CHAPTER 3

Understanding Structure

In this chapter, you will learn about:

- The disadvantages of unstructured spaghetti code
- The three basic structures—sequence, selection, and loop
- Using a priming input to structure a program
- The need for structure
- Recognizing structure
- Structuring and modularizing unstructured logic

The Disadvantages of Unstructured Spaghetti Code

Professional business applications usually get far more complicated than the examples you have seen so far in Chapters 1 and 2. Imagine the number of instructions in the computer programs that guide an airplane's flight or audit an income tax return. Even the program that produces your paycheck at work contains many, many instructions. Designing the logic for such a program can be a time-consuming task. When you add several thousand instructions to a program, including several hundred decisions, it is easy to create a complicated mess. The popular name for logically snarled program statements is **spaghetti code**, because the logic is as hard to follow as one noodle through a plate of spaghetti. Not only is spaghetti code confusing, the programs that contain it are prone to error, difficult to reuse, and hard to use as building blocks for larger applications. Programs that use spaghetti code logic are **unstructured programs**; that is, they do not follow the rules of structured logic that you will learn in this chapter. **Structured programs** *do* follow those rules, and eliminate the problems caused by spaghetti code.

For example, suppose that you start a job as a dog washer and that you receive the instructions shown in Figure 3-1. This flowchart is an example of unstructured spaghetti code. A computer program that is structured similarly might "work"—that is, it might produce correct results—but it would be difficult to read and maintain, and its logic would be hard to follow.

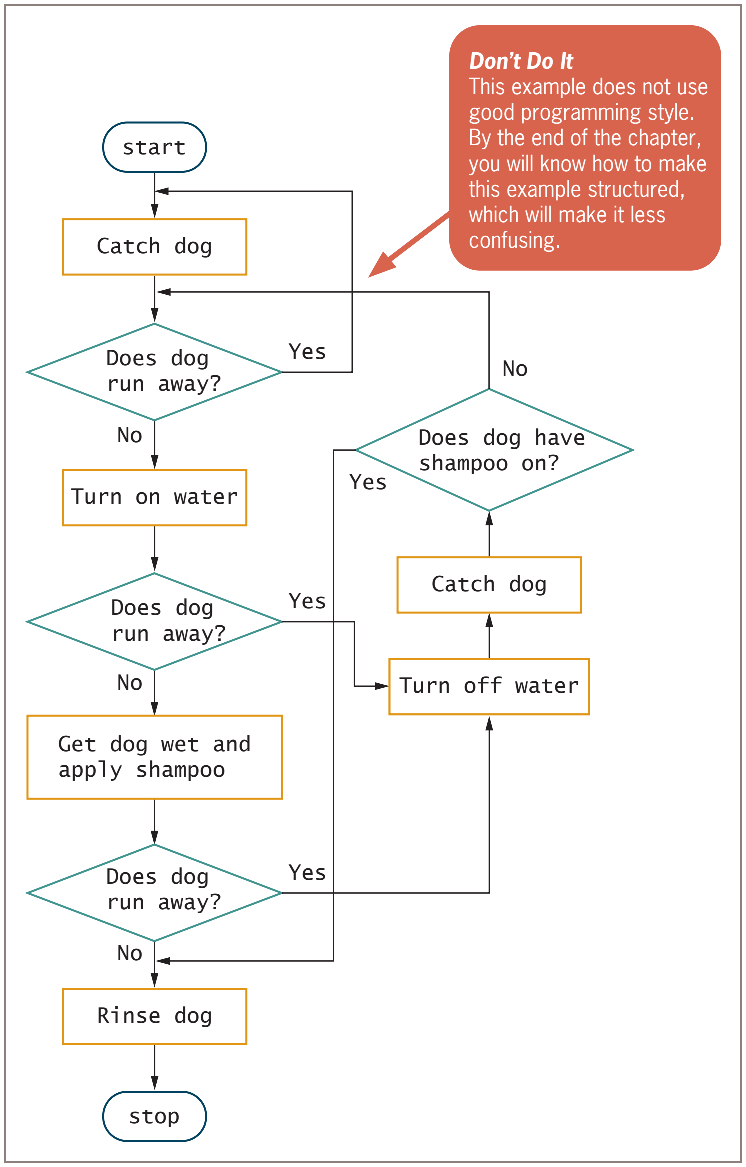

Don't Do It
This example does not use good programming style. By the end of the chapter, you will know how to make this example structured, which will make it less confusing.
start
Catch dog
Does dog run away?
Yes
No
Turn on water
Does dog run away?
Yes
No
Get dog wet and apply shampoo
Does dog run away?
Yes
No
Rinse dog
stop
Turn off water
Catch dog
Does dog have shampoo on?
Yes
No

Figure 3-1 Spaghetti code logic for washing a dog

You might be able to follow the logic of the dog-washing procedure in Figure 3-1 for two reasons:

- You probably already know how to wash a dog.
- The flowchart contains a limited number of steps.

However, imagine that you were not familiar with dog washing, or that the process was far more complicated. (For example, imagine you must wash 100 dogs concurrently while applying flea and tick medication, giving them haircuts, and researching their genealogy.) Depicting more complicated logic in an unstructured way would be cumbersome. By the end of this chapter, you will understand how to make the unstructured process in Figure 3-1 clearer and less error-prone.

Software developers say that a program that contains spaghetti code has a shorter life than one with structured code. This means that programs developed using spaghetti code exist as production programs in an organization for less time. Such programs are so difficult to alter that when improvements are required, developers often find it easier to abandon the existing program and start from scratch. This takes extra time and costs more money.

TWO TRUTHS & A LIE

The Disadvantages of Unstructured Spaghetti Code

1. The popular name for logically snarled program statements is spaghetti code.
2. Programs written using spaghetti code cannot produce correct results.
3. Programs written using spaghetti code are more difficult to follow than other programs.

The false statement is #2. Programs written using spaghetti code can produce correct results, but they are more difficult to understand and maintain than programs that use structured techniques.

Understanding the Three Basic Structures

In the mid-1960s, mathematicians proved that any program, no matter how complicated, can be constructed using one or more of only three structures. A **structure** is a basic unit of programming logic; each structure is one of the following:

- sequence
- selection
- loop

With these three structures alone, you can diagram any task, from doubling a number to performing brain surgery. You can diagram each structure with a specific configuration of flowchart symbols.

The first of these three basic structures is a sequence, as shown in Figure 3-2. With a **sequence structure**, you perform an action or task, and then you perform the next action, in order. A sequence can contain any number of tasks, but there is no option to branch off and skip any of the tasks. Once you start a series of actions in a sequence, you must continue step by step until the sequence ends.

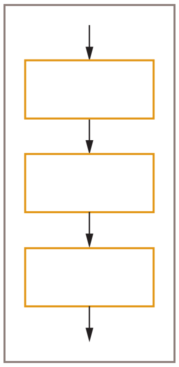


Figure 3-2 Sequence structure

As an example, driving directions often are listed as a sequence. To tell a friend how to get to your house from school, you might provide the following sequence, in which one step follows the other and no steps can be skipped:

```
go north on First Avenue for 3 miles
turn left on Washington Boulevard
go west on Washington for 2 miles
stop at 634 Washington
```

The second of the three structures is a **selection structure** or **decision structure**, as shown in Figure 3-3. With this structure, you ask a question and, depending on the answer, you take one of two courses of action. Then, no matter which path you follow, you continue with the next task. (In other words, a flowchart that describes a selection structure must begin with a decision symbol, and the branches of the decision must join at the bottom of the structure. Pseudocode that describes a selection structure must start with `if`. Pseudocode uses the **end-structure statement** `endif` to clearly show where the structure ends.)

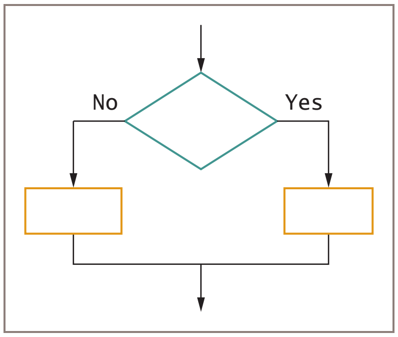


Figure 3-3 Selection structure

Some people call the selection structure an **`if-then-else`** because it fits the following statement:

```
if someCondition is true then
   do oneProcess
else
   do theOtherProcess
endif
```

For example, you might provide part of the directions to your house as follows:

```
if traffic is backed up on Washington Boulevard then
   continue for 1 block on First Avenue and turn left on Adams Lane
else
   turn left on Washington Boulevard
endif
```

Similarly, a payroll program might include a statement such as:

```
if hoursWorked is more than 40 then
   calculate regularPay and overtimePay
else
   calculate regularPay
endif
```

These `if-else` examples can also be called **dual-alternative `if`s** (or **dual-alternative selections**), because they contain two alternatives—the action taken when the tested condition is true and the action taken when it is false. Note that it is perfectly correct for one branch of the selection to be a "do nothing" branch. In each of the following examples, an action is taken only when the tested condition is true:

```
if it is raining then
   take an umbrella
endif
if employee participates in the dental plan then
   deduct $40 from employee gross pay
endif
```

The previous examples without `else` clauses are **single-alternative `if`s** (or **single-alternative selections**); a diagram of their structure is shown in Figure 3-4. In these cases, you do not take any special action if it is not raining or if the employee does not belong to the dental plan. The case in which nothing is done is often called the **null case**.

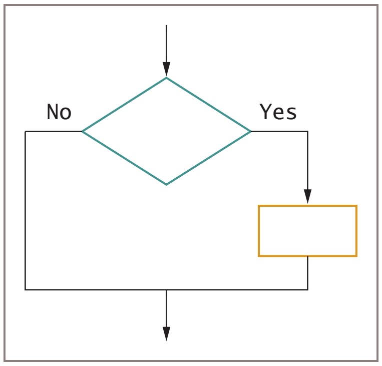


Figure 3-4 Single-alternative selection structure

The last of the three basic structures, shown in Figure 3-5, is a loop. In a **loop structure**, you continue to repeat actions while a condition remains true. The action or actions that occur within the loop are the **loop body**. In the most common type of loop, a condition is evaluated; if the answer is true, you execute the loop body and evaluate the condition again. If the condition is still true, you execute the loop body again and then reevaluate the original condition. This continues until the condition becomes false, and then you exit the structure. (In other words, a flowchart that describes a loop structure always begins with a decision symbol that has a branch that returns to a spot prior to the decision. Pseudocode that describes a loop starts with `while` and ends with the end-structure statement `endwhile`.) You may hear programmers refer to looping as **repetition** or **iteration**.

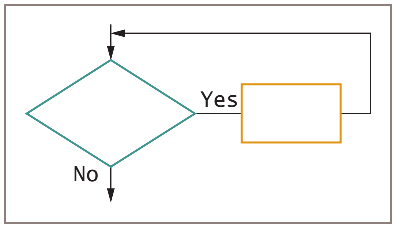


Figure 3-5 Loop structure

Some programmers call this structure a **`while`...do**, or more simply, a **`while` loop**, because it fits the following statement:

```
while testCondition continues to be true
   do someProcess
endwhile
```

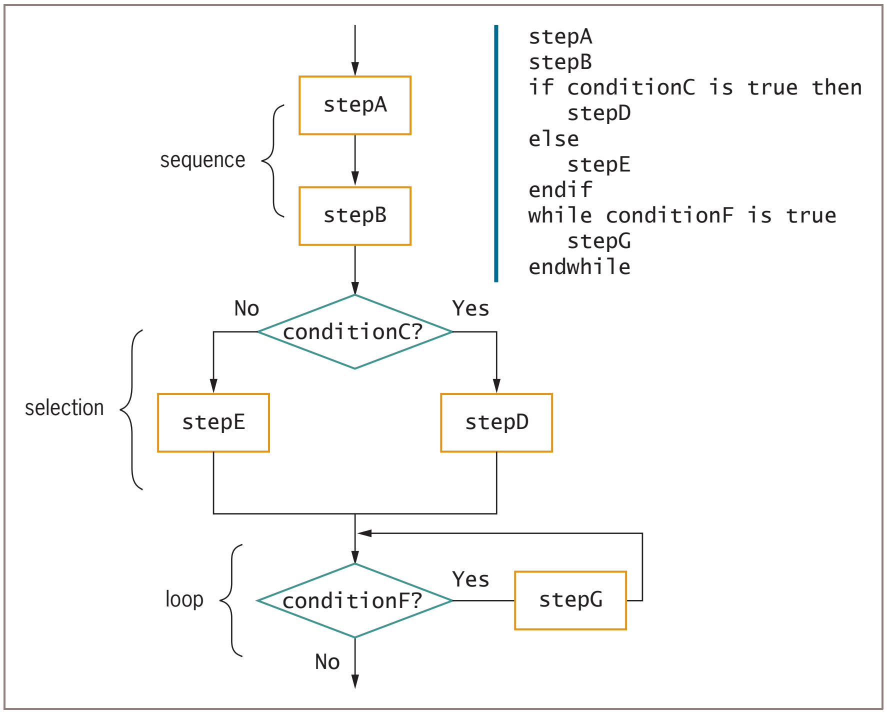
When you provide directions to your house, part of the directions might be:

```
while the address of the house you are passing remains below 634
   travel forward to the next house
   look at the address on the house
endwhile
```

You encounter examples of looping every day, as in each of the following:

```
while you continue to be hungry
   take another bite of food
   determine whether you still feel hungry
endwhile
while unread pages remain in the reading assignment
   read another unread page
   determine whether there are more pages to be read
endwhile
```

All logic problems can be solved using only these three structures—sequence, selection, and loop. The structures can be combined in an infinite number of ways. For example, you can have a sequence of tasks followed by a selection, or a loop followed by a sequence. Attaching structures end to end is called **stacking structures**. For example, Figure 3-6 shows a structured flowchart achieved by stacking structures, and shows pseudocode that follows the flowchart logic.



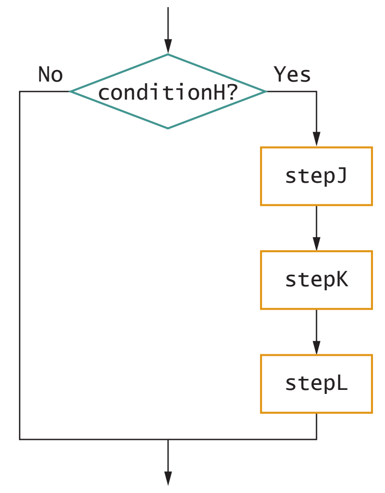
```
stepA
stepB
if conditionC is true then
   stepD
else
   stepE
endif
while conditionF is true
   stepG
endwhile
```

Figure 3-6 Structured flowchart and pseudocode with three stacked structures

Whether you are drawing a flowchart or writing pseudocode, you can use either of the following pairs to represent decision outcomes: *Yes* and *No* or *true* and *false*. This book follows the convention of using *Yes* and *No* in flowchart diagrams and *true* and *false* in pseudocode.

The pseudocode in Figure 3-6 shows a sequence, followed by a selection, followed by a loop. First `stepA` and `stepB` execute in sequence. Then a selection structure starts with the test of `conditionC`. The instruction that follows the `if` clause (`stepD`) occurs when its tested condition is true, the instruction that follows `else` (`stepE`) occurs when the tested condition is false, and any instructions that follow `endif` occur in either case. In other words, statements beyond the `endif` statement are "outside" the decision structure. Similarly, the `endwhile` statement shows where the loop structure ends. In Figure 3-6, while `conditionF` continues to be true, `stepG` continues to execute. If any statements followed the `endwhile` statement, they would be outside of, and not a part of, the loop.

Besides stacking structures, you can replace any individual steps in a structured flowchart diagram or pseudocode with additional structures. In other words, any sequence, selection, or loop can contain other sequences, selections, or loops. For example, you can have a sequence of three tasks on one side of a selection, as shown in Figure 3-7. Placing a structure within another structure is called **nesting structures**.

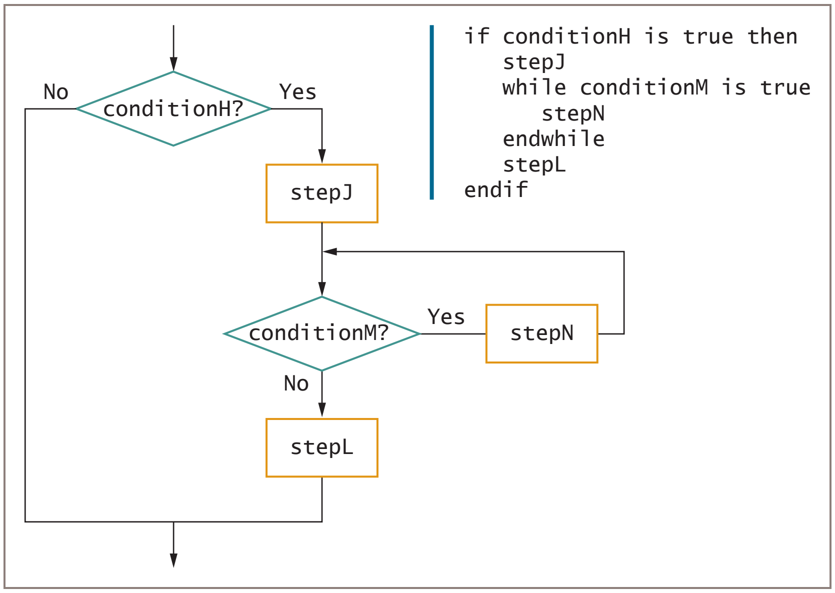


```
if conditionH is true then
   stepJ
   stepK
   stepL
endif
```

Figure 3-7 Flowchart and pseudocode showing nested structures—a sequence nested within a selection

In the pseudocode for the logic shown in Figure 3-7, the indentation shows that all three statements (`stepJ`, `stepK`, and `stepL`) must execute if `conditionH` is true. The three statements constitute a **block**, or a group of statements that executes as a single unit.

In place of one of the steps in the sequence in Figure 3-7, you can insert another structure. In Figure 3-8, the process named `stepK` has been replaced with a loop structure that begins with a test of the condition named `conditionM`.



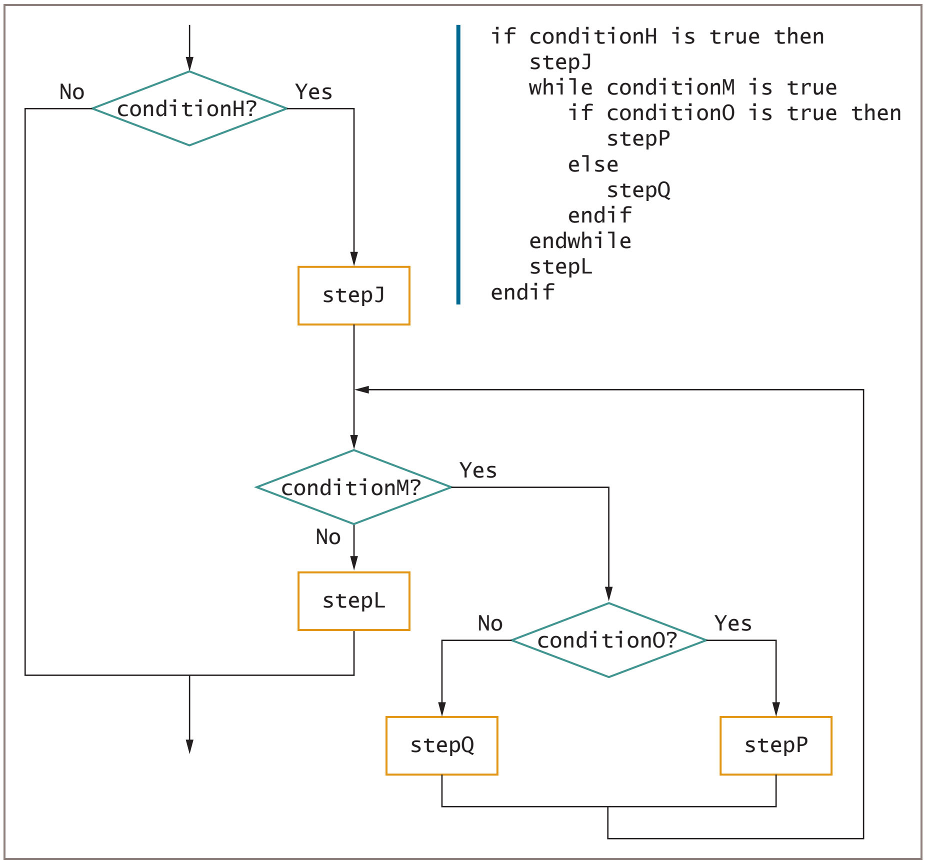
```
if conditionH is true then
   stepJ
   while conditionM is true
      stepN
   endwhile
   stepL
endif
```

Figure 3-8 Flowchart and pseudocode showing nested structures—a loop nested within a sequence, nested within a selection

In the pseudocode shown in Figure 3-8, notice that `if` and `endif` are vertically aligned. This shows that they are all "on the same level." Similarly, `stepJ`, `while`, `endwhile`, and `stepL` are aligned, and they are evenly indented. In the flowchart in Figure 3-8, you could draw a vertical line through the symbols containing `stepJ`, the entry and exit points of the `while` loop, and `stepL`. The flowchart and the pseudocode represent exactly the same logic.

When you nest structures, the statements that start and end a structure are always on the same level and are always in pairs. Structures cannot overlap. For example, if you have an `if` that contains a `while`, then the `endwhile` statement will come before the `endif`. On the other hand, if you have a `while` that contains an `if`, then the `endif` statement will come before the `endwhile`.

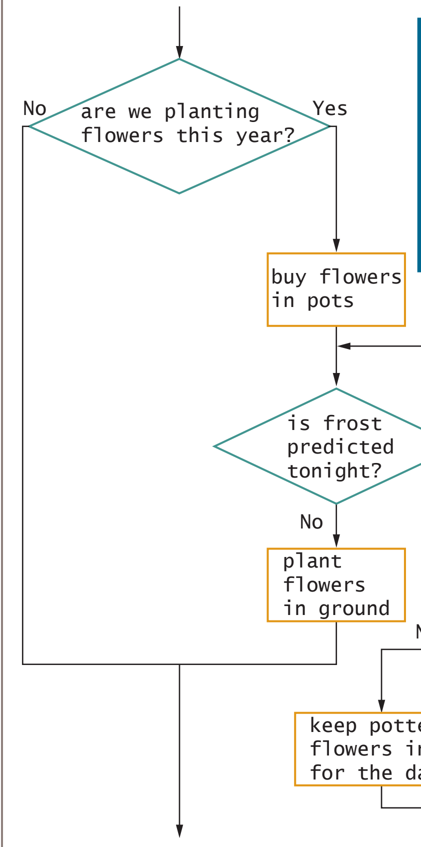
There is no limit to the number of levels you can create when you nest and stack structures. For example, Figure 3-9 shows logic that has been made more complicated by replacing `stepN` with a selection. The structure that performs `stepP` or `stepQ` based on the outcome of `conditionO` is nested within the loop that is controlled by `conditionM`. In the pseudocode in Figure 3-9, notice how the `if`, `else`, and `endif` that describe the condition selection are aligned with each other and within the `while` structure that is controlled by `conditionM`. As before, the indentation used in the pseudocode reflects the logic laid out graphically in the flowchart.



```
if conditionH is true then
   stepJ
   while conditionM is true
      if conditionO is true then
         stepP
      else
         stepQ
      endif
   endwhile
   stepL
endif
```

Figure 3-9 Flowchart and pseudocode for a selection within a loop within a sequence within a selection

Many of the preceding examples are generic so that you can focus on the relationships of the shapes without worrying what they do. Keep in mind that generic instructions like `stepA` and generic conditions like `conditionC` can stand for anything. For example, Figure 3-10 shows the process of buying and planting flowers outdoors in the spring after the danger of frost is over. The flowchart and pseudocode structures are identical to the ones in Figure 3-9. In the exercises at the end of this chapter, you will be asked to develop more scenarios that fit the same pattern.


are we planting flowers this year?
No
Yes
buy flowers in pots
is frost predicted tonight?
Yes
No
plant flowers in ground
is it over 50F today?
No
Yes
keep potted flowers inside for the day
bring potted flowers outdoors for the day

```
if we are planting flowers this year then
   buy flowers in pots
   while frost is predicted tonight
      if it is over 50F today then
         bring potted flowers outdoors for the day
      else
         keep potted flowers inside for the day
      endif
   endwhile
   plant flowers in ground
endif
```

Figure 3-10 The process of buying and planting flowers in the spring

The possible combinations of logical structures are endless, but each segment of a structured program is a sequence, a selection, or a loop. The three structures are shown together in Figure 3-11. Notice that each structure has one entry point and one exit point. One structure can attach to another only at one of these points.

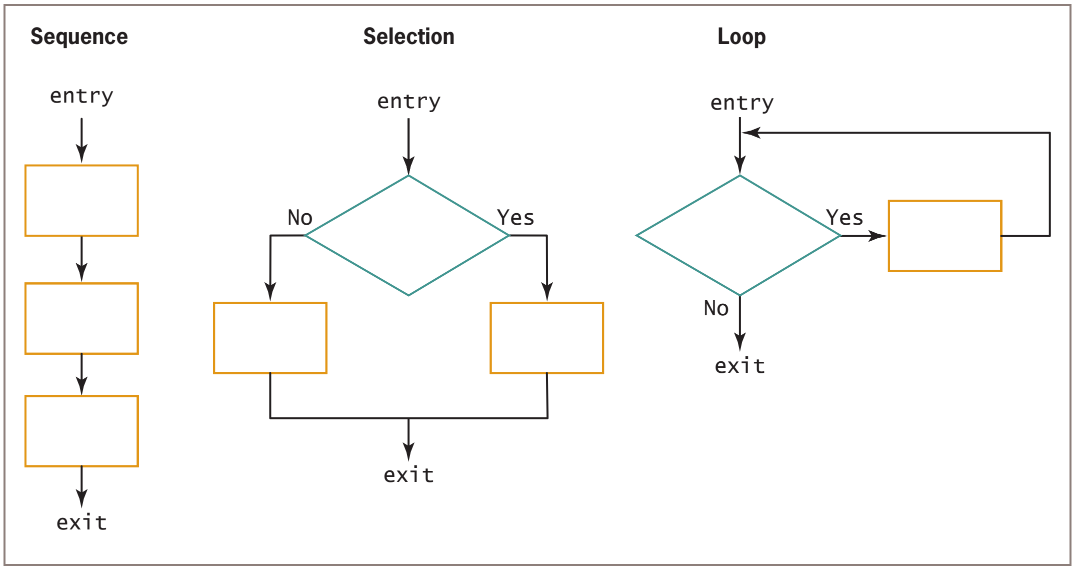


Figure 3-11 The three structures

Try to imagine physically picking up any of the three structures using the entry and exit "handles." These are the spots at which you could connect one structure to another. Similarly, any complete structure, from its entry point to its exit point, can be inserted within the process symbol of any other structure.

In summary, a structured program has the following characteristics:

- A structured program includes only combinations of the three basic structures—sequence, selection, and loop. Any structured program might contain one, two, or all three types of structures.
- Each of the structures has a single entry point and a single exit point.
- Structures can be stacked or connected to one another only at their entry or exit points.
- Any structure can be nested within another structure.

A structured program is never required to contain examples of all three structures. For example, many simple programs contain only a sequence of several tasks that execute from start to finish without any needed selections or loops. As another example, a program might display a series of numbers, looping to do so, but never making any decisions about the numbers.

Watch the video *Understanding Structure*.

TWO TRUTHS & A LIE

Understanding the Three Basic Structures

1. Each structure in structured programming is a sequence, selection, or loop.
2. All logic problems can be solved using only three structures—sequence, selection, and loop.
3. The three structures cannot be combined in a single program.

The false statement is #3. The three structures can be stacked or nested in an infinite number of ways.

Using a Priming Input to Structure a Program

Recall the number-doubling program discussed in Chapter 2; Figure 3-12 shows a similar program. The program inputs a number and checks for the end-of-file condition. If the condition is not met, then the number is doubled, the answer is displayed, and the next number is input.

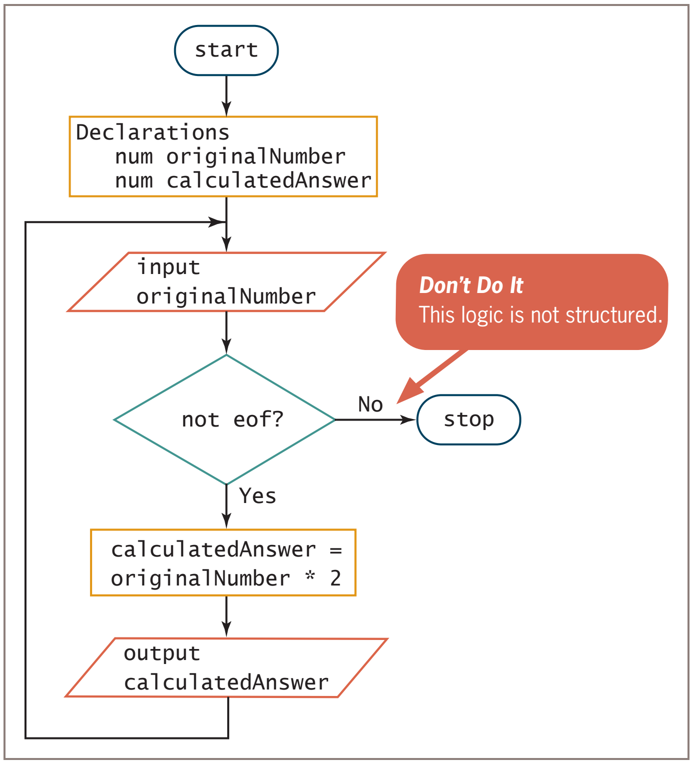


Figure 3-12 Unstructured flowchart of a number-doubling program

Recall from Chapter 1 that this book uses `eof` to represent a generic end-of-data condition when the exact tested parameters are not important to the discussion. In this example, the test is for `not eof?`, because processing will continue while the end of the data has not been reached.

Is the program represented by Figure 3-12 structured? At first, it might be hard to tell. The three allowed structures were illustrated in Figure 3-11, and the flowchart in Figure 3-12 does not look exactly like any of those three shapes. However, because you may stack and nest structures while retaining overall structure, it might be difficult to determine whether a flowchart as a whole is structured. It is easiest to analyze the flowchart in Figure 3-12 one step at a time. The beginning of the flowchart looks like Figure 3-13. Is this portion of the flowchart structured? Yes, it is a sequence of two events.

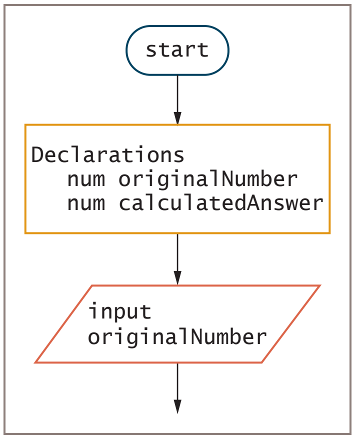


Figure 3-13 Beginning of a number-doubling flowchart

Adding the next piece of the flowchart looks like Figure 3-14. The sequence is finished; either a selection or a loop is starting. You might not know which one, but you do know the sequence is not continuing because sequences cannot contain questions. With a sequence, each task or step must follow without any opportunity to branch off. So, which type of structure starts with the question in Figure 3-14? Is it a selection or a loop?

Selection and loop structures differ as follows:

- In a selection structure, the logic goes in one of two directions after the question, and then the flow comes back together; the question is not asked a second time within the structure.
- In a loop, if the answer to the question results in the loop being entered and the loop statements executing, then the logic returns to the question that started the loop. When the body of a loop executes, the question that controls the loop is always asked again.

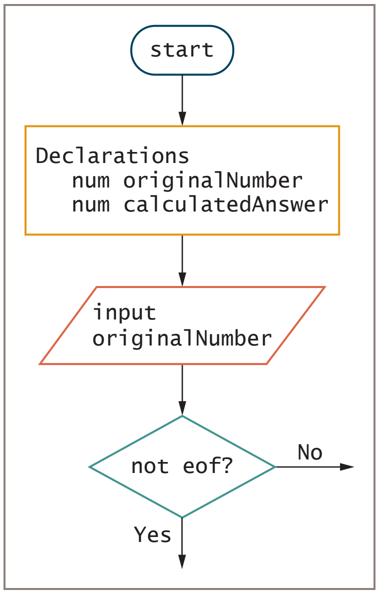


Figure 3-14 Number-doubling flowchart continued

If the end-of-file condition is not met in the number-doubling problem in the original Figure 3-12, then the result is calculated and output, a new number is obtained, and the logic returns to the question that tests for the end of the file. In other words, while the answer to the `not eof?` question continues to be *Yes*, a body of statements continues to execute. Therefore, the `not eof?` question starts a structure that is more like a loop than a selection.

The number-doubling problem *does* contain a loop, but it is not a structured loop. In a structured loop, the rules are:

1. You ask a question.
2. If the answer indicates you should execute the loop body, then you do so.
3. If you execute the loop body, then you must go right back to repeat the question.

The flowchart in Figure 3-12 asks a question. If the answer is *Yes* (that is, while `not eof?` is true), then the program performs two tasks in the loop body: It does the arithmetic and it displays the results. Doing two things is acceptable because two tasks with no possible branching constitute a sequence, and it is fine to nest a structure within another structure. However, when the sequence ends, the logic does not flow right back to the loop-controlling question. Instead, it goes *above* the question to get another number. For the loop in Figure 3-12 to be a structured loop, the logic must return to the `not eof?` question when the embedded sequence ends.

The flowchart in Figure 3-15 shows the flow of logic returning to the `not eof?` question immediately after the sequence. Figure 3-15 shows a structured flowchart, but it has one major flaw—the flowchart does not do the job of continuously doubling different numbers.

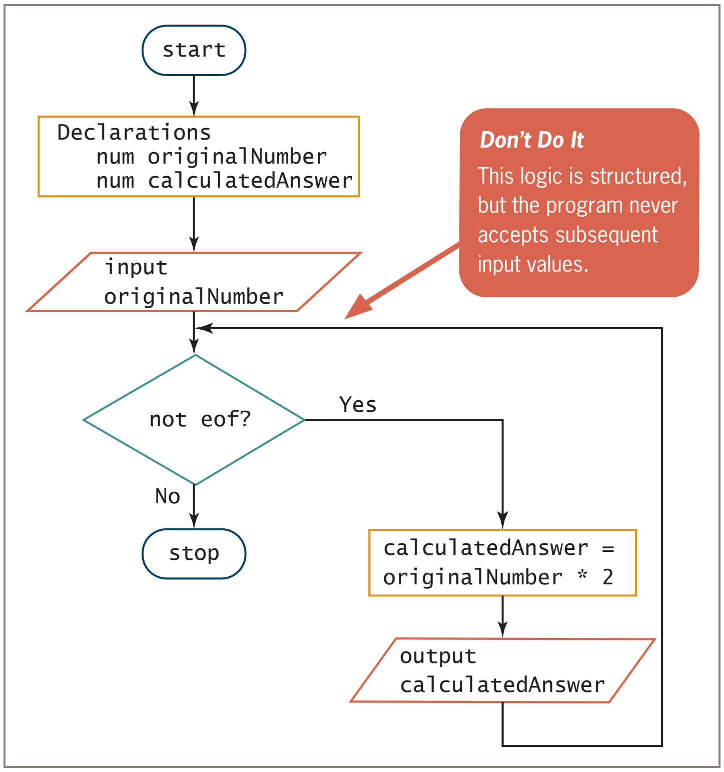


Figure 3-15 Structured, but nonfunctional, flowchart of number-doubling problem

Follow the flowchart through a typical program run, assuming the `eof` condition is an input value of 0. Suppose that when the program starts, the user enters a 9 for the value of `originalNumber`. That is not `eof`, so the number is multiplied by 2, and 18 is displayed as the value of `calculatedAnswer`. Then the question `not eof?` is asked again. The `not eof?` condition must still be true because a new value representing the sentinel (ending) value cannot be entered. The logic never returns to the `input originalNumber` task, so the value of `originalNumber` never changes. Therefore, 9 doubles again and the answer 18 is displayed again. The `not eof?` result is still true, so the same steps are repeated. This goes on *forever*, with the answer 18 being output repeatedly. The program logic shown in Figure 3-15 is structured, but it does not work as intended. Conversely, the program in Figure 3-16 works, but it is not structured because after the tasks execute within a structured loop, the flow of logic must return directly to the loop-controlling question. In Figure 3-16, the logic does not return to this question; instead, it goes "too high" outside the loop to repeat the `input originalNumber` task.

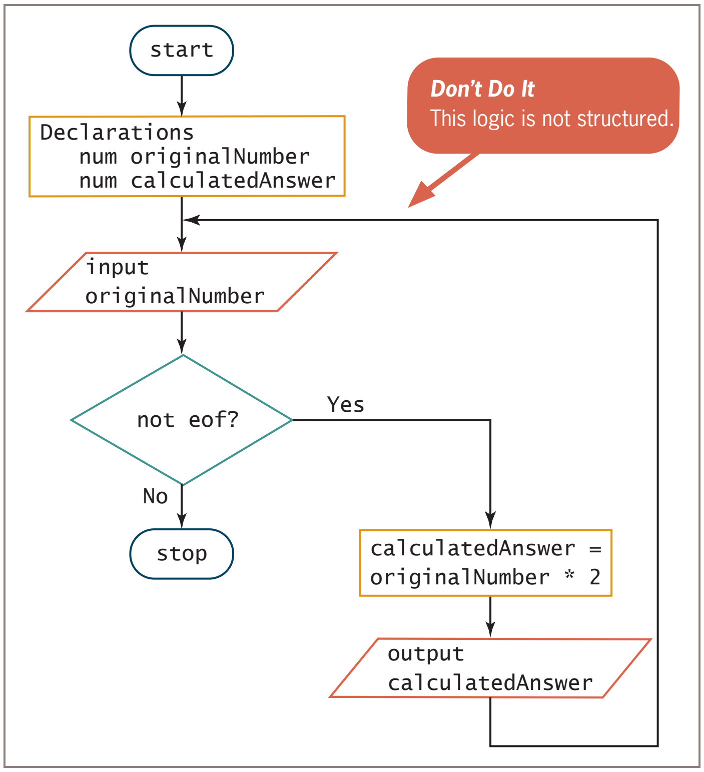


Figure 3-16 Functional but unstructured flowchart

How can the number-doubling problem be both structured and work as intended? Often, for a program to be structured, you must add something extra. In this case, it is a priming input step. A **priming input** or **priming read** is an added statement that gets the first input value in a program. For example, if a program will receive 100 data values as input, you input the first value in a statement that is separate from the other 99. You must do this to keep the program structured.

Consider the solution in Figure 3-17; it is structured *and* it does what it is supposed to do. It contains a shaded, additional `input originalNumber` statement. The program logic contains a sequence and a loop. The loop contains another sequence.

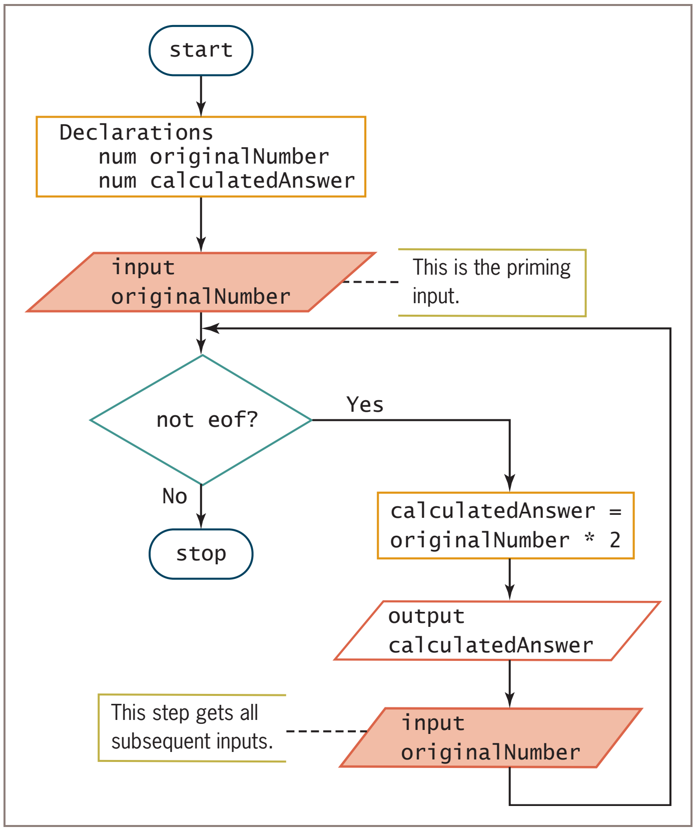


Figure 3-17 Functional, structured flowchart for the number-doubling problem

The additional `input originalNumber` step shown in Figure 3-17 is typical in structured programs. The first of the two input steps is the priming input. The term *priming* comes from the fact that the read is first, or *primary* (it gets the process going, as in "priming the pump"). The purpose of the priming input step is to control the upcoming loop that begins with the `not eof?` question. The last element within the structured loop gets the next, and all subsequent, input values. This is also typical in structured loops—the last step executed within the loop alters the condition tested in the question that begins the loop, which in this case is the `not eof?` question.

In Chapter 2, you learned that the group of preliminary tasks that sets the stage for the main work of a program is called the housekeeping section. The priming read is an example of a housekeeping task.

Figure 3-18 shows another way you might attempt to draw the logic for the number-doubling program. At first glance, the figure might seem to show an acceptable solution to the problem—it is structured, contains a single loop with a sequence of three steps within it, and appears to eliminate the need for the priming input statement. When the program starts, the `not eof?` question is asked, and if it is not the end of input data, then the program gets an input number, doubles it, and displays it. Then, if the `not eof?` condition remains true, the program gets another number, doubles it, and displays it. The program might continue while many numbers are input. The last time the `input originalNumber` statement executes, it encounters `eof`, but the program does not stop—instead, it calculates and displays a result one last time. Depending on the language you are using and on the type of input being used, you might receive an error message or you might output garbage. In either case, this last output is extraneous—no value should be doubled and output after the `eof` condition is encountered. As a general rule, an end-of-file test should always come immediately after an input statement because the end-of-file condition will be detected at input. Therefore, the best solution to the number-doubling problem remains the one shown in Figure 3-17—the solution containing the priming input statement.

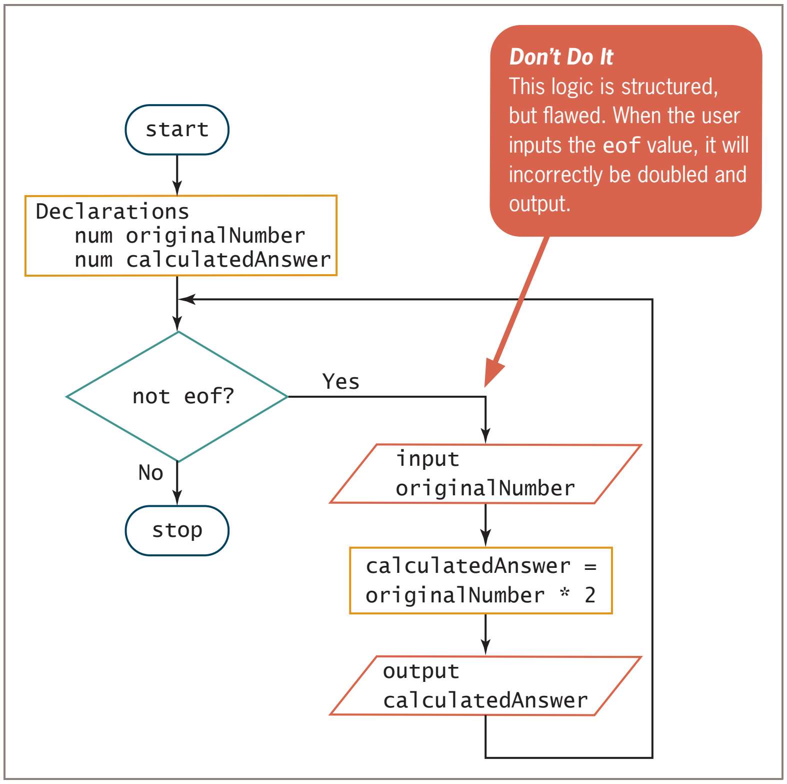

Don't Do It
This logic is structured, but flawed. When the user inputs the eof value, it will incorrectly be doubled and output.
start
Declarations
num originalNumber
num calculatedAnswer
not eof?
Yes
No
stop
input originalNumber
calculatedAnswer = originalNumber * 2
output calculatedAnswer

Figure 3-18 Structured but incorrect solution to the number-doubling problem

TWO TRUTHS & A LIE

Using a Priming Input to Structure a Program

1. A priming input is the statement that repeatedly gets all the data that is input in a program.
2. A structured program is sometimes longer than an unstructured one.
3. A program can be structured yet still be incorrect.

The false statement is #1. A priming input gets the first input.

Understanding the Reasons for Structure

At this point, you may very well be saying, "I liked the original number-doubling program back in Figure 3-12 just fine. I could follow it. Also, the first program had one less step in it, so it was less work. Who cares if a program is structured?"

Until you have some programming experience, it is difficult to appreciate the reasons for using only the three structures—sequence, selection, and loop. However, staying with these three structures is better for the following reasons:

- *Clarity*—The number-doubling program is small. As programs get bigger, they get more confusing if they are not structured.
- *Professionalism*—All other programmers (and programming teachers you might encounter) expect your programs to be structured. It is the way things are done professionally.
- *Efficiency*—Most newer computer languages support structure and use syntax that lets you deal efficiently with sequence, selection, and looping. Older languages, such as assembly languages, COBOL, and RPG, were developed before the principles of structured programming were discovered. However, even programs that use those older languages can be written in a structured form. Newer languages such as C#, C++, and Java enforce structure by their syntax.

In older languages, you could leave a selection or loop before it was complete by using a "go to" statement. The statement allowed the logic to "go to" any other part of the program whether it was within the same structure or not. Structured programming is sometimes called **goto-less programming**.

- *Maintenance*—You and other programmers will find it easier to modify and maintain structured programs as changes are required in the future.
- *Modularity*—Structured programs can be easily broken down into modules that can be assigned to any number of programmers. The routines are then pieced back together like modular furniture at each routine's single entry or exit point. Additionally, a module often can be used in multiple programs, saving development time in the new project.

TWO TRUTHS & A LIE

Understanding the Reasons for Structure

1. Structured programs are clearer than unstructured programs.
2. You and other programmers will find it easier to modify and maintain structured programs as changes are required in the future.
3. Structured programs are not easily divided into parts, making them less prone to error.

The false statement is #3. Structured programs can be easily broken down into modules that can be assigned to any number of programmers.

Recognizing Structure

When you are beginning to learn about structured program design, it is difficult to detect whether a flowchart of a program's logic is structured. For example, is the flowchart segment in Figure 3-19 structured?

Yes, it is. It has a sequence and a selection structure.

Is the flowchart segment in Figure 3-20 structured?

Yes, it is. It has a loop, and within the loop is a selection.

Is the flowchart segment in the upper-left corner of Figure 3-21 structured?

No, it is not built from the three basic structures. One way to straighten out an unstructured flowchart segment is to use the "spaghetti bowl" method; that is, picture the flowchart as a bowl of spaghetti that you must untangle. Imagine you can grab one piece of pasta at the top of the bowl and start pulling. As you "pull" each symbol out of the tangled mess, you can untangle the separate paths until the entire segment is structured.

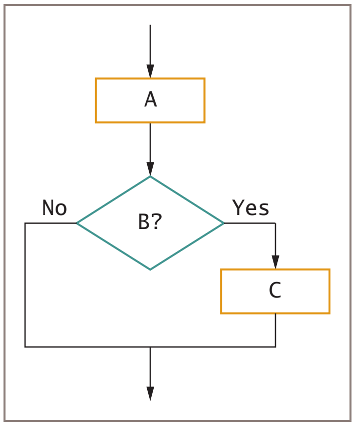


Figure 3-19 Example 1

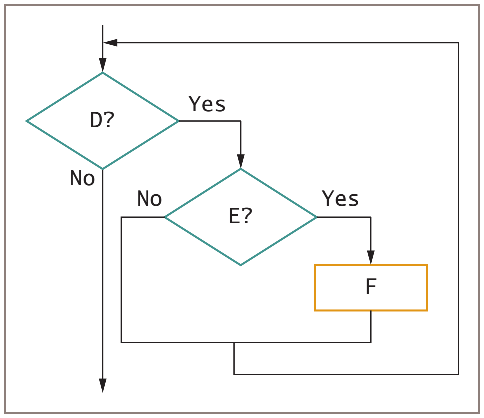


Figure 3-20 Example 2

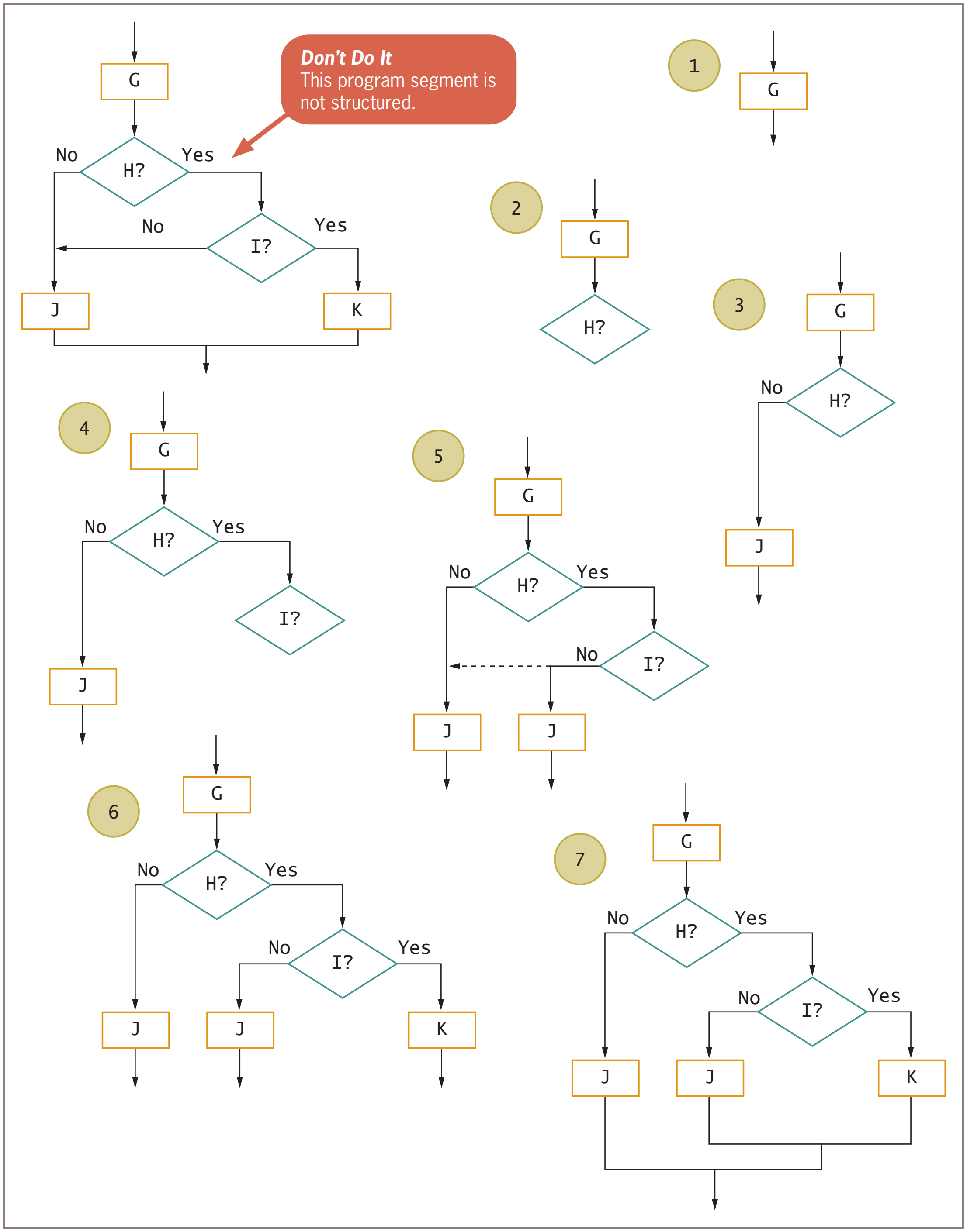


Figure 3-21 Example 3 and process to structure it

Look at the diagram in the upper-left corner of Figure 3-21. If you could start pulling the arrow at the top, you would encounter a procedure box labeled G. (See Figure 3-21, Step 1.) A single process like G is part of an acceptable structure—it constitutes at least the beginning of a sequence structure.

Imagine that you continue pulling symbols from the tangled segment. The next item in the flowchart is a question that tests a condition labeled H, as you can see in Figure 3-21, Step 2. At this point, you know the sequence that started with G has ended. Sequences never have decisions in them, so the sequence is finished; either a selection or a loop is beginning with question H. A loop must return to the loop-controlling question at some later point. You can see from the original logic that whether the answer to H is *Yes* or *No*, the logic never returns to H. Therefore, H begins a selection structure, not a loop structure.

To continue detangling the logic, you would pull up on the flowline that emerges from the left side (the *No* side) of Question H. You encounter J, as shown in Step 3 of Figure 3-21. When you continue beyond J, you reach the end of the flowchart.

Now you can turn your attention to the *Yes* side (the right side) of the condition tested in H. When you pull up on the right side, you encounter Question I. (See Step 4 of Figure 3-21.)

In the original version of the flowchart in Figure 3-21, follow the line on the left side of Question I. The line emerging from the left side of selection I is attached to J, which is outside the selection structure. You might say the I-controlled selection is becoming entangled with the H-controlled selection, so you must untangle the structures by repeating the step that is causing the tangle. (In this example, you repeat Step J to untangle it from the other usage of J.) Continue pulling on the flowline that emerges from J until you reach the end of the program segment, as shown in Step 5 of Figure 3-21.

Now pull on the right side of Question I. Process K pops up, as shown in Step 6 of Figure 3-21; then you reach the end.

At this point, the untangled flowchart has three loose ends. The loose ends of Question I can be brought together to form a selection structure; then the loose ends of Question H can be brought together to form another selection structure. The result is the flowchart shown in Step 7 of Figure 3-21. The entire flowchart segment is structured—it has a sequence followed by a selection inside a selection.

If you want to try structuring a more difficult example of an unstructured program, see Appendix E.

TWO TRUTHS & A LIE

Recognizing Structure

1. Some processes cannot be expressed in a structured format.
2. An unstructured flowchart can achieve correct outcomes.
3. Any unstructured flowchart can be “detangled” to become structured.

The false statement is #1. Any set of instructions can be expressed in a structured format.

Structuring and Modularizing Unstructured Logic

Recall the dog-washing process illustrated in Figure 3-1 at the beginning of this chapter. When you look at it now, you should recognize it as an unstructured process. Can this process be reconfigured to perform precisely the same tasks in a structured way? Of course!

Figure 3-22 demonstrates how you might approach structuring the dog-washing logic. Part 1 of the figure shows the beginning of the process. The first step, *Catch dog*, is a simple sequence. This step is followed by a question. When a question is encountered, the sequence is over, and either a loop or a selection starts. In this case, after the dog runs away, you must catch the dog and determine whether he runs away again, so a loop begins. To create a structured loop like the ones you have seen earlier in this chapter, you can repeat the *Catch dog* process and return immediately to the *Does dog run away?* question.

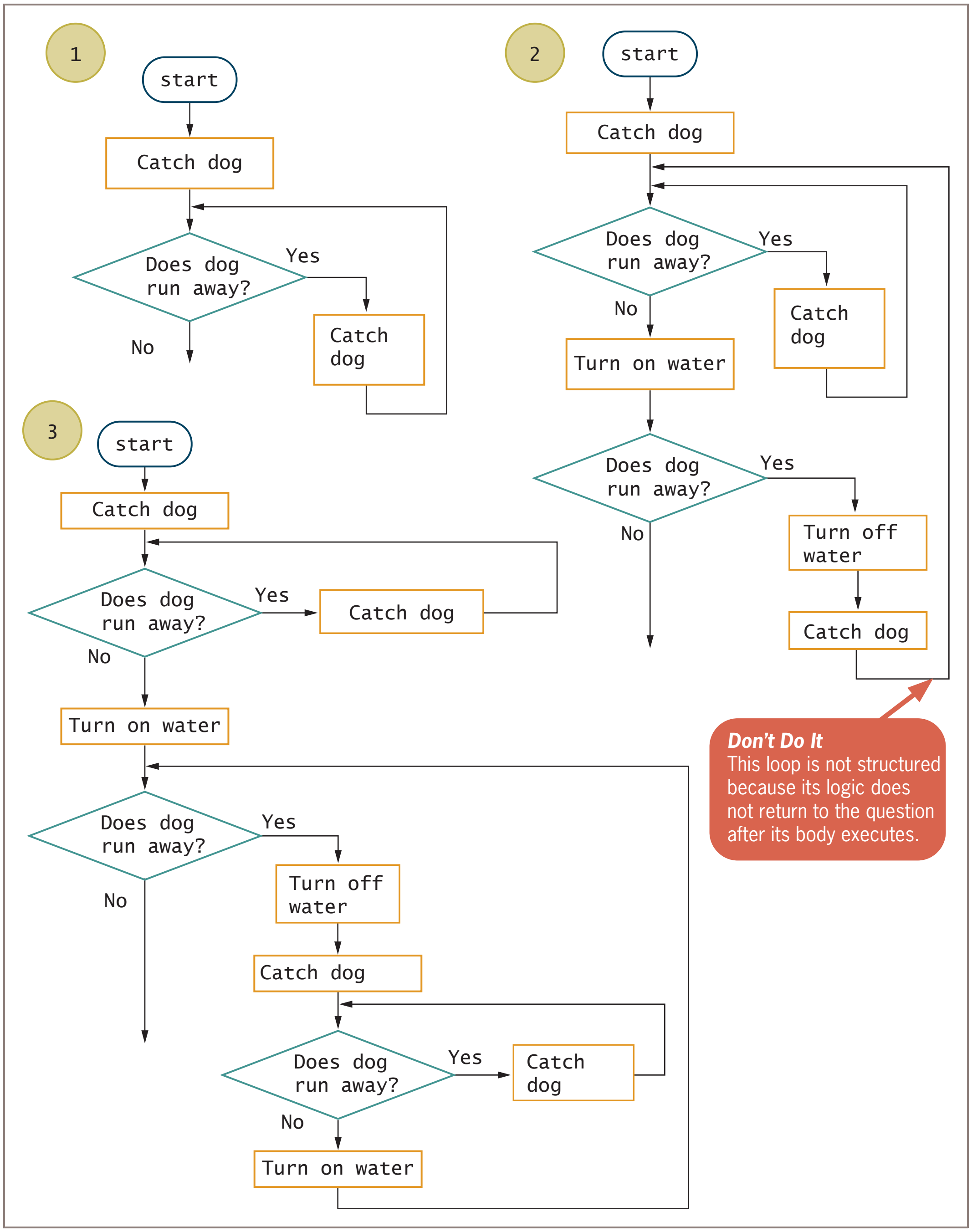
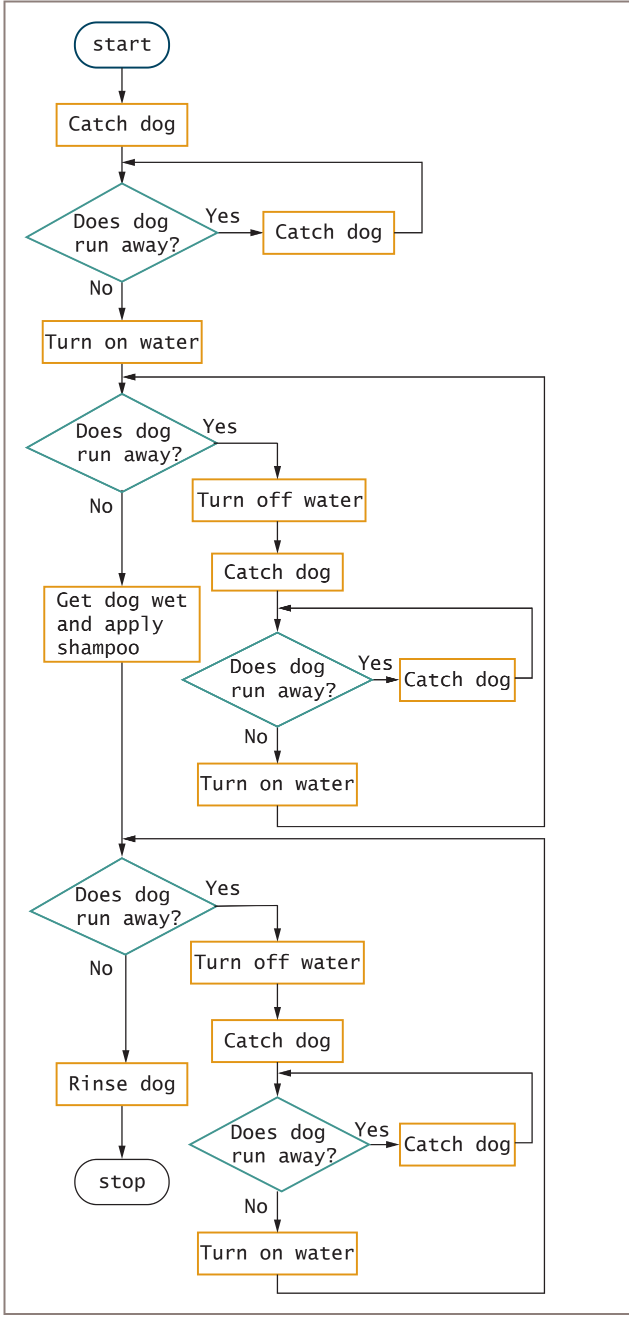


Figure 3-22 Steps to structure the dog-washing process

In the original flowchart in Figure 3-1, you turn on the water when the dog does not run away. This step is a simple sequence, so it can correctly be added after the loop. When the water is turned on, the original logic checks whether the dog runs away after this new development. This starts a loop. In the original flowchart, the lines cross, creating a tangle, so you repeat as many steps as necessary to detangle the lines. After you turn off the water and catch the dog, you encounter the question *Does dog have shampoo on?* Because the logic has not yet reached the shampooing step, there is no need to ask this question; the answer at this point always will be *No*. When one of the logical paths emerging from a question can never be traveled, you can eliminate the question. Part 2 of Figure 3-22 shows that if the dog runs away after you turn on the water, but before you've gotten the dog wet and shampooed him, you must turn the water off, catch the dog, and return to the step that asks whether the dog runs away.

The logic in Part 2 of Figure 3-22 is not structured because the second loop that begins with the question *Does dog run away?* does not immediately return to the loop-controlling question after its body executes. So, to make the loop structured, you can repeat the actions that occur before returning to the loop-controlling question. The flowchart segment in Part 3 of Figure 3-22 is structured; it contains a sequence, a loop, a sequence, and a final, larger loop. This last loop contains its own sequence, loop, and sequence.

After the dog is caught and the water is on, you wet and shampoo the dog. Then, according to the original flowchart in Figure 3-1, you once again check to see whether the dog has run away. If he has, you turn off the water and catch the dog. From this location in the logic, the answer to the *Does dog have shampoo on?* question will always be *Yes*; as before, there is no need to ask a question when there is only one possible answer. So, if the dog runs away, the last loop executes. You turn off the water, continue to catch the dog as he repeatedly escapes, and turn the water on. When the dog is caught at last, you rinse the dog and end the program. Figure 3-23 shows both the complete flowchart and pseudocode.

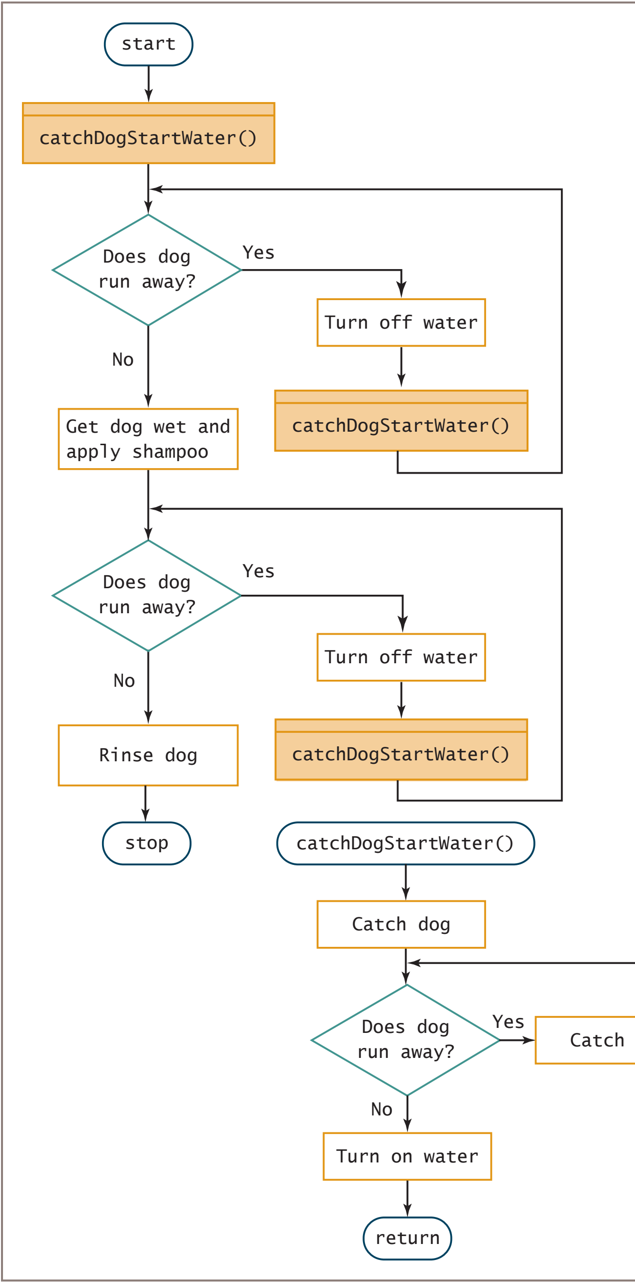


```
start
   Catch dog
   while dog runs away
      Catch dog
   endwhile
   Turn on water
   while dog runs away
      Turn off water
      Catch dog
      while dog runs away
         Catch dog
      endwhile
      Turn on water
   endwhile
   Get dog wet and apply shampoo
   while dog runs away
      Turn off water
      Catch dog
      while dog runs away
         Catch dog
      endwhile
      Turn on water
   endwhile
   Rinse dog
stop
```

Figure 3-23 Structured dog-washing flowchart and pseudocode

The flowchart in Figure 3-23 is complete and is structured. It contains alternating sequence and loop structures.

Figure 3-23 includes three places where the sequence-loop-sequence of catching the dog and turning the water on are repeated. If you wanted to, you could modularize the duplicate sections so that their instruction sets are written once and contained in their own module. Figure 3-24 shows a modularized version of the program; the three module calls are shaded.



```
start
   catchDogStartWater()
   while dog runs away
      Turn off water
      catchDogStartWater()
   endwhile
   Get dog wet and apply shampoo
   while dog runs away
      Turn off water
      catchDogStartWater()
   endwhile
   Rinse dog
stop

catchDogStartWater()
   Catch dog
   while dog runs away
      Catch dog
   endwhile
   Turn on water
return
```

Figure 3-24 Modularized version of the dog-washing program

No matter how complicated it is, any set of steps can always be reduced to combinations of the three basic sequence, selection, and loop structures. These structures can be nested and stacked in an infinite number of ways to describe the logic of any process and to create the logic for every computer program written in the past, present, or future.

For convenience, many programming languages allow two variations of the three basic structures. The `case` structure is a variation of the selection structure and the `do` loop is a variation of the `while` loop. You can learn about these two structures in Appendix F. Even though these extra structures can be used in most programming languages, all logical problems can be solved without them.

Watch the video *Structuring Unstructured Logic*.

TWO TRUTHS & A LIE

Structuring and Modularizing Unstructured Logic

1. When you encounter a question in a logical diagram, a sequence should be ending.
2. In a structured loop, the logic returns to the loop-controlling question after the loop body executes.
3. If a flowchart or pseudocode contains a question to which the answer never varies, you can eliminate the question.

The false statement is #1. When you encounter a question in a logical diagram, either a selection or a loop should start. Any structure might end before the question is encountered.

Chapter Summary

- Spaghetti code is the popular name for unstructured program statements that do not follow the rules of structured logic.
- Clearer programs can be constructed using only three basic structures: sequence, selection, and loop. These three structures can be combined in an infinite number of ways by stacking and nesting them. Each structure has one entry and one exit point; one structure can attach to another only at one of these points.
- A priming input is the statement that gets the first input value prior to starting a structured loop. The last step within the loop gets the next and all subsequent input values.

- Programmers use structured techniques to promote clarity, professionalism, efficiency, and modularity.
- One way to order an unstructured flowchart segment is to imagine it as a bowl of spaghetti that you must untangle.
- Any set of logical steps can be rewritten to conform to the three structures.

Key Terms

Spaghetti code is snarled, unstructured program logic.

Unstructured programs are programs that do *not* follow the rules of structured logic.

Structured programs are programs that do follow the rules of structured logic.

A **structure** is a basic unit of programming logic; each structure is a sequence, selection, or loop.

A **sequence structure** contains a series of steps executed in order. A sequence can contain any number of tasks, but there is no option to branch off and skip any of the tasks.

A **selection structure** or **decision structure** contains a question, and, depending on the answer, takes one of two courses of action before continuing with the next task.

An **end-structure statement** designates the end of a pseudocode structure.

An **`if-then-else`** is another name for a selection structure.

Dual-alternative `ifs` (or **dual-alternative selections**) define one action to be taken when the tested condition is true and another action to be taken when it is false.

Single-alternative `ifs` (or **single-alternative selections**) take action on just one branch of the decision.

The **null case** is the branch of a decision in which no action is taken.

A **loop structure** continues to repeat actions while a test condition remains true.

A **loop body** is the set of actions that occur within a loop.

Repetition and **iteration** are alternate names for a loop structure.

In a **`while`...do**, or more simply, a **`while` loop**, a process continues while some condition continues to be true.

Stacking structures is the act of attaching structures end to end.

Nesting structures is the act of placing a structure within another structure.

A **block** is a group of statements that executes as a single unit.

A **priming input** or **priming read** is the statement that reads the first input data record prior to starting a structured loop.

Goto-less programming is a name to describe structured programming, because structured programmers do not use a "go to" statement.

Review Questions

1. Snarled program logic is called ________ code.
   a. snake c. string
   b. spaghetti d. gnarly
2. The three structures of structured programming are ________.
   a. sequence, order, and process c. sequence, selection, and loop
   b. selection, loop, and iteration d. if, else, and then
3. A sequence structure can contain ________.
   a. any number of tasks c. no more than three tasks
   b. exactly three tasks d. only one task
4. Which of the following is *not* another term for a selection structure?
   a. decision structure c. dual-alternative `if` structure
   b. `if-then-else` structure d. loop structure
5. The structure in which you ask a question, and, depending on the answer, take some action and then ask the question again, can be called all of the following except a(n) ________.
   a. iteration c. repetition
   b. loop d. `if-then-else`
6. Placing a structure within another structure is called ________ the structures.
   a. stacking c. building
   b. untangling d. nesting
7. Attaching structures end to end is called ________.
   a. stacking c. building
   b. untangling d. nesting
8. The statement `if age >= 65 then seniorDiscount = "yes"` is an example of a ________.
   a. sequence c. dual-alternative selection
   b. loop d. single-alternative selection
9. The statement `while temperature remains below 60, leave the furnace on` is an example of a ________.
   a. sequence c. dual-alternative selection
   b. loop d. single-alternative selection

10. The statement `if age < 13 then movieTicket = 4.00 else movieTicket = 8.50` is an example of a ________.

    a. sequence
    b. loop
    c. dual-alternative selection
    d. single-alternative selection

11. Which of the following attributes do all three basic structures share?

    a. Their flowcharts all contain exactly three processing symbols.
    b. They all have one entry and one exit point.
    c. They all contain a decision.
    d. They all begin with a process.

12. Which is true of stacking structures?

    a. Two incidences of the same structure cannot be stacked adjacently.
    b. When you stack structures, you cannot nest them in the same program.
    c. Each structure has only one point where it can be stacked on top of another.
    d. When you stack structures, the top structure must be a sequence.

13. When you input data in a loop within a program, the input statement that precedes the loop ________.

    a. is the only part of the program allowed to be unstructured
    b. cannot result in `eof`
    c. is called a priming input
    d. executes hundreds or even thousands of times in most business programs

14. A group of statements that executes as a unit is a ________.

    a. block
    b. family
    c. chunk
    d. cohort

15. Which of the following is acceptable in a structured program?

    a. placing a sequence within the true half of a dual-alternative decision
    b. placing a decision within a loop
    c. placing a loop within one of the steps in a sequence
    d. All of these are acceptable.

16. In a selection structure, the structure-controlling question is ________.

    a. asked once at the beginning of the structure
    b. asked once at the end of the structure
    c. asked repeatedly until it is false
    d. asked repeatedly until it is true

17. When a loop executes, the structure-controlling question is ___________.
    a. asked exactly once
    b. never asked more than once
    c. asked either before or after the loop body executes
    d. asked only if it is true, and not asked if it is false
18. Which of the following is *not* a reason for enforcing structure rules in computer programs?
    a. Structured programs are clearer to understand than unstructured ones.
    b. Other professional programmers will expect programs to be structured.
    c. Structured programs usually are shorter than unstructured ones.
    d. Structured programs can be broken down into modules easily.
19. Which of the following is *not* a benefit of modularizing programs?
    a. Modular programs are easier to read and understand than nonmodular ones.
    b. If you use modules, you can ignore the rules of structure.
    c. Modular components are reusable in other programs.
    d. Multiple programmers can work on different modules at the same time.
20. Which of the following is true of structured logic?
    a. You can use structured logic with newer programming languages, such as Java and C#, but not with older ones.
    b. Any task can be described using some combination of the three structures.
    c. Structured programs require that you break the code into easy-to-handle modules that each contain no more than five actions.
    d. All of these are true.

Exercises

1. In Figure 3-10, the process of buying and planting flowers in the spring was shown using the same structures as the generic example in Figure 3-9. Use the same logical structure as in Figure 3-9 to create a flowchart or pseudocode that describes some other process you know.
2. Each of the flowchart segments in Figure 3-25 is unstructured. Redraw each segment so that it does the same thing but is structured.

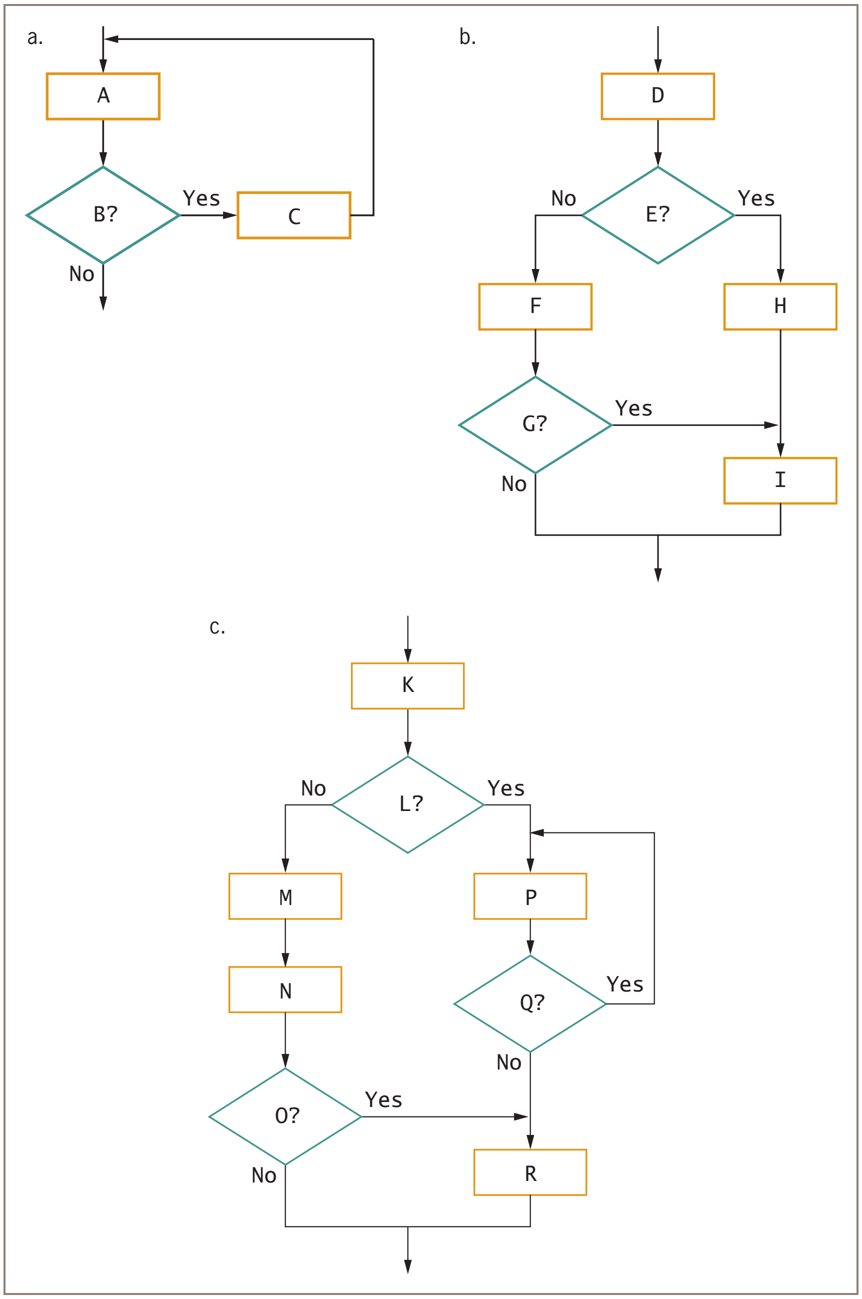


Figure 3-25 Flowcharts for Exercise 2 (continues)

(continued)

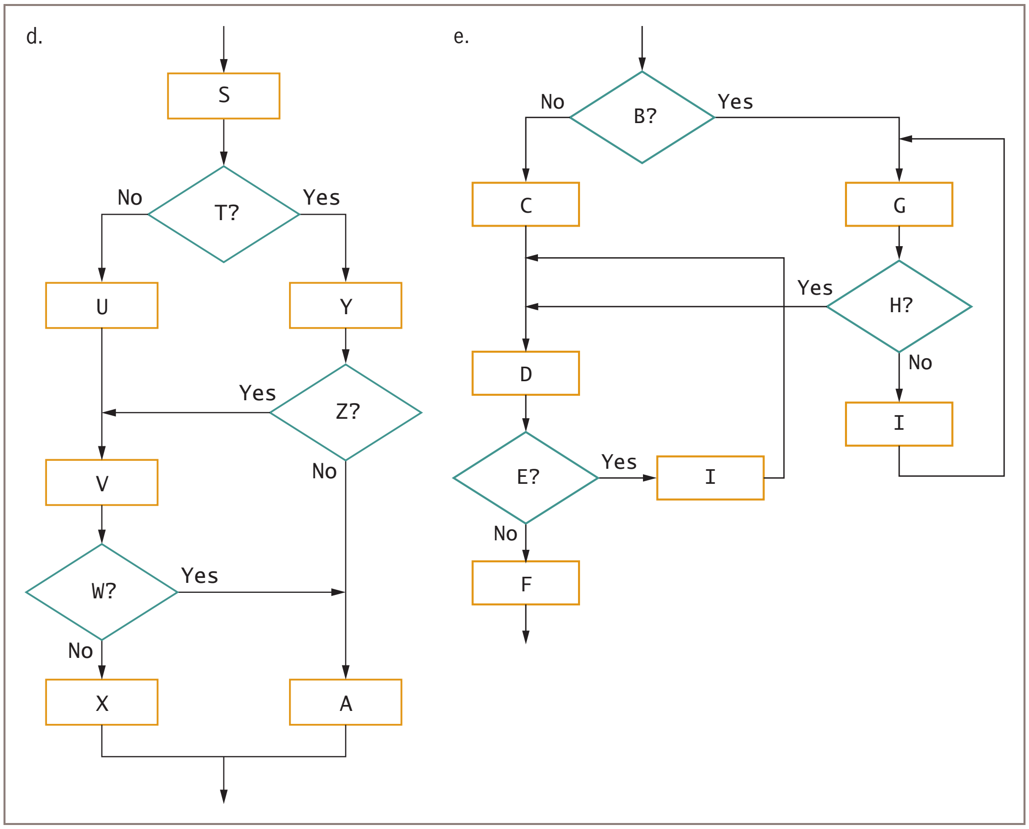


Figure 3-25 Flowcharts for Exercise 2

3. Write pseudocode for each example (a through e) in Exercise 2, making sure your pseudocode is structured but accomplishes the same tasks as the flowchart segment.

4. Assume that you have created a mechanical arm that can hold a pen. The arm can perform the following tasks:

    - Lower the pen to a piece of paper.
    - Raise the pen from the paper.
    - Move the pen 1 inch along a straight line. (If the pen is lowered, this action draws a 1-inch line from left to right; if the pen is raised, this action just repositions the pen 1 inch to the right.)
    - Turn 90 degrees to the right.
    - Draw a circle that is 1 inch in diameter.

    Draw a structured flowchart or write structured pseudocode describing the logic that would cause the arm to draw or write the following. Have a fellow student act as the mechanical arm and carry out your instructions. Don't reveal the desired outcome to your partner until the exercise is complete.

a. a 1-inch square

b. a 2-inch by 1-inch rectangle

c. a string of three beads

d. a short word (for example, *cat*)

e. a four-digit number

5. Assume that you have created a mechanical robot that can perform the following tasks:

- Stand up.
- Sit down.
- Turn left 90 degrees.
- Turn right 90 degrees.
- Take a step.

Additionally, the robot can determine the answer to one test condition:

- Am I touching something?

a. Place two chairs 20 feet apart, directly facing each other. Draw a structured flowchart or write pseudocode describing the logic that would allow the robot to start from a sitting position in one chair, cross the room, and end up sitting in the other chair. Have a fellow student act as the robot and carry out your instructions.

b. Draw a structured flowchart or write pseudocode describing the logic that would allow the robot to start from a sitting position in one chair, stand up and circle the chair, cross the room, circle the other chair, return to the first chair, and sit. Have a fellow student act as the robot and carry out your instructions.

6. Draw a structured flowchart or write pseudocode that describes the process of guessing a number between 1 and 100. After each guess, the player is told that the guess is too high or too low. The process continues until the player guesses the correct number. Pick a number and have a fellow student try to guess it following your instructions.

7. Looking up a word in a dictionary can be a complicated process. For example, assume that you want to look up *logic*. You might open the dictionary to a random page and see *juice*. You know this word comes alphabetically before *logic*, so you flip forward and see *lamb*. That is still not far enough, so you flip forward and see *monkey*. You have gone too far, so you flip back, and so on. Draw a structured flowchart or write pseudocode that describes the process of looking up a word in a

dictionary. Pick a word at random and have a fellow student attempt to carry out your instructions.

8. Draw a structured flowchart or write structured pseudocode describing how to find your classroom from the front entrance of the school building. Include at least two decisions and two loops.

9. Draw a structured flowchart or write structured pseudocode describing how to tidy up an apartment. Include at least two decisions and two loops.

10. Draw a structured flowchart or write structured pseudocode describing how to wrap a present. Include at least two decisions and two loops.

11. Draw a structured flowchart or write structured pseudocode describing the steps a grocery store clerk should follow to check out a customer. Include at least two decisions and two loops.

Find the Bugs

12. Your downloadable files for Chapter 3 include DEBUG03-01.txt, DEBUG03-02.txt, and DEBUG03-03.txt. Each file starts with some comments that describe the problem. Comments are lines that begin with two slashes (//). Following the comments, each file contains pseudocode that has one or more bugs you must find and correct.

Game Zone

13. Choose a simple children's game and describe its logic, using a structured flowchart or pseudocode. For example, you might try to explain Rock, Paper, Scissors; Musical Chairs; Duck, Duck, Goose; the card game War; or the elimination game Eenie, Meenie, Minie, Moe.

14. Choose a television game show such as *Deal or No Deal* or *Jeopardy!* and describe its rules using a structured flowchart or pseudocode.

15. Choose a sport such as baseball or football and describe the actions in one limited play period (such as an at-bat in baseball or a possession in football) using a structured flowchart or pseudocode.

Up for Discussion

16. Find more information about one of the following people and explain why he or she is important to structured programming: Edsger Dijkstra, Corrado Bohm, Giuseppe Jacopini, and Grace Hopper.
17. Computer programs can contain structures within structures and stacked structures, creating very large programs. Computers also can perform millions of arithmetic calculations in an hour. How can we possibly know the results are correct?
18. Develop a checklist of rules you can use to help you determine whether a flowchart or pseudocode segment is structured.