5.4 Modular Design

Top-Down Design

Full-featured software usually requires large programs. Writing the code for an event procedure in such a Visual Basic program might pose a complicated problem. One method programmers use to make a complicated problem more understandable is to divide it into smaller, less complex subproblems. Repeatedly using a "divide-and-conquer" approach to break up a large problem into smaller subproblems is called **stepwise refinement**. Stepwise refinement is part of a larger methodology of writing programs known as **top-down design**, in which the more general tasks occur near the top of the design and tasks representing their refinement occur below. Top-down design and structured programming emerged as techniques to enhance programming productivity. Their use leads to programs that are easier to read and maintain. They also produce programs containing fewer initial errors, with these errors being easier to find and correct. When such programs are later modified, there is a much smaller likelihood of introducing new errors.

The goal of top-down design is to break a problem into individual tasks, or **modules**, that can easily be transcribed into pseudocode, flowcharts, or a program. First, a problem is restated as several simpler problems depicted as modules. Any modules that remain too complex are broken down further. The process of refining modules continues until the smallest modules can be coded directly. Each stage of refinement adds a more complete specification of what tasks must be performed. The main idea in top-down design is to go from the general to the specific. This process of dividing and organizing a problem into tasks can be pictured using a hierarchy chart. When using top-down design, certain criteria should be met:

1. The design should be easily readable and emphasize small module size.
2. Modules proceed from general to specific as you read down the chart.
3. The modules, as much as possible, should be single minded. That is, they should perform only a single well-defined task.
4. Modules should be independent of each other as much as possible, and any relationships among modules should be specified.

The following example illustrates this process.

Example 1 Figure 5.19 is the beginning of a hierarchy chart for a program that gives information about a car loan. The inputs are the amount of the loan, the duration (in years), and the interest rate. The output consists of the monthly payment and the amount of interest paid for the first month. In the broadest sense, the program calls for obtaining the input, making calculations, and displaying the output. Figure 5.19 shows these tasks as the first row of a hierarchy chart.

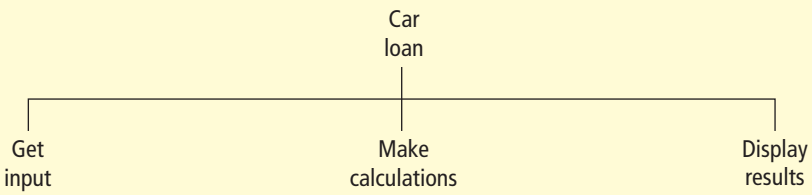

FIGURE 5.19 Beginning of a hierarchy chart for the car loan program.

Each task can be refined into more specific subtasks. (See Fig. 5.20 for the final hierarchy chart.) Most of the subtasks in the third row are straightforward and do not require further refinement. For instance, the first month's interest is computed by multiplying the amount of the loan by one-twelfth of the annual rate of interest. The most complicated subtask, the computation of the monthly payment, has been broken down further. This task is carried out by applying a standard formula found in finance books; however, the formula requires the number of payments.

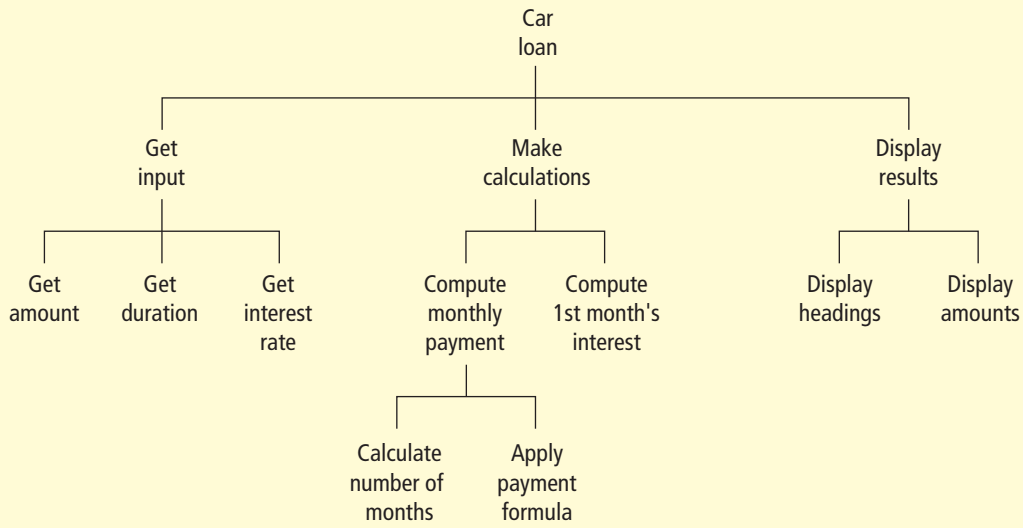

FIGURE 5.20 **Hierarchy chart for the car loan program.**

It is clear from the hierarchy chart that the top modules manipulate the modules beneath them. While the higher-level modules control the flow of the program, the lower-level modules do the actual work. By designing the top modules first, we can delay specific processing decisions.

Structured Programming

A program is said to be **structured** if it meets modern standards of program design. Although there is no formal definition of the term **structured program**, computer scientists agree that such programs should have modular design and use only the three types of logical structures discussed in Chapter 1: sequences, decisions, and loops.

Sequences: Statements are executed one after another.

Decisions: One of several blocks of program code is executed based on a test for some condition.

Loops (iteration): One or more statements are executed repeatedly as long as a specified condition is true.

One major shortcoming of the earliest programming languages was their reliance on the GoTo statement. This statement was used to branch (that is, jump) from one line of a program to another. It was common for a program to be composed of a convoluted tangle of jumps and branches that produced confusing code referred to as **spaghetti code**. At the heart of structured programming is the assertion of E. W. Dijkstra that GoTo statements should be eliminated entirely because they lead to complex and confusing programs. Two Italians, C. Bohm and G. Jacopini, were able to prove that GoTo statements are not needed and that any program can be written using only the three types of logic structures discussed before.

Structured programming requires that all programs be written using sequences, decisions, and loops. Nesting of such statements is allowed. All other logical constructs, such as GoTos, are not allowed. The logic of a structured program can be pictured using a flowchart that flows smoothly from top to bottom without unstructured branching (GoTos). The portion of a flowchart shown in Fig. 5.21(a) contains the equivalent of a GoTo statement and, therefore, is not structured. A correctly structured version of the flowchart in which the logic flows from the top to the bottom appears in Fig. 5.21(b).

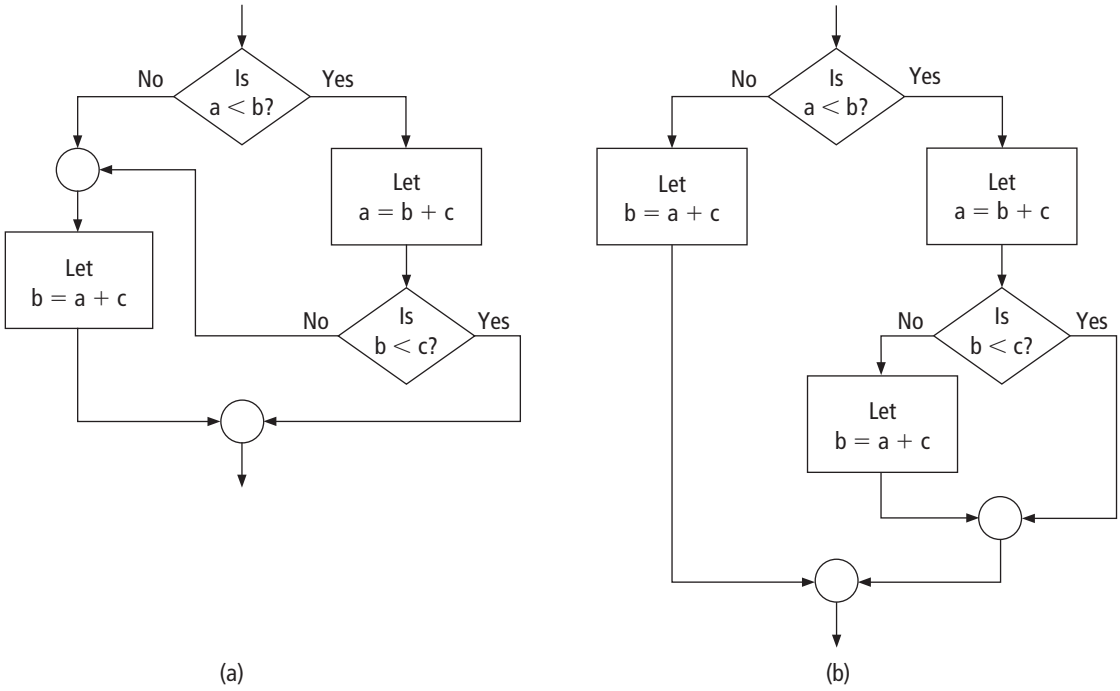

FIGURE 5.21 **Flowcharts illustrating the removal of a GoTo statement.**

Advantages of Structured Programming

The goal of structured programming is to create correct programs that are easy to write, debug, understand, and change. Let us now take a closer look at the way modular design, along with a limited number of logical structures, contributes to attaining these goals.

1. *Easy to write.*

   Modular design increases the programmer's productivity by allowing him or her to look at the big picture first and focus on the details later. During the actual coding, the programmer works with a manageable chunk of the program and does not have to think about an entire complex program.

   Several programmers can work on a single large program, each taking responsibility for a specific module.

   Studies have shown that structured programs require significantly less time to write than standard programs.

   Often, procedures written for one program can be reused in other programs requiring the same task. Not only is time saved in writing a program, but reliability is enhanced, because reused procedures will already be tested and debugged. A procedure that can be used in many programs is said to be **reusable**.

2. *Easy to debug.*

   Because each procedure is specialized to perform just one task or several related tasks, a procedure can be checked individually to determine its reliability. A dummy program, called a **driver**, is set up to test the procedure. The driver contains the minimum definitions needed to call the procedure to be tested. For instance, if the procedure to be tested is a function, the driver program assigns diverse values to the arguments and then examines the corresponding function return values. The arguments should contain both typical and special-case values.

   The program can be tested and debugged as it is being designed with a technique known as **stub programming**. In this technique, the key event procedures and perhaps some of the smaller procedures are coded first. Dummy procedures, or stubs, are written for the remaining procedures. Initially, a stub procedure might consist of a message box to indicate that the procedure has been called, and thereby confirm that the procedure was called at the right time. Later, a stub might simply display values passed to it in order to confirm not only that the procedure was called, but also that it received the correct values from the calling procedure. A stub also can assign new values to one or more of its parameters to simulate either input or computation. This provides greater control of the conditions being tested. The stub procedure is always simpler than the actual procedure it represents. Although the stub program is only a skeleton of the final program, the program's structure can still be debugged and tested. (The stub program consists of some coded procedures and the stub procedures.)

   Old-fashioned unstructured programs consist of a sequence of instructions that are not grouped for specific tasks. The logic of such a program is cluttered with details and therefore difficult to follow. Needed tasks are easily left out and crucial details easily neglected. Tricky parts of the program cannot be isolated and examined. Bugs are difficult to locate because they might be present in any part of the program.

3. *Easy to understand.*

   The interconnections of the procedures reveal the modular design of the program.

   The meaningful procedure names, along with relevant comments, identify the tasks performed by the modules.

   The meaningful variable names help the programmer to recall the purpose of each variable.

4. *Easy to change.*

   Because a structured program is self-documenting, it can easily be deciphered by another programmer.

   Modifying a structured program often amounts to inserting or altering a few procedures rather than revising an entire complex program. The programmer does not even have to look at most of the program. This is in sharp contrast to the situation with unstructured programs, where one must understand the entire logic of the program before any changes can be made with confidence.

Object-Oriented Programming

An object is an encapsulation of data and code that operates on the data. Like controls, objects have properties, respond to methods, and raise events. The most effective type of programming for complex problems is called **object-oriented** design. An object-oriented program can be viewed as a collection of cooperating objects. Many modern programmers use a blend of traditional structured programming along with object-oriented design.

Visual Basic.NET was the first version of Visual Basic that was truly object oriented; in fact, every element such as a control or a string is actually an object. This book illustrates the building

blocks of Visual Basic in the early chapters and then puts them together using object-oriented techniques in Chapter 11. Throughout the book, an object-oriented approach is taken whenever feasible.

A Relevant Quote

We end this section with a few paragraphs from *Dirk Gently's Holistic Detective Agency*, by Douglas Adams, Simon & Schuster, 1987:

> "What really is the point of trying to teach anything to anybody?"
>
> This question seemed to provoke a murmur of sympathetic approval from up and down the table.
>
> Richard continued, "What I mean is that if you really want to understand something, the best way is to try and explain it to someone else. That forces you to sort it out in your own mind. And the more slow and dim-witted your pupil, the more you have to break things down into more and more simple ideas. And that's really the essence of programming. By the time you've sorted out a complicated idea into little steps that even a stupid machine can deal with, you've certainly learned something about it yourself. The teacher usually learns more than the pupil. Isn't that true?"

5.5 A Case Study: Weekly Payroll

This case study processes a weekly payroll using the 2009 Employer's Tax Guide. Table 5.3 shows typical data used by a company's payroll office. (***Note:*** A withholding allowance is sometimes referred to as an *exemption*.) These data are processed to produce the information in Table 5.4 that is supplied to each employee along with his or her paycheck. The program should request the data from Table 5.3 for an individual as input and produce output similar to that in Table 5.4.

TABLE 5.3 Employee data.

Name	Hourly Wage	Hours Worked	Withholding Allowances	Marital Status	Previous Year-to-Date Earnings
Al Clark	$45.50	38	4	Married	$88,600.00
Ann Miller	$44.00	35	3	Married	$68,200.00
John Smith	$17.95	50	1	Single	$30,604.75
Sue Taylor	$25.50	43	2	Single	$36,295.50

TABLE 5.4 Payroll information.

Name	Current Earnings	Yr. to Date Earnings	FICA Tax	Income Tax Wh.	Check Amount
Al Clark	$1,729.00	$90,329.00	$132.27	$163.44	$1,433.29

The items in Table 5.4 should be calculated as follows:

Current Earnings: hourly wage times hours worked (with time-and-a-half after 40 hours)

Year-to-Date Earnings: previous year-to-date earnings plus current earnings

FICA Tax: sum of 6.2% of earnings if part of the first $106,800 of earnings (social security benefits tax) and 1.45% of earnings (Medicare tax)