Some Notes on Program Design

A good program is not an accident. Programming is not accomplished by luck, coincidence, or happenstance. Good programming is by design.

Programming by design requires some forethought with regard to the structure and outcome of the programming effort. What sort of software product will result? Who will use it? Is it expected to change over time? Programming by design encompasses a large variety of techniques that have been developed since the 1960s, beginning with modular programming up to the current interest in automated program design environments.

What Is A Good Program?

Before examining techniques for constructing a good program, it is reasonable to try to identify the species. Over the years, our conception of what makes a program "good" has undergone some radical changes. A case in point: FORTRAN programs were considered "good" if they produced efficient, executable code. While this is a valid objective, it is not sufficient today.

A good program does what it is supposed to do in the best possible way. Now, as a definition, this is altogether too vague, especially the part about "the best possible way." Several desirable program characteristics have been identified. A "good" program ought to be valid, readable, modifiable, efficient with regard to programmer time, and efficient with regard to storage space and processing time. These are general criteria that can be applied to any program. Other criteria may be more specialized and application dependent, such as the real-time constraints built into an on-line query system.

Validity. First and foremost, a program must be correct-that is, it should do what it is supposed to do without error. We can distinguish two kinds of validity, internal validity and external validity. *Internal validity* refers to the verifiability of the program. A program with internal validity meets the specifications and design goals that have been set down for it and does not contain "bugs." This is usually the objective when a program is tested for correctness. Of course, while a program may be proved to contain errors, it is impossible to prove that a program is correct.

External validity refers to the usefulness of the program. A program with external validity solves the problem that motivated its design in the first place. After all, there is no value in constructing a perfect program for the wrong problem. These two types of program validity can be expressed by the questions:

- Is the program right (internal validity)?
- Is it the right program (external validity)?

Readability. In addition to providing a means by which a programmer communicates instructions to a machine, a program ought to be understandable by human beings. This requirement becomes increasingly important as the programming function becomes increasingly more managerial and less technical. This changing role of programming may be seen in several trends: larger programming efforts, programming teams, auditing of programs, separation of analysis and design, maintenance programming.

Modifiability. There no longer exists the concept of a "throw-away" program. If it ever did exist, it was tied to the image of the lone scientist/theoretician, working out of a closet somewhere, who writes a long, unintelligible program in order to solve a terribly complicated mathematical problem and, once solved, the program is no longer needed. If this sort of programmer ever did exist outside of the collective imagination of the science fiction community, it was a short-lived existence.

The reality is that software is now treated as an asset of the firm that owns and uses it. Programs are no longer coded-they are *designed.* Programs are no longer executed they are *implemented.* Programs are no longer "thrown away" --they are *maintained.* Maintenance programming accounts for much of a firm's programming effort. This means that most programming time is spent modifying programs as opposed to creating new applications. It is only sensible that some of that effort be put into a program right at the start to make it easy to modify later on.

Efficiency. It is indicative of the way that programming has changed over the last several decades that, while in 1954 the designers of FORTRAN put machine efficiency at the top of their list of requirements (although not before program correctness!), nowadays it comes last. This is not to minimize the importance of efficiency in programming. Efficiency considerations should permeate the entire programming effort. However, efficiency ought to be sacrificed in favor of modifiability or readability.

Efficiency concerns lie in two arenas: the human and the machine. If a program is *programmer-efficient,* then human time has been optimized; if it *is machine-efficient,* then CPU time and storage space have been optimized. Which would you suppose should have higher priority? Consider the following trends:

1. Computers are faster, more powerful, contain more storage space, and cost less.
2. Programs are larger, more complex, and take longer to write.
3. Programmers work slowly (due to 2) and cost a lot.

Given these trends, it is clear why programmer efficiencies have taken precedence over machine efficiencies.

In addition, there is the concept of an *elegant* program. This does not, as some would have you believe, apply to a program that employs clever tricks or to single-statement programs. (These are sometimes called "write-only" programs since that is all you can really do with them.) An elegant program goes beyond simply meeting the specifications. It is composed of *clear, simple, readable* code. It gets the job done in the best possible way.

The Well-Written Program

What does it take to write a good program? A measure of problem solving, skill, certainly. Beyond that, it is unclear.

If we look at people whom we identify as good programmers, we would find a host of different skills and, probably, very few common characteristics. One problem with this, of course, is that we might identify the good programmers incorrectly. Someone who works very hard on a project and lets you know about it may not be as good at the job as someone who works quickly, quietly, and efficiently - but the latter may appear to be someone who gets easy assignments rather than a good programmer. Beyond having problem-solving ability, then, what can we say about a good programmer? There is one common characteristic that comes up repeatedly-the ability to *communicate.*

The fact is that a person who is capable of writing English can write a program. And someone who is capable of writing *well* is capable of writing a good program. In fact, this has been identified repeatedly as a major factor in the design of a good program. According to some it is *the* major factor. The myth of the semi-illiterate hacker is just that - a myth.

Writing and Programming

How is writing a program similar to "just plain writing"?

Communication. First of all, the primary activity in both writing and programming is the same. This activity is communication. Human beings use writing to communicate with each other, sometimes across vast distances, or over great spans of time. Similarly, a program is a medium of communication between programmer and machine, between programmer and programmer and, sometimes, between programmer and nontechnical personnel. The communication skills that you acquire in one spoken and written language can usually be transferred and put to good use when you communicate in another language. In other words, a good communicator has skills that transcend the syntax of any one particular language. The same is true for computer programming languages. A good programmer can communicate in FORTRAN or C++ as well as in MODULA-2 or Ada because it is not the language that imparts this skill, it is simply an ability to communicate in a programming context.

Reading. Many people think that to be a good programmer you simply start writing programs, using brute force if necessary to accomplish the task. The truth is more civilized. A good programmer, like a good writer, ought to begin by reading the work of others. This gives you a chance to see what is being done and, perhaps, adapt it to your own problem.

Keep it simple. We have all seen written work that is unintelligible, but uses such nice big words and technical jargon that you think, "This must be good work." This syndrome is especially evident in student projects and articles in scholarly journals. This sort of work is designed to impress the reader so much that she will not bother reading it too critically. It is bad writing masquerading as good. On the other hand, we sometimes wrongly label a written work as bad if is it written simply, using clear language within a good organizational structure. We may think, "If I can understand it so easily, it must be trivial." This problem too has an exact counterpart in the realm of programming. An unintelligible program may impress us, while a simple and elegant one may appear to be "easy." As a writer or reader of programs, be careful of this pitfall.

Outline. In class, we saw that a control hierarchy chart representing a top-down program is strikingly similar to an outline. A composition, term project, or piece of fiction lacking a solid organizational structure is just as unreadable as a disorganized program.

Plan ahead. In both writing and programming, you have to have a plan. You think ahead, not only when you make up the outline but also when you approach each new paragraph. This enables you to control the complexity of the product regardless of the complexity of the original problem. For example, the structured constructs - sequence, selection, iteration - with which we build programs help us to plan ahead in a formalized manner.

In fact, research in the area of cognitive behavior has shown that experts in many different fields-including writing and programming-store information in meaningful and relevant "chunks." Looking specifically at programming experts, researchers have shown that a program can be decomposed into a set of interrelated goals, and each goal is realized by a particular plan-that is, a chunk of programming knowledge.

Cohesion. We all remember learning that a paragraph should contain only a single thought. Paragraphs, sections, and chapters share the requirement of program modules: They must be cohesive.

Esthetics. In much written work, "white space" is important. It enhances the readability of the finished product even if it does not change its meaning. For instance, in "action" novels, there is often a lot of dialogue and short paragraphs. The page is more pleasing to the eye and you end up turning pages faster in keeping with the

pace of the story. As another example, in print advertising, advertisements with a lot of copy are less likely to be read even if they provide valuable information. Similarly, in order to enhance program readability, we use a number of techniques that come under the heading of "esthetics." These include such devices as the use of blank spaces and blank lines, indentation, and the placement of borders around blocks of comments.

Read it through. Finally, just as you would ask a friend or colleague to review a piece of writing you have just finished - whether it is a composition, article, story, or novel - programmers submit their finished work to structured walkthroughs by a committee of their peers.

Structured Programming Methodology

The term *structured programming* refers to a collection of programming techniques for implementing program and control abstractions in a hierarchical manner. The use of structured programming methodology in program design increases the likelihood of producing a program that is correct, readable, modifiable, and efficient. The techniques of structured programming may be used with any of the computer programming languages on the software ladder, from the lowest machine code to the highest machine-independent languages.

The term *structured programming* has come to represent different things to different people. We use it here to represent a body of techniques that may be applied to a large and complex program design task in order to produce programs that are logically manageable, well-structured, and easily tested for correctness. Structured programming methodology dictates that correctness be incorporated into the design of the program at the outset rather than having to "debug" errors out of it after the program has been completed.

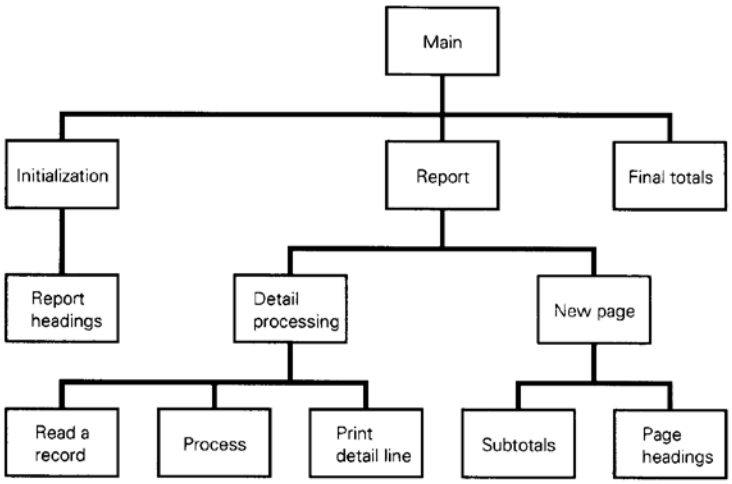
Structured programming methodology encompasses three techniques:

1. the *modular* decomposition of programs into collections of modules, or subprograms
2. the *top-down* or hierarchical ordering of these modules
3. the use of *structured control constructs* at the detailed logic flow level.

The development of a program's structures utilizes the techniques of modular decomposition and top-down design to design a program in hierarchical levels of program abstraction. These techniques have already been discussed fully in Chapter 5. The statement-level control structures are drawn from the structured control constructs that are abstractions of control. These have already been discussed in Chapter 6. In order to pull these three techniques together under the umbrella of structured programming, we briefly review them here.

Modular Decomposition. Modularization of a program involves the identification, definition, and construction of relatively independent blocks of program code, called modules. These may be external subprograms, internal subprograms, or simply groups of statements within a program unit, in order of decreasing independence. Each module performs a clearly defined task and is characterized by a single entry point and a single exit point. Thus the module can represent a single action and is a means of building abstraction into a program.

Top-Down Design. Although modularization dictates that the program be decomposed into blocks of code, it doesn't say anything about how this decomposition ought to proceed. The top-down program-design technique assumes a modular program and organizes these modules into a hierarchical structure. Much like the outline you create before embarking on a term project, top-down design is characterized by stepwise refinement and delayed decision. An overview of a modular, hierarchical program is depicted in the figure below.



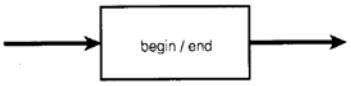
At any level in the hierarchy, the program can be regarded as complete and tested for correctness. We can do this by assuming an underlying abstract machine that contains operations corresponding to the lower level modules as yet uncoded. This is a boon to the management of complexity in programming but, of course, at the next level we are rudely awakened and forced to resume our coding. In this way, quite a bit of programming work is done while we defer lower level decisions to a later time. Of course, this design technique should not be used to justify procrastination-endlessly deferring critical decisions so that the abstract machine stays abstract.

Structured Control Constructs. Structured control constructs are statement-level control structures that may be described as either (low-level) stylized sequences of instructions or high-level language statements, each with a single entry point

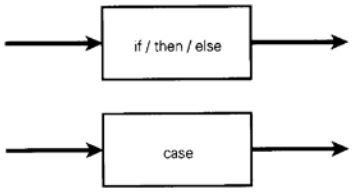
and a single exit point, which fall into one of the following three categories: simple sequence, selection, iteration. A construct is a building block. In this case, these are the building blocks from which the program is constructed, piece by piece.

The **goto** statement is not a structured control construct, and it is to be avoided as much as possible. Abstract control structures are more problem oriented and less procedural than the equivalent code using goto statements. These constructs facilitate the clear, natural expression of algorithms. Additionally, it is theoretically (and provably) possible to construct any program with only the three structured constructs mentioned and without a single use of the **goto.** However, there is no reason to be obsessive about it. Our main concern is with the readability and natural expressivity of the program. If the most natural way of expressing an algorithm is with a **goto** statement, then it is the one to use, but try *not* using it first. Forcibly removing a well-placed **goto** results in the same kind of cute programming tricks that characterized first- and second-generation programming and early FORTRAN programming; they were done then in the interest of machine efficiency.

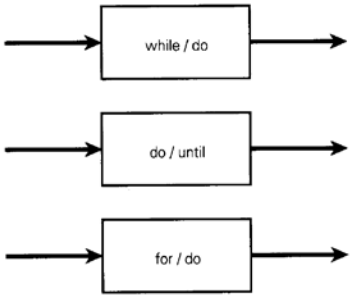
Structured control constructs are control abstractions, just as subprogram units are program abstractions. The single-entry/single -exit requirement in both cases allows us to think in terms of a single action performed by a particular program or



(a) Simple sequence



(b) Selection



(c) Iteration

control structure. We use the Black Box model to visualize this abstraction. In fact, the "module" is central to this idea of abstraction. Fig. 14.2 illustrates this view of a module as any block of code. Thus, a program is a module; internal and external subprograms are modules. Structured control constructs, with the single-entry / single-exit characteristic, may be considered modules as well. The figure below pictures specific control structures as Black Box abstractions.

The *simple sequence* construct is, as the name implies, the simplest construct. Simple sequence is the default control structure. It consists of a statement followed by another statement (followed by another statement and so on):

Statement l
Statement 2
Statement 3

Each of these statements may be a single executable statement (with the exception of the **goto** statement), a subprogram call, or another structured control construct. The compound statement is a language structure that implements the simple sequence construct, in the following way:

```
{
statementl
statement2
statement3
}
```

The *selection* construct is used when one wishes to choose from among two or more mutually exclusive alternatives. Selection may take the form of the **if/then, the if/then/else,** or the **case.** To choose between two alternative statements:

```
if (condition)
   statementl;
else
   statement2;
```

A statement may be a single statement, a subprogram call, or a control construct. If, for example, more than one statement is needed in the "then" clause or the "else" clause, a simple sequence construct is used. Sometimes we only want to "do something special" if the condition is true:

```
if (condition)
   statement;
```

This is the control construct that typifies control-break processing, for example, end-of-page processing.

To select from among more than two mutually exclusive alternatives we can always use a nested **if/then/else,** in which the "else" clause contains another **if/then/else** construct. This is just one of the ways in which we can combine these individual building blocks in order to produce a structured program. A switch / case structure does the same thing. The switch/case statement is not a critical one, but it assists in the natural expression of certain algorithms whose clarity might otherwise be masked by a large sequence of nested **if/else** constructs.

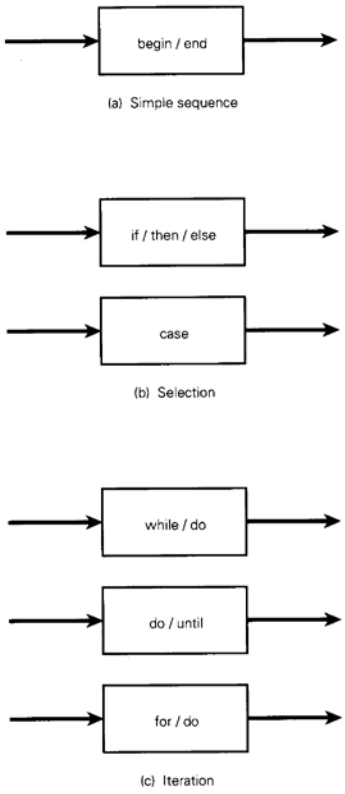
The *iteration* construct is used when a statement must be repeated. The only two forms of this construct that are theoretically necessary are the **while** (test-before) and **do/while** (test-after). The **while** is used in situations where the test is to be done before entering the loop:

```
while(condition)
     statement;
```

The do/while is used where the test is to be done after each repetition of the loop:

```
do
statement;
while(condition)
```

As usual, statement can be a single statement, a subprogram call, or another structured control construct. The difference between these two iteration constructs is, obviously, the position of the exit condition, the test for exiting the loop. The **do/while** construct will execute the loop at least once; with the **while** construct, the body of the loop may be bypassed completely if the condition is not met the first time into the loop.

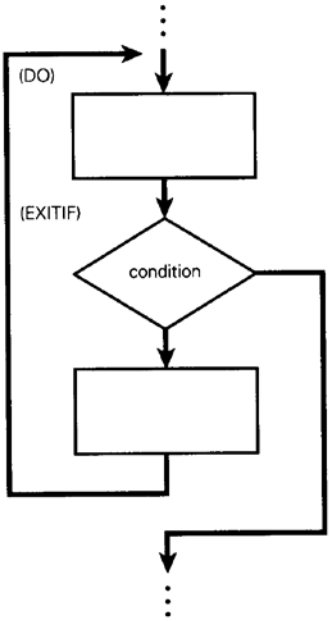


A more general form of the iteration construct that is seldom used includes both of these constructs as special cases. Pictured in flowchart form in the figure below, it might be coded something like this:

```
do {
     statementl;
if (condition) break;
     statement2;
}//end do
```

statementl and statement2 are defined as above. At each iteration:

- statementl, or the first part of the loop, is executed;
- then a test on the exit condition is performed in order to determine whether to continue iterating or to exit the loop;
- then statement2, or the second part of the loop, is executed;
- and then control of execution returns to the beginning of the loop to start again.



This "test-in-the-middle" loop is sometimes called a *general loop* since any loop can be considered a special case. For example, a null (missing) statement1 reduces to a **while** loop and a null statement2 reduces to a **do/while** loop. From a syntactic point of view, this construct is fictitious. Programming language designers steer clear of it because the **break** is too closely related to the conditional branch instruction **(if/goto).** In fact, when such a structure is needed, that is how it would be coded.

Another iteration construct is the *indexed* (or counting) loop. The indexed loop is characterized by an index variable that is incremented (or, sometimes, decremented) at each iteration and serves to move, or control, the loop.

```
for (i=1; i<=n; i++) {
   statement;
}
```

This construct actually contains four well-defined parts:

- the *initialization* of the index variable is performed before the loop is entered;
- the body of the loop;
- the *incrementation* of the index variable;
- the *testing* of an exit condition, which compares the index variable against a final value.

This construct, though not theoretically necessary, is useful for code segments which, rather than being condition-controlled, must be repeated a certain number of times. This is the oldest form of the iteration construct.

These three constructs - simple sequence, selection, and iteration - can be nested to build any program. For example:

```
while (condition)                                   =>iteration
{
   statement;                                       =>simple sequence
   statement;
      if (condition)                                =>selection
         do {                                       =>iteration
            statement;
         }while
      else
         {                                          =>simple sequence
         statement;
         statement;
         statement;
         }
}
```

Notice that this nesting of control structures still meets the single-entry/single-exit requirement. Of course, the smaller the modules in the program, the fewer levels of nesting necessary, and the more understandable the program code will be.

It is important to emphasize that the high-level language syntax elements discussed here are not prerequisites for the design and development of a well-structured program. You can program effectively in any programming language provided you are not bound by that language's built-in control abstractions. Every high-level syntax structure, from the **compound** statement to the **switch/case** statement, can be coded using only low level structures.

Software Engineering

We have come a long way from the octal-coded programs and open subroutines of the 1950s. Computers are more powerful. Programs have become bigger and more complex. The technology of program design is aimed at managing this complexity.

Software engineering encompasses techniques for analysis, design, testing, validation, and maintenance of software. The discipline of software engineering can probably be traced back to the advent of structured programming techniques of the early 1970s.

Structured programming was a huge improvement over trial-and-error and spaghetti code. In fact, we can say that it has been a definite success. However, structured programming only goes so far. For very large and complex software systems, structured programming methodology breaks down. That is why other techniques have been developed, not to replace structured programming methods but to enhance them. These other techniques include data abstraction, information hiding, structured system analysis,

database methods, database-centered fourth-generation tools (4GTs), object-oriented programming, object-oriented design, editing environments, rapid prototyping, and automatic documentation.

Software Engineering Concepts

The following are some of the important concepts of software engineering. All share the same objective: reducing the complexity of the problem, thereby making the resulting programming task more manageable.

Modularity. Modularity is probably the oldest software engineering concept, dating back to the early 1960s. It predates and provides a solid basis for the ideas of structured programming. High cohesion and low coupling are considered critical qualities for modules.

Structure. Modular programs may be structured in some way using one of a number of design methods-for example, functional decomposition, data flow, by data structure design, etc. The structure of a program is designed by specifying well-defined relationships among the modules. For example, the modules of a program may be related in a network of relationships or in a hierarchy-that is, a tree structure. A modular program constructed using *top-down design* results in a hierarchical structure. The resulting structure diagram is in tree form. Hierarchy is good at reducing the complexity of a problem, but not very good at encouraging "reusable code." This is because the rigid tree structure does not provide for common modules.

Information hiding. When modules are well designed, they are relatively independent. They communicate with each other only through well-defined interfaces. A "user" module does not require access to all the implementation details-such as local variables, algorithms-of the "used" module. This unnecessary information may be hidden from the user, protecting the integrity of individual modules and reducing the confusion that comes along with too much information.

Abstraction. All abstraction uses the concept of information hiding. Any abstraction allows one to ignore the tedious details (at least temporarily) involved in building a system and concentrate on the larger picture. Abstraction is the major concept used in *bottom-up design:* the construction of a large program by building layers upon layers of abstraction. As we have seen throughout this text, abstraction comes in many forms. As it relates to software engineering, we can consider data abstraction, program abstraction, and control abstraction.

Data abstraction involves the encapsulation and hiding of variables and operations needed to implement a data object. *Program abstraction* is the guiding concept underlying modularity. *Control abstraction* involves the use of high-level

control structures such as the selection and iteration constructs, monitors, coroutines, exception handlers, parallel processing-without concern for how they are actually implemented on a sequential machine.