOG: =FALSE: I: =0:
   DO WHILE (EOG=FALSE): I: =I+1:
      READ__CARD (CI.EOG.EOF.CRITEM        ):CG(I)=CI:
   END:                RETURN:
READ__CARD(CI.EOG.EOF.CRITEM           ):
   NEW__CARD: =READ STF:
      IF NEW__CARD__ITEM ≠CRITEM THEN
         DO: EOG: =TRUE: CRITEM: =NEW__CARD__ITEM:
                    END:
      ELSE   CI: =NEW__CARD:
      IF CI=EOF__STF THEN EOF =TRUE: RETURN:
```

Figure 25. Quick fix no. 1.

```
EXECUTIVE:   EOF: =FALSE: WRITE HEADING:
   CRITEM: =XXXX:
   DO WHILE(EOF=FALSE):
      GET__CARD__GROUP(CG.EOF.CRITEM      ):
      TRANSFORM__CG__TO__NC__LINE(CG.NC):
      WRITE__NC__LINE (NC.EOF):
   END:                      STOP:

GET__CARD__GROUP(CG.EOF.CRITEM         ): EOG =FALSE: I =0
   DO WHILE (EOG=FALSE): I =I+1
      READ__CARD (CI.EOG.EOF CRITEM       ): CG(I)=CI
   END:                RETURN:

READ__CARD(CI.EOG.EOF.CRITEM           ):
   NEW__CARD =READ STF:
      IF NEW__CARD__ITEM ≠CRITEM THEN
         DO: EOG =TRUE: CRITEM =NEW  CARD  ITEM:
         UNREAD STF END:
      ELSE   CI: =NEW__CARD:
      IF CI=EOF  STF THEN EOF =TRUE: RETURN
```

Figure 26. Quick fixes no. 2 and no. 3.

```
EXECUTIVE:   EOF: = FALSE;WRITE HEADING:  SW1 = FALSE.
   CRITEM: = XXXX:       READ__CARD(CI.EOG.EOF.CRITEM.SW1):
   DO WHILE(EOF = FALSE):
      GET__CARD__GROUP(CG.EOF.CRITEM.SW1):
      TRANSFORM__CG__TO__NC__LINE(CG.NC):
      WRITE__NC__LINE(NC.EOF):
   END:                            STOP:

GET__CARD__GROUP(CG.EOF CRITEM.SW1): EOG: = FALSE: I: = 0:
      DO WHILE (EOG = FALSE). I: = I + 1:
      READ__CARD (CI.EOG.EOF.CRITEM.SW1):CG(I) = CI:
   END: SW1 = TRUE: RETURN:

READ__CARD(CI.EOG.EOF.CRITEM.SW1):IF SW1 = FALSE THEN
   NEW__CARD: = READ STF:
      IF NEW__CARD__ITEM ≠ CRITEM THEN
         DO: EOG: = TRUE: CRITEM = NEW__CARD__ITEM:
        SW1: = TRUE:                      END:
      ELSE DO: CI: = NEW__CARD:SW1: = FALSE: END:
      IF CI = EOF__STF THEN EOF: = TRUE: RETURN:
```

Figure 27. Quick fix no. 4.

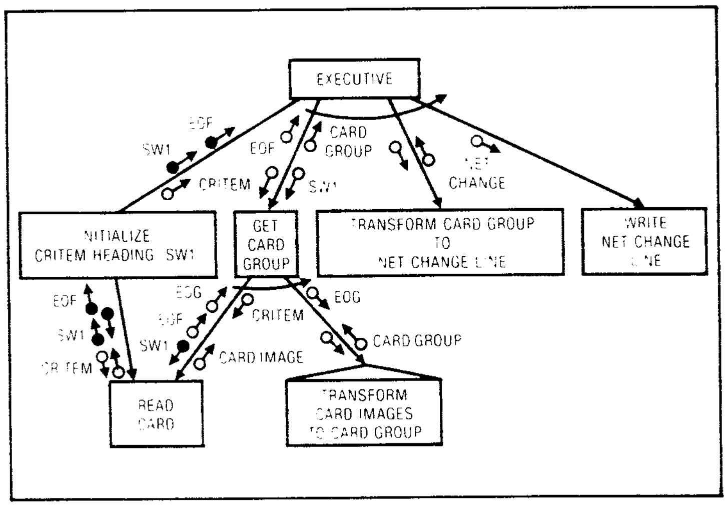


Figure 28. Data flow design with fixes.

```
EXECUTIVE: EOF: = FALSE: WRITE HEADING: SW1: = FALSE:
      CRITEM: = XXXX: CI: = READ STF:
      DO WHILE (CI NOT EOF - STF):
         EOG: = FALSE I: = 0:
         DO WHILE (EOG = FALSE): I: = I + 1:
             IF SW1 = FALSE THEN NEW__CARD: = READ STF:
             IF NEW__CARD__ITEM ≠ CRITEM THEN
                DO: EOG: = TRUE: CRITEM: = NEW__CARD__ITEM:
                SW1: = TRUE:
                END:
             ELSE DO: CI: = NEW__CARD SW1: = FALSE:
                END:
             CG(I): = CI:
         END:
         SW1: = TRUE:
         TRANSFORM__CG__TO__NC__LINE (CG.NC):
         WRITE__NC__LINE (NC.EOF):
      END:
      STOP:
```

Figure 29. Data flow design packaged in one module.

read and write operations should be thought of as general-purpose routines callable from anywhere within the program structure. To hope to constrain them to the "ears" of a structure chart seems, at best, unwise.

Optimization. In this program, things look much better if we simply package the whole thing as one module, as shown in Figure 29.

The point, however, is that the problems Ivy had were representative of larger problems that could appear in larger programs each time the data flow design method is applied.

Data structure design

Slightly different forms of the data structure design method were developed concurrently by Michael Jackson[7] in England and J. D. Warnier[23] in France. In this discussion, Jackson's formulation and notation are used.

The basic premise is that a program views the world through its data structures and that, therefore, a correct model of the data structures can be transformed into a program that incorporates a correct model of the world. The importance of this view, stated earlier as the principle of correspondence, is emphasized by Michael Jackson's words that "a program that doesn't directly correspond to the problem environment is not poor, is not bad, but is wrong!"

When the program structure is derived from the data structure, the relationship between different levels of each resulting hierarchy tends to be a "is composed of" relationship. For example, an output report is composed of a header followed by a report body followed by a report summary. This is generally a static relationship that does not change during the execution of the program, thus forming a firm base for modeling the problem.

Since a data-structure specification usually lends itself well to being viewed as correct or incorrect, the program structure based on a data-structure specification can often be viewed as being correct or incorrect. Jackson[7] purports that two people solving the same problem should come up with program structures that are essentially the same. Thus, this method satisfies the principle of consistency to a large degree.

Design strategy. The programming process can be partitioned into the following steps:

1. (1) Form a system network diagram that models the problem environment.
2. (2) Define and verify the data-stream structures.
3. (3) Derive and verify the program structures.
4. (4) Derive and allocate the elementary operations.
5. (5) Write the structure text and program text.

These steps can usually be performed and verified independently, separating concerns by partitioning both the design process and the problem solution. For large problems, the objective is a network of hierarchies, each representing a simple program. Jackson's premise is that although simple programs are difficult to write, complex programs are impossible. The trick, then, is to partition complex problems into simple programs.

These simple programs, individually implemented as true hierarchical modular structures, are connected in a data-flow network. This network can be placed into a "calls" or "is called by" hierarchy by a scheduling procedure called "program inversion"[7] that is performed as a separate step after the structure of the program has been defined.

The system network diagram for a simple program is represented schematically in the upper left corner of Figure 30 and is explained further in the example. In its simplest form, it represents a network of functions that consume, transform, and produce sequential files. Below this network diagram, the data structure of each file is shown to be represented by data structure diagrams. The notation used is similar to that shown in Figure 4.

If the data structures correspond well, a program structure diagram can be drawn that encompasses both data structures. When a diagram cannot encompass both data structures, a *structure clash*[7] exists.

Since the program structure models the data structures, and since most operations are performed on data elements, one can list and allocate executable operations to each component of the program structure. These elementary operations are denoted by the small squares in the structure diagram and are shown in a list in the lower left corner of Figure 30.

The basic data structure design procedure can also be represented schematically, as shown in Figure 31. The arrows represent the flow of work and the results required in following the basic design procedure. Note that the final program structure is formed by first finding data structure correspondences and then by adding in the executable operations also derived from the data structures. The problem model is usually documented by a system network diagram, the data structures and program structure by structure diagrams, and the program text by structure text. Examples of each of these are given below in the McDonald's example.

The major problem with Jackson's data structure design methodology is that, in Jackson's words, "it is being developed from the bottom up." That is, although it is clear at this point how to apply it to small problems, the "correct" method for extending it to large system problems is still being developed.

McDonald's example. Ida is a data structure designer from way back. She went to London to study the Jackson design method. She learned French just so she could read Warnier's six paperbacks. (She says they lose a lot in translation.) With the McDonald's problem, she is certain that a data structure design is the only way to go—after all, it fits into Jackson's "stores movement" problem solution format.[7]

Model step. The system network diagram for the McDonald problem was given at the bottom of Figure 11.

Data step. The second design step was to construct and verify the input and output data structures for each file shown in the system network diagram. The three basic program constructs of sequence, iteration, and selection—shown in Figure 4—apply equally well to describing the structure of a data file (see Figure 32). Note that the SORTED TRANSACTION FILE is an iteration of ITEM GROUP, which is an iteration of TRANSACTION RECORD, which—in turn—is a selection of a RECEIVED RECORD or a DISTRIBUTED RECORD. The output report is a sequence of the REPORT HEADING followed by the REPORT BODY followed by the REPORT SUMMARY. The REPORT BODY is an iteration of REPORT LINE. (Note that REPORT BODY is an iteration of REPORT LINE—not REPORT LINES. Plural names in data component boxes often indicate an error in naming.)

After the input and output data structures were diagrammed, one-to-one correspondences were shown by arrows. For example, one SORTED TRANSACTION FILE is consumed in producing one REPORT, and one ITEM GROUP is consumed in producing one REPORT LINE.

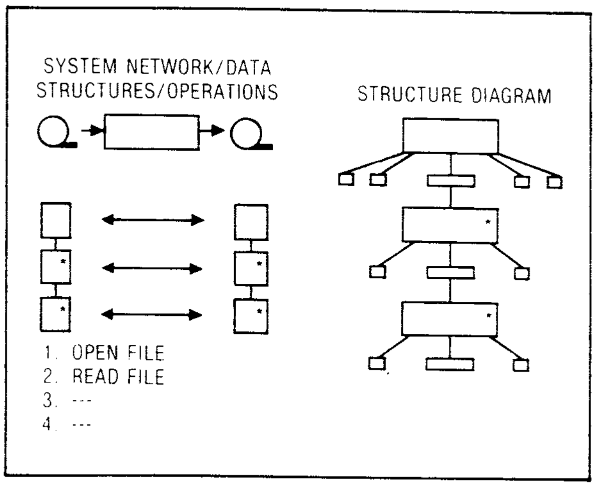


Figure 30. Data structure design method.

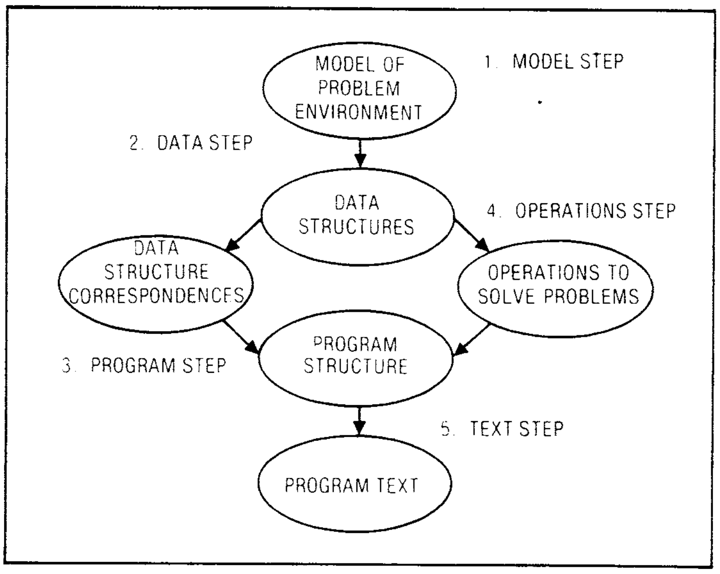


Figure 31. Basic data structure design procedure.

Program step. From these data structures, a program structure was constructed encompassing all of the parts in each data structure. Where there were one-to-one correspondences, the modules took the form of CONSUME. . . TO PRODUCE. . . .Where there were modules corresponding to only the input data structure, the form was CONSUME. . . .Where there were modules corresponding to only the output data structure, the form was PRODUCE. . . .All of these are shown in Figure 32.

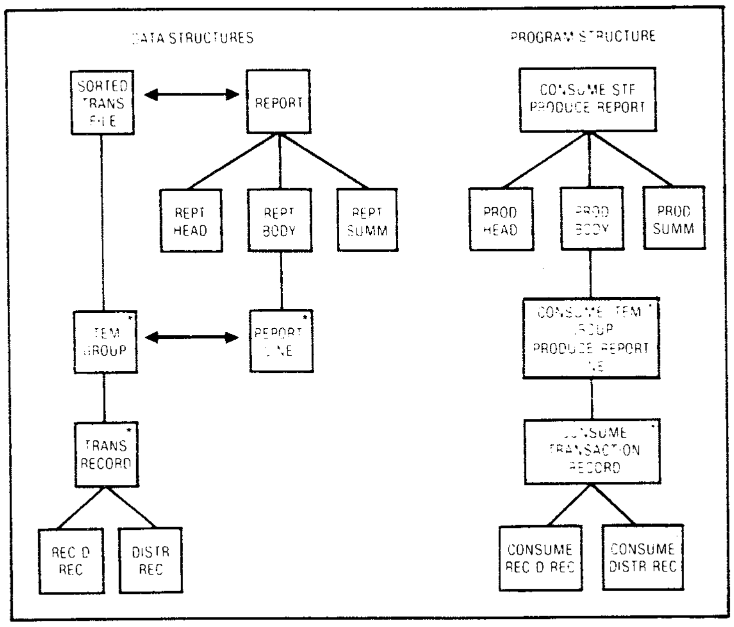


Figure 32. Data and program structures.

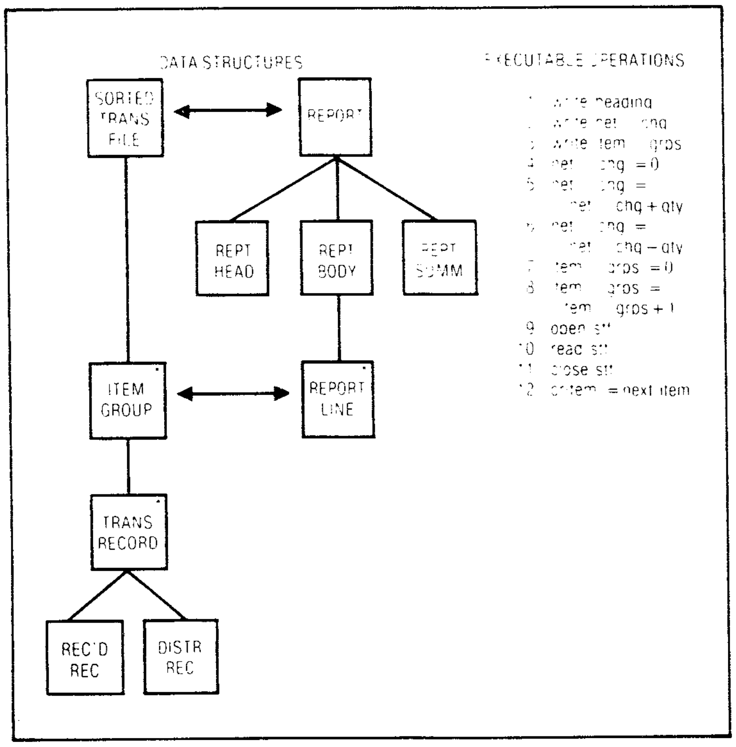


Figure 33. Listing executable operations.

In finding the producer-consumer correspondences, the question is always, "Does one instance of this result in one instance of that?" This question should be asked again during the verification procedure, which consists of showing that both the input and output data structures are subtrees of the program structure.

Operations step. The input, output, and computational operations are identified by working back from the output data structure to the input data structure (see Figure 33). If necessary, the equivalent of a data flow diagram can be drawn with nodes to represent intermediate variables.

The operations must be sufficiently rich to guarantee that each output can be produced and each input can be consumed. Computational operations must also be specified that will implement the algorithms which link the two. One other type of operation is usually required for completeness. In our example, this is shown as operation 12. This operation provides the key that will be used in determining item group boundaries, and it is usually called a "structure-derived operation."

While it is desirable to have a complete list of operations from the start, it is not required, since missing operations will become apparent later in the process.

The last part of the operations step involves allocating all of these executable operations to the program structure (see Figure 34). In each case, the questions to ask are "How often should this operation be executed?" and "Should this operation occur before or after. . .?" For this example, the answer to the first question could be once per report, once per sorted transaction file, once per heading, once per summary, once per item group, once per report line, once per transaction record, once per item received record, or once per item distributed record. Clearly, you open the sorted transaction file, or STF, before you read it, you close the STF before you stop, etc. Using the read-ahead rule,[7] you read one card ahead and then read to replace (see operation 10).

At the end of this step, one should verify that all outputs are produced, all intermediate results are produced, and all inputs are consumed.

Text step. Although the structure of the program is now secure, Jackson recommends one final step before coding. That fifth step is to translate the structure diagram into structure text, as shown in Figure 35. This is done using the three basic program constructs shown in Figure 4.

Structure text is straightforward and easy to understand as long as your program labels are short and descriptive. With long program labels, it can become a mess. In Figure 35, the letters C and P are sometimes used as abbreviations for CONSUME and PRODUCE.

Ida's final program was produced by translating the structure text of Figure 35 into the target programming language of Figure 36. The major disadvantage of this method is that the number of distinct steps and procedures involved imply that both the data structure diagram and the structure text should be kept as permanent parts of the documentation. This seems unlikely unless an automated structure chart drawer is available that will store the structure chart with the program in a convenient form so that it can be updated when the program is changed. At this point, available machine aids are far from adequate. Another disadvantage is that for certain classes of problems, the data structure diagrams and the resulting program structures can get unduly cumbersome.

A programming calculus

While a "proof of correctness" is disappointingly difficult to develop after a program has been written, the constructive proof-of-correctness discipline taught by Dijkstra[20] and Gries[24] is relatively encouraging. Dijkstra's design discipline can be methodically applied to obtain a modest-sized "elegant" program with a "deep logical beauty." Using this method, the program and the proof are constructed hand-in-hand.

Design strategy. The initial design task consists of formally specifying the required result as an assertion stated in the predicate calculus. Given this desired postcondition, one must derive and verify the appropriate preconditions while working back through the program being constructed. The program and even individual statements play a dual role in that they must be viewed in both an operational way and as predicate transformers. The method is a top-down method to the extent that both the resulting program and the predicates can be formed in stages by a sequence of stepwise refinements.

Definitions. The programming language is used as a set of statements that perform predicate transformations of the form[20,24,25]

$$\{Q\}S\{R\} \tag{1}$$

In this expression, Q represents a precondition that is true before execution of statement S. R represents a postcondition that is true after the execution of statement S. In simple terms, the formula means that the execution of statement S beginning in a state satisfying Q will terminate in a state satisfying R, but Q need not describe the largest set of such states. When Q does describe the largest set of such states, it is called the weakest (least restrictive) precondition.

When Q is the weakest precondition, this is expressed in the form

$$Q = wp(S, R) \tag{2}$$

When S is an *assignment* statement, it takes the form

$$x := e \tag{3}$$

where e is an expression. Its definition as a predicate transformer is given by

$$wp(\text{"}x := e\text{"}, R) \equiv R_e^x \tag{4}$$

The term R_e^x is used to denote the textual substitution of expression e for each free occurrence of x within expression R.

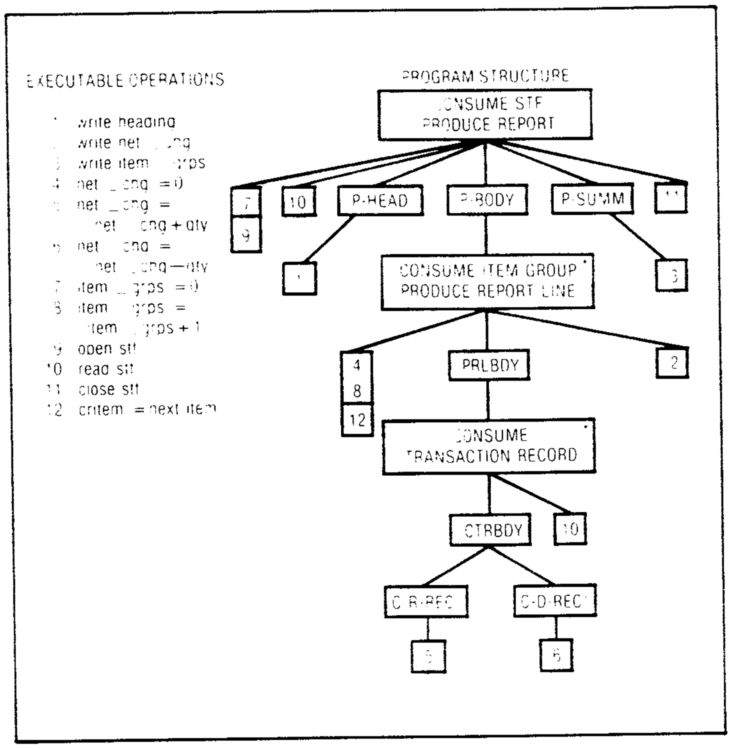


Figure 34. Allocating executable operations.

When s is a *Selection* statement, it takes the form

$$\begin{aligned} \text{IF} \equiv \textbf{if} \quad & B_1 \rightarrow S_1 \\ [] \quad & B_2 \rightarrow S_2 \\ & \ldots \\ [] \quad & B_n \rightarrow S_n \\ \textbf{fi} \quad & \end{aligned} \tag{5}$$

```
P seq
 item__grps: = 0;
 open stf; read stf;
 write heading;
 P-BODY itr until (eof-stf)
   C-ITEM-GRP-P-REPT-LINE seq
     net__chg: = 0;
     item__grps: = item__grps + 1;
     critem: = next item;
     PRLBDY itr while (next item = critem)
       C-TRANS-REC seq
         CTRBDY sel (code - R)
           net__chg: = net__cng + qty;
         CTRBDY alt (code - D)
           net__chg: = net__chg - qty;
         CTRBDY end
         read stf;
       C-TRANS-REC end
     PRLBDY end
     write net__chg;
   C-ITEM-GRP-P-REPT-LINE end
 P-BODY end
 write item__grps; close stf;
P end
```

Figure 35. Structure text.

This is valid where $n > 0$, the B_i are boolean expressions called guards, and the S_i are statements. The vertical bar "$[]$" serves to separate otherwise unordered alternatives. To execute IF, at least one of the guards B_i must be true. To execute it, choose one guard that is true (nondeterministically) and execute the corresponding statement. The expression $B_1 \rightarrow S_1$ may be read as "When guard B_1 is true, statement S_1 may be executed." The definition of IF as a predicate transformer is given by

$$wp(\text{IF}, R) \equiv \text{BB and } (\mathbf{A}\, i : 1 \le i \le n : B_i \Rightarrow wp(S_i, R)) \quad \textbf{(6)}$$

where $\text{BB} \equiv B_1$ or B_2 or . . . or B_n. Thus, we must prove that executing any S_i when B_i and Q are true establishes R. In practice, given R and S_i, we derive B_i using the weakest precondition.

The *iteration* statement takes the form

$$\begin{array}{lll} \text{DO} & \equiv & \textbf{do}\ B_1 \rightarrow S_1 \\ & & []\ \ B_2 \rightarrow S_2 \\ & & \quad . . . \\ & & []\ \ B_n \rightarrow S_n \\ & & \textbf{od} \end{array} \quad \textbf{(7)}$$

Iteration continues as long as at least one of the guards $B_1, . . ., B_n$ is true. From the set of statements with true guards, one is selected (nondeterministically) for execution. The definition of DO as a predicate transformer is given by

$$wp(\text{DO}, R) \equiv (\text{E}\, k : k \ge 0 : \text{H}_k(R)) \quad \textbf{(8)}$$

where $\text{H}_k(R)$ for $k \ge 0$ is the weakest precondition such that execution of DO will terminate after, at most, k "iterations" and in a state satisfying R. In developing a program that contains a loop structure, this is found by completing the following steps:

1. Write a formal specification of the desired result R.
2. Determine an invariant P.
3. Derive a loop body such that:
   a. $Q \Rightarrow P$ (i.e., P is initially true).
   b. $\mathbf{A}\, i : \{P \text{ and } B_i\}\ S_i\ \{P\}$
      (i.e., P remains invariantly true).
   c. (P and not BB) $\Rightarrow R$
      (i.e., upon termination R is true).
   d. Show the loop terminates:
      (1) (P and BB) $\Rightarrow t > 0$
          (i.e., before termination, t is positive).
      (2) $\mathbf{A}\, i : \{P \text{ and } B_i\}\ \tau := t;\ S_i\ \{t \le \tau - 1\}$
          (i.e., t decreases with each iteration).

In step 3d, t is a termination function chosen to be positive during the execution of the iteration and zero upon completion; τ is a fresh integer variable.

```
PB:      item__grps: = 0:
         open stf:
         read stf:
         write heading:
PBB:     do while (not eof—stf):
          net__chg: = 0:
          item__grps: = item__grps + 1:
          critem: = next item:
PRLBB:    do while (next item—critem):
CTRBB:     if (code = R) then
             net__chg: = net__chg + qty:
           else if (code = D) then
CTRBE:       net__chg: = net__chg − qty:
            read stf:
PRLBE:    end:
          write net__chg:
PBE:     end:
         write summary:
PE:      close stf:
```

Figure 36. Final data structure design program.

McDonald's example. The problem is initially simplified by eliminating the R/D field of each transaction and considering the quantity field $q(j)$ as positive for items received and negative for items dispersed. The R/D field is reintroduced later when the final pseudocode is written so that the resulting program can be readily compared with previous designs.

Problem specification. Note that the STF is an iteration of item group, which is an iteration of transaction, which—in turn—is a sequence of the item field $f(j)$ and the quantity field $q(j)$ for $1 \le j < m$ and $f(m) = EOF$.

Since the transaction file has been sorted, an item group $g(j,k)$ can be defined as

$$g(j,k) \equiv f(j) \ne f(j-1) \text{ and } f(j) = f(k) \text{ and } f(k) \ne f(k+1) \quad \textbf{(9)}$$

Similarly, a partial item group $\hat{g}(j,k)$ can be defined as

$$\hat{g}(j,k) \equiv f(j) \ne f(j-1) \text{ and } f(j) = f(k) \text{ and } f(k) = f(k+1) \quad \textbf{(10)}$$

The number of complete groups over the range 1 through k can be defined as

$$n(1,k) \equiv (\mathbf{N}\, p : 1 \le p \le k : f(p) \ne f(p+1)) \quad \textbf{(11)}$$

where $\mathbf{N}\, p$ is the number of distinct values of p over the range $1 \le p \le k$ for which $f(p) \ne f(p+1)$.

Formal specification of R. Step 1 in the design procedure is to develop, in a top-down fashion, a formal specification of the desired result R. A first attempt could look something like this:

R : The total number of item groups i is calculated. The net change in inventory c is calculated for each item group.

More formally:

$$R : (\mathbf{A}\, j,k : 1 \le j \le k < m \text{ and } g(j,k) : i = n(1,k) \text{ and } c(i) = \sum_{r=j}^{k} q(r)) \quad \textbf{(12)}$$

This whole expression may be read "If R is true, the following holds: For all j and k such that $1 \leq j \leq k < m$ and the transactions between j and k inclusive form a group, it is true that i is set equal to the number of complete groups over the interval 1 through k and $c(i)$ is formed as the sum of the quantity fields within each complete group." This notation is consistent with that used by Dijkstra and Gries.[25] Note that the result assertion specifies *what* is to be derived—not *how* it is to be derived.

When we examine (12), the requirement for counting the number of complete groups and the summation required for computing $c(i)$ suggest that at least one loop structure is required in the program. Thus, our next step is to determine an invariant P.

Determine an invariant P. When step 2 is performed, the result assertion (12) is weakened to form an invariant P. In this example, R may be weakened by introducing a variable $\hat{m}$, where $1 \leq \hat{m} \leq m$, such that

$$\begin{aligned} P&: (1 \leq \hat{m} \leq m) \\ &\text{and } (\mathbf{A}\, j,k : 1 \leq j \leq k < \hat{m} \text{ and } g(j,k) \\ &\qquad : i = n(1,k) \text{ and } c(i) = \sum_{r=j}^{k} q(r)) \\ &\text{and } (\mathbf{A}\, j,k : 1 \leq j \leq k < \hat{m} \text{ and } \hat{g}(j,k) \text{ and } k = \hat{m} - 1 \\ &\qquad : \hat{c} = \sum_{r=j}^{k} q(r)) \end{aligned} \qquad \textbf{(13)}$$

This invariant states that "If P is true, the following holds: For all the complete item groups, i is set to the group number and $c(i)$ is set to the sum of the q fields within each group. For the last partial item group, $\hat{c}$ is set to the sum of the q fields."

Derive the program. Step 3d(1) requires that a termination function t be chosen that is greater than zero while any of the guards B_i are still true and that decreases with every execution of S_i. For our example, a termination function that can readily be made to satisfy these conditions is

$$t = m - \hat{m} \qquad \textbf{(14)}$$

This function will equal zero on termination if the guard B_1 is set to terminate the iteration when $\hat{m} = m$.

With this in mind, the basic program form becomes

```
"define Q to establish P";
{P}
do $\hat{m} \neq m \rightarrow$
    "decrease t while maintaining P"          (15)
od
{R}
```

A statement that would decrement t is

$$\hat{m} := \hat{m} + 1 \qquad \textbf{(16)}$$

As noted in step 3a, we must next show that P is initially true. In other words, the initial state $\{Q\}$ must be so defined that Q implies P. That is

$$Q \Rightarrow P \qquad \textbf{(17)}$$

Note that if the program starts with the statement

$$\hat{m}, i, \hat{c} := 1,0,0 \qquad \textbf{(18)}$$

then P holds.

To satisfy step 3b, we must show that P remains true after statement (16) is executed. One way of meeting this requirement is to introduce further guards B_i by embedding an IF selection statement within the loop construct. The IF statement format is shown in (5). The program is now in the form

```
$\hat{m}, i, \hat{c}$ := 1,0,0;
{P}
do $\hat{m} \neq m \rightarrow$
    if
        "statements that reestablish P"       (19)
    fi;
    $\hat{m}$ := $\hat{m}$ + 1
    {P}
od
{R}
```

The guards of the IF statement must be chosen such that P remains true after statement (16) is executed. In general, guards B_i must be chosen so that

$$P \text{ and } B_i \Rightarrow wp(S_i, P) \qquad \textbf{(20)}$$

In our example, we must find

$$\begin{aligned} &wp(\text{"}\hat{m} := \hat{m} + 1\text{"}, P) \\ &\quad = (1 \leq \hat{m} + 1 \leq m) \\ &\qquad \text{and } (\mathbf{A}\, j,k : 1 \leq j < k < \hat{m} + 1 \text{ and } g(j,k) \\ &\qquad\qquad : i = n(1,k) \text{ and } c(i) = \sum_{r=j}^{k} q(r)) \\ &\qquad \text{and } (\mathbf{A}\, j,k : 1 \leq j \leq k < \hat{m} + 1 \text{ and } \hat{g}(j,k) \\ &\qquad\qquad \text{and } k = \hat{m} \\ &\qquad\qquad : \hat{c} = \sum_{r=j}^{k} q(r)) \end{aligned} \qquad \textbf{(21)}$$

Consequently, we must choose guards for the IF that ensure that all three of the terms of (21) are satisfied when (16) is executed. The first term

$$1 \leq \hat{m} + 1 \leq m \qquad \textbf{(22)}$$

can be established directly from P and the iteration guard ($\hat{m} \neq m$). The second term of (21)

$$\mathbf{A}\, j,k : 1 \leq j \leq k < \hat{m} + 1 \text{ and } g(j,k) \qquad \textbf{(23)}$$

$$: i = n(1,k) \text{ and } c(i) = \sum_{r=j}^{k} q(r)$$

can be established if P is true and if

$$f(\hat{m}) \neq f(\hat{m}+1) \rightarrow i := i+1; c(i) := = \hat{c} = q(\hat{m}); = \hat{c} := 0 \quad \textbf{(24)}$$

where initially

$$\hat{c} = \sum_{r=j}^{\hat{m}-1} q(r) \quad \textbf{(25)}$$

It can also be shown that the third term of (21)

$$\cdot \ \mathrm{A}\, j,k : 1 \leq j \leq k < \hat{m}+1 \text{ and } \hat{g}(j,k) \text{ and } k = \hat{m} \quad \textbf{(26)}$$

$$: \hat{c} = \sum_{r=j}^{k} q(r)$$

is true if P is true and if

$$f(\hat{m}) = f(\hat{m}+1) \rightarrow \hat{c} := \hat{c} + q(\hat{m}) \quad \textbf{(27)}$$

With these observations, our final program becomes

```
$\hat{m}, i, \hat{c} := 1,0,0;$
$\{P\}$
do $\hat{m} \neq m \rightarrow$
    if $f(\hat{m}) = f(\hat{m}+1) \rightarrow \hat{c} := \hat{c} + q(\hat{m})$                    (28)
    $▯ f(\hat{m}) \neq f(\hat{m}+1) \rightarrow i := i+1; c(i) := \hat{c} + q(\hat{m}); \hat{c} := 0$
    fi;
    $\hat{m} := \hat{m}+1 \{P\}$
od
$\{R\}$
```

Show the loop terminates. During the design procedure, we have verified the truth of the first three conditions stated earlier in step 3 regarding the invariant P. Step 3d can be satisfied by showing that t remains positive while at least one guard is true and that each iteration decreases t. These final termination conditions should be verified by the reader.

Pseudocode. A pseudocode version of the final program is shown in Figure 37. Note that the meanings of code R, code D, and EOF have been restored. In this figure, i becomes *item_grps,* $\hat{c}$ becomes *net_chg,* $\hat{m}$ corresponds to *curr_rec,* and $f(\hat{m})$ corresponds to *curr_item.*

Conclusion

Comparison. While there is something to be learned from examining the pseudocode resulting from the four solutions to the McDonald's problem, more significant points can be made by comparing the differences in program structure.

Functional decomposition solution. A skeleton structure diagram for the functional decomposition solution is shown in Figure 38. Note that the basic operation was assumed to be PROCESS CARD and that this design treats the first card as different from the subsequent cards quite early. This results in a structure that favors the addition of changes that can be keyed to the beginning of a group. Changes that must be keyed to the end of a group are added with more difficulty.

Data flow design solution. The data flow design solution had the basic structure shown in Figure 39. Note that in this case there is a difference between a lead card and the last card in a group. That is the last card in a group gets treated in a special way—not the first card. For later modification, this means that operations added to the end of a group will be favored while operations added to the beginning of a group will not fit in as naturally.

Even through the concept of group is well localized on the diagram, the data flow nature of the processing forces one to do some of the group operations at the PROCESS CARD level. In this case, an end-of-group control signal must be inserted at the PROCESS CARD level so that other end-of-group processing functions can be activated within the PROCESS GROUP function.

The resulting program has sequential cohesion. Both data and control are passed up the structure diagram. Generating the end-of-group indication still causes a few minor problems, but in this particular example keying on the last card in a group turns out to be a better choice than keying on the first card in a group.

Data structure design solution. The data structure design solution is shown in Figure 40. In this solution, all of the cards are processed by the same PROCESS CARD function. The keying of groups is done automatically

```
item__grps: = 0;
net__chg: = 0;
open stf;
curr__rec: = read stf;
write heading;
do curr__rec ≠ EOF
  next__rec: = read stf;
  if curr__item = next__item →
    if code = R → net__chg: = net__chg + qty;
    ▯ code = D → net__chg: = net__chg − qty;
    fi
  ▯ curr__item ≠ next__item →
    item__grps: = item__grps + 1;
    if code = R → net__chg: = net__chg + qty;
    ▯ code = D → net__chg: = net__chg − qty;
    fi
    write net__chg;
    net__chg: = 0;
  fi
  curr__rec: = next__rec;
od
write item__grps;
close stf;
```

Figure 37. Programming calculus McDonald's solution.

within the control structure through use of the read-ahead rule. Thus, no cards are treated with special functions, and everything looks clean. This solution can handle further additions that require special action at the beginning or at the end of a group with relative ease.

Programming calculus solution. The programming calculus solution structure diagram is shown in Figure 41. Note that this solution treats the last card as special, favoring extensions that occur at the end of a group. Since most of the McDonald's problem group functions are end-of-group functions, the resulting solution looks quite clean. In the long run, this solution could cause problems when new beginning-of-group operations are added.

Of these four solutions, the one that seems to model the problem best is the data structure design solution shown in Figure 40. A proper choice of loop invariants using the predicate calculus may have led to the same structure, but the choice that was made didn't. Likewise, a proper application of functional decomposition also could have led to the same structure. Even the data flow design method does not completely rule out this preferred solution.

It might be argued that all four design methodologies were capable of yielding any of the other solutions. My view is that it would take a designer with unusual insight to reliably proceed toward the preferred solution using any of these techniques. I do feel, however, that for the class of problems represented by the McDonald's example, people are more likely to derive the preferred solution

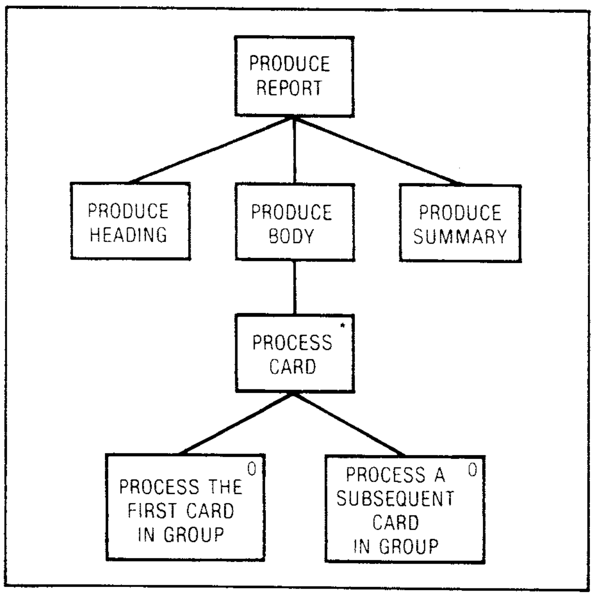


Figure 38. Functional decomposition structure diagram.

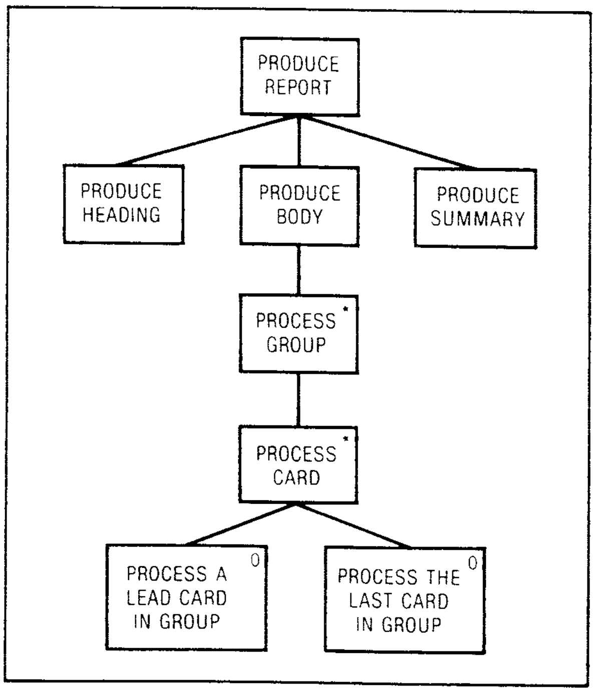


Figure 39. Data flow structure diagram.

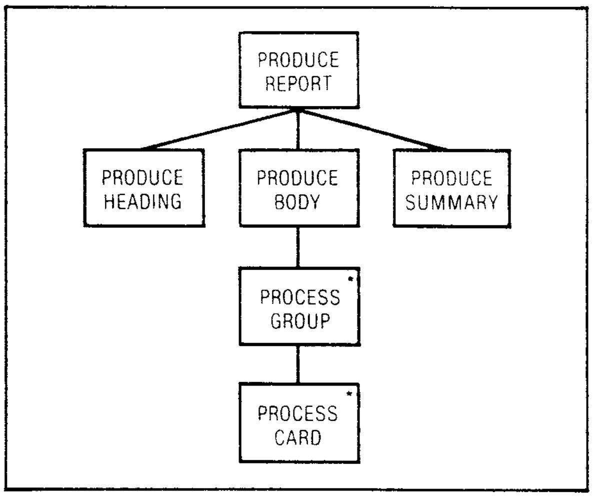


Figure 40. Data structure structure diagram.

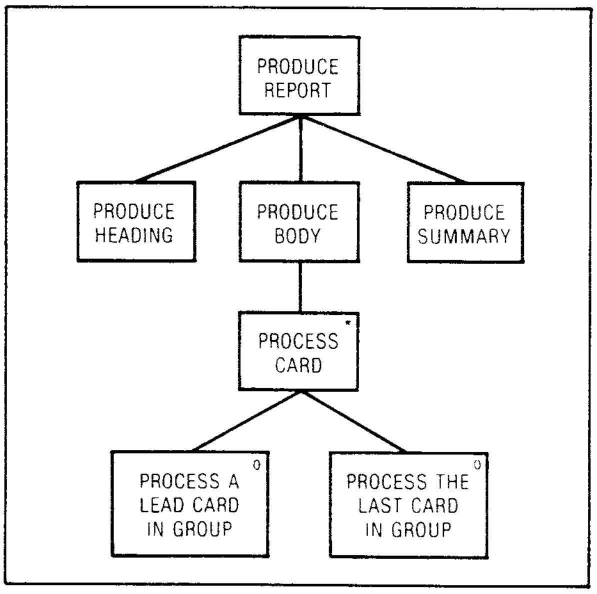


Figure 41. Programming calculus structure diagram.

by using data structure design than by using any of the other methods. For a different class of problems, of course, the results could be different.

Critique. Since functional decomposition has been around for more than a decade, there have been many well-documented success stories and even a few well-documented failures.

One success story was summarized by its designers, using the diagrams in Figure 42. Perceived project visibility was dramatically improved by the application of functional decomposition together with other techniques. They also felt that project staffing could be reduced over that normally required. In this particular project, there was a lot of personnel turnover but this was taken in stride, partly owing to the beneficial effects of the new techniques.

It seems that functional decomposition can lead to a "good" hierarchical program structure if carefully applied. If not used carefully, however, it can lead toward logical cohesion and, occasionally, toward telescoping—that is, toward defining smaller and smaller modules that are not independent but have strong coupling with each other. Applying functional decomposition to obtain mathematical functions (e.g., square root) is relatively straightforward.

For any given problem, the number of potential decompositions can be large. This makes applying the technique much more of an art than a science. I know that Dijkstra can do beautiful functional decompositions.

In Johnson's[11] words, functional decomposition seems to be a triumph of individual intellect over lack of an orderly strategy. The question "Decomposition with respect to what?" is always a point to ponder. The measures of "goodness" are difficult to apply consistently. Finally, this method requires that the intellectual tasks of problem modeling and program construction be addressed simultaneously. Ideally, these two tasks would be separated.

Because the concept of data flow design came later than pure functional decomposition, it has not been used as extensively. Apparently, several projects within IBM have used the "composite design" version of data flow design with varying degrees of success. I have personally seen a number of success stories that praise data flow design.

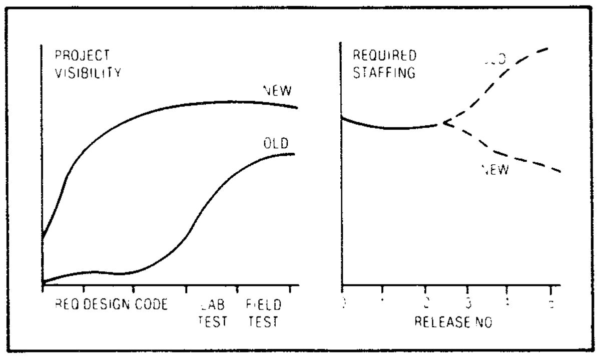


Figure 42. Experience with functional decomposition as used with hierarchical structure, stepwise refinement, high-level language, teams, walkthroughs, cause/effect charts, etc.

The concepts of coupling and cohesion, which accompanied the introduction of the data flow design method,[8] have been thought-provoking and, on some occasions, revealing. They have given people a language to express previously unverbalized thoughts.

The data flow design method can be used to produce a hierarchical program structure with all of its intrinsic advantages. The tendency is strongly toward modules with sequential cohesion at the system level, although anything can happen within the central transform. The data flow chart, which forms the basis for decomposing with respect to data flow, is a useful contribution and may be the best approach currently available at the system design level. It's not clear that the later step of putting things into a "calls" hierarchy using "transform centered design"[8] is as useful. It seems to produce a structure with a lot of data passing and adds artificial "afferent" (input) and "efferent" (output) ears to the structure chart, while reverting back to standard functional decomposition for the "central transform" which is the heart of the problem. It isn't clear that anything is gained over simply using functional decomposition from the start.

In summary, the concepts of cohesion and coupling represent a real step forward from straight functional decomposition. A qualitative measure of goodness is not as good as a quantitative measure, but it is a start. Furthermore, the data flow chart separates the modeling of the problem from detailing the structure of the program. The hard part becomes deriving the correct data flow chart (or "bubble chart"), which is still an art for large systems.

Neither the Jackson nor the Warnier methodologies for data structure design were widely used in this country until quite recently. They have, however, been used for a number of years in Europe. Jackson's method seems closer to a true methodology than the other design methodologies currently available. It is repeatable, teachable, and reliable in many applications. It usually results in a program structure that faithfully models the problem.

The data structure design method results in a hierarchical program structure, if the data structure is hierarchical. It produces multiple, independent hierarchies, if they are present in the problem environment. It is difficult to determine the level of cohesion of the modules within the resulting program structure. Sometimes it tends to be functional, in other cases communicational. By modeling the data structures and therefore the problem environment first, the problem modeling task is done before the program construction task.

While there is still no clear methodology for large systems and deriving the "correct" data structures can be difficult, it still seems that this method is a big step forward. Being able to ask whether a structure is right or wrong is somehow much more satisfying than trying to decide if it is good, better, or best.

Both Jackson's and Warnier's data structure design methods were first applied in business data processing. At this point, they have also been applied to a number of on-line problems, although they are still unproven for large real-time applications.

The number of people who use the programming calculus method regularly and proficiently is extremely small. The primary disadvantage is that a relatively high degree of logical and mathematical maturity is required to produce even "simple" programs. The mathematical proofs involved are usually several times longer than the program derived.

A second and perhaps less important disadvantage is that this method admits the existence of multiple solutions to the same problem. Different choices of an invariant assertion can lead to different program structures. The resulting programs do not necessarily portray accurate and consistent models of the problem's environment or its solution. That is, a "correct" program may still have the "wrong" structure.

In spite of these problems, Dijkstra's programming calculus design discipline is an encouraging step forward on the road to developing correct programs. It is a method that you should be aware of, for it holds promise for the future.

Summary. As shown in Figure 43, many claims have been made about the different strategies for designing software. For functional decomposition, the proponents have largely said, "D is good design, believe me." For data flow design methods, people have said, "Program C is better than program D. Let me tell you why." For data structure design methods, the claim is that "B is right; C and D are wrong. A program that works isn't necessarily right." In the programming calculus, the contention is that "Program A is probably correct. B, C, and D are unproven."

In my view, there is still much room for innovation in the area of program design methodologies. The current state of the art was represented schematically by Johnson[11] in the form of Figure 44. Functional decomposition has been described as the ideal methodology for people who already know the answer. The other three methodologies seem less reliant on knowing the answer before you start.

All of the methodologies rely on some magic. For functional decomposition, the magic gets applied very close to the end product. In the other three methodologies, at least some of the magic gets applied at the problem-model level. Clearly, we need to get a much better handle on the magic part of all four of the methodologies.

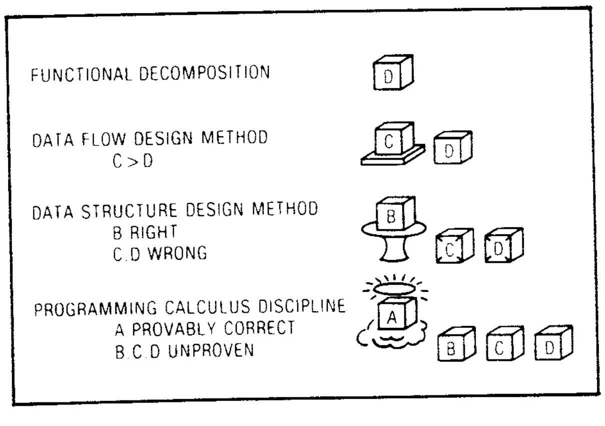


Figure 43. Summary of program design methodology claims.

Prognosis. Many different design methodologies are available. Although the first three methodologies discussed in this article have been used extensively and to a large measure successfully, there is still much to be done.

Desired results. The results I would like to see from the work yet to come are

- a complete methodology for partitioning "big" problems,
- better documentation for both the shortcomings and attributes of existing methodologies,
- guidelines for combining methods when appropriate,
- a generally accepted metric for quantifying program complexity (or entropy),
- help for the four out of five programmers who are maintaining and enhancing old programs, and
- more published examples of real-time applications.

Hopes for the future. Some possible requirements on a "complete" methodology could include the following:

- It must include a rational procedure for partitioning and modeling the problem.
- It should result in consistent designs when applied by different people.
- It must systematically scale upward to large problems while interacting consistently with a model of the real world.
- It must partition the design process as well as the problem solution.
- The correctness of individual design steps must guarantee the correctness of the final combination.
- It should minimize the innovation required during the design process. The innovation should occur in the algorithm specification phase.

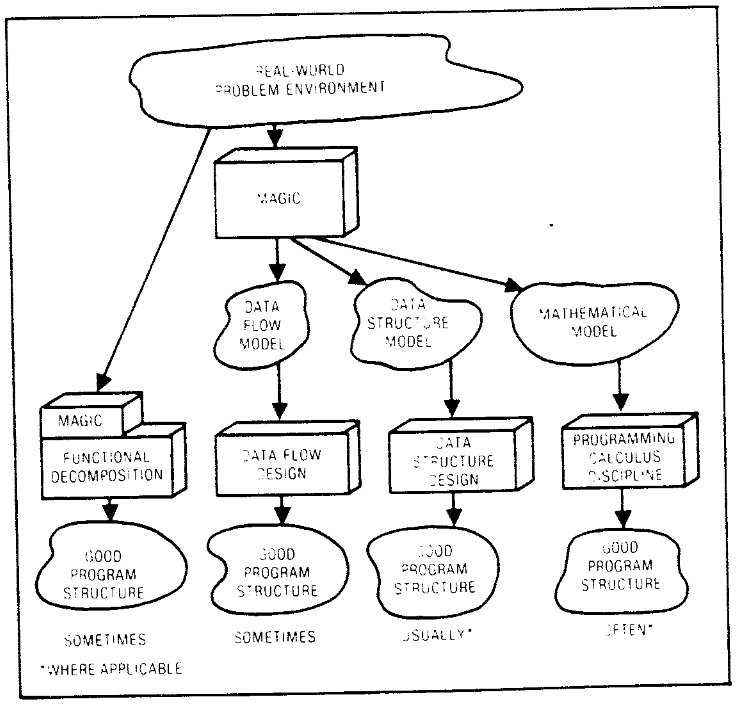


Figure 44. Current state of the art.

An interim procedure. Until we know the "right" method for designing the structure of a large program, I would propose the following interim procedure:

- Define the high-level language and operating system macros bottom-up, using the principle of abstraction to hide the peculiarities of the hardware and to create a desirable virtual machine environment.
- Map the system flow diagram into your virtual machine environment.
- Work through the inversion[7] process or construct a data flow model of the "big" problem.
- Construct data structure diagrams that correspond to each data flow path.
- Cluster and combine bubbles that can be treated by one of Jackson's "simple" programs.
- Use the Jackson data structure design method to combine simple programs and reduce the number of intermediate files.
- Implement each cluster as concurrent, asynchronous processes if you are operating under a suitable programming environment—for example, Simula or Unix.[26]*
- If a decent operating system is not available, construct some scheduling kludge.

While these interim suggestions do not fit together well enough to call them a method, they may form a reasonable approach to follow until a true methodology for large problems is found. ■

*Unix is a trademark of Bell Laboratories.

Acknowledgments

Much of the material presented here was developed and discussed over the last several years with R. D. Gordon and J. W. Johnson. The functional decomposition and data structure design solutions to the McDonald's problem are extensions of work described by Michael Jackson. The programming calculus solution to the McDonald's problem was developed after discussions with F. Schneider, G. Levin, and R. D. Gordon. Their inputs and support are gratefully acknowledged.

Several excellent suggestions from R. R. Conners, B. Dwyer, M. R. Gornick, R. C. Hansen, W. R. Nehrlich, and D. Sharma were incorporated in the final draft of this article.

References

1. J. N. Buxton, "Software Engineering," *Programming Methodology,* ed. D. Gries, Springer-Verlag, New York, 1978, pp. 23-28.
2. B. W. Boehm, "Software Engineering," *IEEE Trans. Computers,* Dec. 1976, Vol. C-25, No. 12.
3. D. L. Parnas, "Software Engineering or Methods for the Multi-Person Construction of Multi-Version Programs," *Programming Methodology,* Lecture Notes in Computer Science, No. 23, Springer-Verlag, New York, 1975, pp. 225-235.
4. V. A. Vyssotsky, "Software Engineering," keynote speech delivered at COMPSAC 79, Nov. 6-8, 1979, Chicago, Ill.
5. G. D. Bergland and R. D. Gordon, *Tutorial: Software Design Strategies,* IEEE Computer Society Press, Silver Spring, Md., 1979, pp. 1-14.
6. G. M. Weinberg, *The Psychology of Computer Programming,* Van Nostrand Reinhold, New York, 1971.
7. M. A. Jackson, *Principles of Program Design,* Academic Press, New York, 1975.
8. E. Yourdon and L. L. Constantine, *Structured Design,* Yourdon Press, New York, 1975.
9. G. J. Myers, *Reliable Software Through Composite Design,* Petrocelli/Charter, New York, 1975.
10. B. Liskov and S. Zilles, "Programming with Abstract Data Types," *SIGPLAN Notices,* Vol. 9, No. 4, Apr. 1974, pp. 50-59.
11. J. W. Johnson, "Software Design Techniques," *Proc. Nat'l Electronics Conf.,* Chicago, Ill., Oct. 12, 1977.
12. F. P. Brooks, Jr., *The Mythical Man-Month,* Addison-Wesley Pub. Co. Reading, Mass., 1975.
13. E. W. Dijkstra, "The Humble Programmer," *Comm. ACM,* Vol. 15, No. 10, Oct. 1972, pp. 859-866.
14. *Implications of Using Modular Programming,* Central Computer Agency, Her Majesty's Stationery Office, London, 1973.
15. C. Alexander, *Notes on Synthesis of Form,* Harvard University Press, Cambridge, Mass., 1964.
16. L. A. Belady and M. M. Lehman, "Characteristics of Large Systems," *Proc. Research Directions in Software Technology,* Brown University, Oct. 1977.
17. F. T. Baker, "Structured Programming in a Production Programming Environment," *IEEE Trans. Software Eng.,* June 1975, Vol. SE-1, No. 2, pp. 241-252.

18. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Comm. ACM*, Vol. 15, No. 12, Dec. 1972, pp. 1053-1058.

19. R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming Theory and Practice*, Addison-Wesley, Reading, Mass., 1979.

20. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976.

21. N. Wirth, *Systematic Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

22. B. H. Liskov, "A Design Methodology for Reliable Software Systems," *AFIPS Conf. Proc.*, Vol. 41, FJCC 72, 1972, pp. 191-199.

23. J. D. Warnier, *Logical Construction of Programs*, Van Nostrand Reinhold Co., New York, 1974.

24. D. Gries, "An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs," *IEEE Trans. Software Eng.*, Vol. SE-2, No. 4, Dec. 1976, pp. 238-244.

25. E. W. Dijkstra and D. Gries, "Introduction to Programming Methodology," Ninth Institute in Computer Science, University of California, Santa Cruz, Aug. 1979.

26. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell System Technical J.*, Vol. 57, No. 6, Part 2, July-Aug. 1978, pp. 1905-1929.

Glenn D. Bergland is head of Bell Telephone Laboratories' Digital Systems Research Department in Murray Hill, New Jersey. In 1966, when he joined Bell Telephone Laboratories in Whippany, New Jersey, he conducted research in highly parallel computer architectures. In 1972, he became head of the Advanced Switching Architecture Department in Naperville, Illinois. Later, he became head of the Software Systems Department, which was involved in feature development for the No. 1 electronic switching system. Currently, his major research areas are software design methodologies, digital telecommunications services, personal computing, and nonstop computer systems.

Bergland received the BS, MS, and PhD in electrical engineering from Iowa State University in 1962, 1964, and 1966. He is a member of the IEEE and ACM.