4

Steps in Building a Routine

Contents

Related Topics

Characteristics of high-quality routines: Chapter 5
High-level design: Chapter 7
Commenting style: Chapter 19
Prerequisites to construction: Chapter 3

THIS CHAPTER EXAMINES THE STEPS you typically take to create a routine. Although you could view the whole book as an extended description of how to create a routine, this chapter puts the steps in context. The chapter focuses on programming in the small—on the specific steps for building an individual routine that are critical on projects of all sizes. The chapter also describes the PDL-to-code process, which reduces the work required during design and documentation and improves the quality of both.

If you're an expert programmer, you might just skim this chapter. Look at the summary of steps and review the PDL-to-code process. Few programmers exploit the full power of the process, and it offers many benefits.

4.1 Summary of Steps in Building a Routine

The many low-level details that go into building a routine don't need to be handled in any particular order, but the major activities—designing the routine, checking the design, coding the routine, and checking the code—are done in the order shown in Figure 4-1.

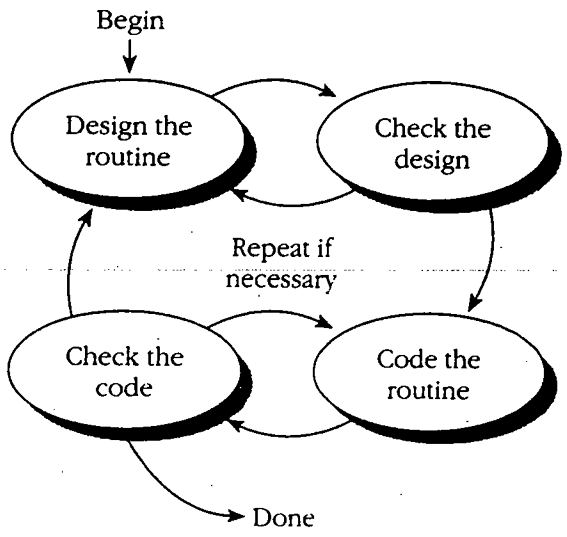

Figure 4-1. *These are the major activities that go into constructing a routine. They're usually performed in the order shown.*

4.2 PDL for Pros

FURTHER READING
If you want to read the original article on PDL, look for "PDL—A Tool for Software Design" (Caine and Gordon 1975). The description of PDL in this section differs substantially from the original conception of the language.

PDL (program design language) was originally developed by the company Caine, Farber & Gordon and has been modified substantially since they published their initial paper on it in 1975. Because PDL resembles English, it's natural to assume that any English-like description that collects your thoughts will have roughly the same effect as any other. In practice, you'll find that some styles of PDL are more useful than others. Here are guidelines for using PDL effectively:

- Use English-like statements that precisely describe specific operations.
- Avoid syntactic elements from the target programming language. PDL allows you to design at a slightly higher level than the code itself. When you use programming-language constructs, you sink to a lower level, eliminating the main benefit of design at a higher level, and you saddle yourself with unnecessary syntactic restrictions.

CROSS-REFERENCE For details on commenting at the level of intent, see "Kinds of Comments" in Section 19.4.

- Write PDL at the level of intent. Describe the meaning of the approach rather than how the the approach will be implemented in the target language.
- Write PDL at a low enough level that generating code from it will be nearly automatic. If the PDL is at too high a level, it can gloss over problematic details in the code. Refine the PDL in more and more detail until it seems as if it would be easier to simply write the code.

Once the PDL is written, you build the code around it and the PDL turns into programming-language comments. This eliminates most commenting effort. If the PDL follows the guidelines, the comments will be complete and meaningful.

Here's an example of a design in PDL that violates virtually all the principles just described:

CODING HORROR

Example of Bad PDL

```
increment resource number by 1
allocate a dlg struct using malloc
if malloc() returns NULL then return 1
invoke OSrsrc_init to initialize a resource for the operating system
*hRsrcPtr = resource number
return 0
```

What is the intent of this block of PDL? Because it's poorly written, it's hard to tell. This so-called PDL is bad because it includes coding details such as **hRsrcPtr* in specific C-language pointer notation, and *malloc()*, a specific C-language function. This PDL block focuses on how the code will be written rather than on the meaning of the design. It gets into coding details—whether the routine returns a *1* or a *0*. If you think about this PDL from the standpoint of whether it will turn into good comments, you'll begin to understand that it's not much help.

Here's a design for the same operation in a much-improved PDL:

Example of Good PDL

```
Keep track of current number of resources in use
If another resource is available
   Allocate a dialog box structure
   If a dialog box structure could be allocated
      Note that one more resource is in use
      Initialize the resource
      Store the resource number at the location provided by the caller
   Endif
Endif
Return TRUE if a new resource was created; else return FALSE
```

This PDL is better than the first because it's written entirely in English; it doesn't use any syntactic elements of the target language. In the first example, the PDL could have been implemented only in C. In the second example, the PDL doesn't restrict the choice of languages. The second block of PDL is also written at the level of intent. What does the second block of PDL mean? It is probably easier for you to understand than the first block.

Even though it's written in clear English, the second block of PDL is precise and detailed enough that it can easily be used as a basis for source-language code. When the PDL statements are converted to comments, they'll be a good explanation of the code's intent.

Here are the benefits you can expect from using this style of PDL:

- PDL makes reviews easier. You can review detailed designs without examining source code. PDL makes low-level design reviews easier and reduces the need to review the code itself.
- PDL supports the idea of iterative refinement. You start with a high-level architecture, refine the architecture to PDL, and then refine the PDL to source code. This successive refinement in small steps allows you to check your design as you drive it to lower levels of detail. The result is that you catch high-level errors at the highest level, mid-level errors at the middle level, and low-level errors at the lowest level—before any of them becomes a problem or contaminates work at more detailed levels.
- PDL makes changes easier. A few lines of PDL are easier to change than a page of code. Would you rather change a line on a blueprint or rip out a wall and nail in the two-by-fours somewhere else? The effects aren't as physically dramatic in software, but the principle of changing the product when it's most malleable is the same. One of the keys to the success of a project is to catch errors at the least-value stage, the stage at which the least has been invested. Much less has been invested at the PDL stage than after full coding, testing, and debugging, so it makes economic sense to catch the errors early.

FURTHER READING
For more information on the advantages of making changes at the least-value stage, see Andy Grove's *High Output Management* (Grove 1983).

- PDL minimizes commenting effort. In the typical coding scenario, you write the code and add comments afterward. In the PDL-to-code approach, the PDL statements become the comments, so it actually takes more work to remove the comments than to leave them in.
- PDL is easier to maintain than other forms of design documentation. With other approaches, design is separated from the code, and when one changes, the two fall out of agreement. With the PDL-to-code process, the PDL statements become comments in the code. As long as the inline comments are maintained, the PDL's documentation of the design will be accurate.

KEY POINT

As a tool for detailed design, PDL is hard to beat. Programmers tend to prefer PDL to flawcharts. (Programmers tend to prefer everything to flawcharts.) One survey found that programmers prefer PDL for the way it eases implementation in a programming language, for its ability to help them detect insufficiently detailed designs, and for the ease of documentation and ease of modification it provides (Ramsey, Atwood, and Van Doren 1983). PDL isn't the only tool for detailed design, but PDL and the PDL-to-code process are useful tools to have in your programmer's toolbox. Try them. The next few sections show you how.

4.3 Design the Routine

CROSS-REFERENCE For details on other aspects of design, see Chapters 5 through 7.

The first step in constructing a routine is to design it. Suppose that you want to write a routine to output an error message depending on an error code, and suppose that you call the routine *RecordErrorMessage()*. Here's the spec for *RecordErrorMessage()*:

> *RecordErrorMessage()* takes an error code as an input argument and outputs an error message corresponding to the code. It's responsible for handling invalid codes. If the program is operating interactively, *RecordErrorMessage()* prints the message to the user. If it's operating in batch mode, *RecordErrorMessage()* logs the message to a message file. After outputting the message, *RecordErrorMessage()* returns a status variable indicating whether it succeeded or failed.

The rest of the chapter uses this routine as a running example. The rest of this section describes how to design the routine. The activities involved in designing a routine are shown in Figure 4-2 on the next page.

CROSS-REFERENCE For details on checking prerequisites, see Chapter 3, "Prerequisites to Construction."

Check the prerequisites. Before doing any work on the routine itself, check to see that the job of the routine is well defined and fits cleanly into the overall architecture. Check to be sure that the routine is actually called for, at the very least indirectly, by the project's requirements.

Define the problem the routine will solve. State the problem the routine will solve in enough detail to allow creation of the routine. If the architecture is sufficiently detailed, the job might already be done. The architecture should at least indicate the following:

CROSS-REFERENCE For details on information hiding, see Section 6.2, "Information Hiding."

- The information the routine will hide
- Inputs to the routine
- Outputs from the routine, including any global variables affected
- How the routine will handle errors

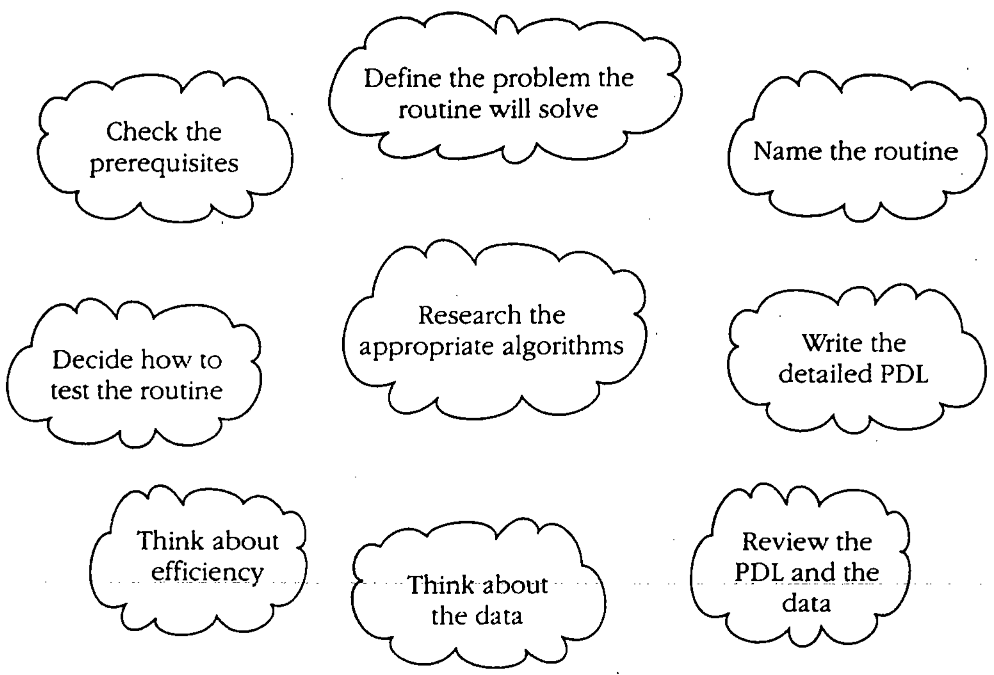
Check the prerequisites
Define the problem the routine will solve
Name the routine
Decide how to test the routine
Research the appropriate algorithms
Write the detailed PDL
Think about efficiency
Think about the data
Review the PDL and the data

Figure 4-2. *You'll perform all of these steps as you design a routine but not necessarily in any particular order.*

Here's how these concerns are addressed in the *RecordErrorMessage( )* example. The routine hides two facts: the error messages and the current processing method (interactive or batch). The input to the routine is an error code. Two kinds of output are called for: The first is the error message; the second is the status that *RecordErrorMessage( )* returns to the calling routine.

Resolution of how the routine will handle errors doesn't follow directly from the problem statement. Suppose, for sake of the example, that the program's convention is to report errors at the point of detection. In that case, the routine must report any errors that it is the first to detect and will assume that others have already been reported. According to the spec, if it detects an error, it must also set the status-return variable to *Failure*.

CROSS-REFERENCE For details on naming routines, see Section 5.2, "Good Routine Names."

Name the routine. Naming the routine might seem trivial, but good routine names are one sign of a superior program, and they're not easy to come up with. In general, a routine should have a clear, unambiguous name. If you have trouble creating a good name, that usually indicates that the purpose of the routine isn't clear. A vague, wishy-washy name is like a politician on the campaign trail. It sounds as if it's saying something, but when you take a hard look, you can't figure out what it means. If you can make the name clearer, do so. If the wishy-washy name results from a wishy-washy architecture, pay attention to the warning sign. Back up and improve the architecture.

In the example, *RecordErrorMessage( )* is unambiguous. It is a good name.

Decide how to test the routine. As you're writing the routine, think about how you can test it. This is useful for you when you do unit testing and for the tester who tests your routine independently.

In the example, the input is simple, so you might plan to test *RecordErrorMessage( )* with all valid error codes and a variety of invalid codes.

Think about efficiency. Depending on your situation, you can address efficiency in one of two ways. In the first situation, in the vast majority of systems, performance isn't critical. In such a case, see that the routine is well modularized and readable so that you can improve it later if you need to. If you have good modularization, you can replace a slow, high-level language routine with a better algorithm or a fast assembler routine and you won't affect any other routines.

CROSS-REFERENCE
For details on efficiency, see Chapter 28, "Code-Tuning Strategies," and Chapter 29, "Code-Tuning Techniques."

In the second situation, in the minority of systems, performance is critical and the architecture should indicate how much memory the routine is allowed to use and how fast it should be. Design your routine so that it will meet its space and speed goals. If either space or speed seems more critical, design so that you trade space for speed or vice versa. It's acceptable during initial construction of the routine to tune it enough to meet its space and speed budgets.

Aside from taking the approaches suggested for these two general situations, it's usually a waste of effort to work on efficiency at the level of individual routines. The big optimizations come from refining the top-level design, not the individual routines. You generally use micro-optimizations only when the top-level design turns out to be inadequate in some respect, and you won't know that until the whole program is done. Don't waste time scraping for incremental improvements until you know they're needed.

Research the algorithms and data structures. The single biggest way to improve both the quality of your code and your productivity is to reuse good code. Many algorithms have already been invented, tested, discussed in the trade literature, reviewed, and improved. Rather than spending your time inventing something when someone has already written a Ph.D. dissertation on it, take a few minutes to look through an algorithms book to see what's already available. If you use a predefined algorithm, be sure to adapt it correctly to your programming language.

Write the PDL. You might not have much in writing after you finish the preceding steps. The main purpose of the steps is to establish a mental orientation that's useful when you actually write the routine.

CROSS-REFERENCE
This discussion assumes that good design techniques are used to create the PDL version of the routine. For details on design, see Chapter 7, "High-Level Design in Construction."

With the preliminary steps completed, you can begin to write the routine as high-level PDL. Go ahead and use your programming editor or your integrated environment to write the PDL—the PDL will be used shortly as the basis for programming-language code.

Start with the general and work toward something more specific. The most general part of a routine is a header comment describing what the routine is supposed to do, so first write a concise statement of the purpose of the routine. Writing the statement will help you clarify your understanding of the routine. Trouble in writing the general comment is a warning that you need to understand the routine's role in the program better. In general, if it's hard to summarize the routine's role, you should probably assume that something is wrong. Here's an example of a concise header comment describing a routine:

Example of a Header Comment for a Routine

```
This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a variable indicating success or failure.
```

After you've written the general comment, fill in high-level PDL for the routine. Here's the PDL for the example:

Example of PDL for a Routine

```
This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a variable indicating success or failure.

set the default status
look up the message based on the error code
if the error code is valid
   determine the processing method
   if doing interactive processing
      print the error message interactively and declare success
   else doing batch processing
      if the batch message file opens properly
         log the error message to the batch file,
         close the file, and declare success
else the message code is not valid
   notify the user that an internal error has been detected
```

Note that the PDL is written at a fairly high level. It certainly isn't written in a programming language. It expresses in precise English what the routine needs to do.

CROSS-REFERENCE For details on effective use of data, see Chapters 8 through 12.

Think about the data. You can design the routine's data at several different points in the process. In the example, the data is simple and data manipulation isn't a prominent part of the routine. If data manipulation is a prominent part of the routine, it's worthwhile to think about the major pieces of data before you think about the routine's logic. Definitions of key data structures are useful to have when you design the logic of a routine.

CROSS-REFERENCE For details on review techniques, see Chapter 24, "Reviews."

Check the PDL. Once you've written the PDL and designed the data, take a minute to review the PDL you've written. Back away from it, and think about how you would explain it to someone else.

Ask someone else to look at it or listen to you explain it. You might think that it's silly to have someone look at 11 lines of PDL, but you'll be surprised. PDL can make your assumptions and high-level mistakes more obvious than programming-language code does. People are also more willing to review a few lines of PDL than they are to review 35 lines of C or Pascal.

Make sure you have an easy and comfortable understanding of what the routine does and how it does it. If you don't understand it conceptually, at the PDL level, what chance do you have of understanding it at the programming-language level? And if you don't understand it, who else will?

Iterate. Try as many ideas as you can in PDL before you start coding. Once you start coding, you get emotionally involved with your code and it becomes harder to throw away a bad design and start over.

The general idea is to refine the routine as PDL until the PDL statements become simple enough that you can fill in code below each statement and leave the original PDL as documentation. Some of the PDL from your first attempt might be high-level enough that you need to decompose it further. Be sure you do decompose it further. If you're not sure how to code something, keep working with the PDL until you are sure. Keep refining and decomposing the PDL until it seems like a waste of time to write it instead of the actual code.

4.4 Code the Routine

Once you've designed the routine, implement it. You can perform implementation steps in a nearly standard order, but feel free to vary them as you need to. Figure 4-3 on the next page shows the steps in implementing a routine.

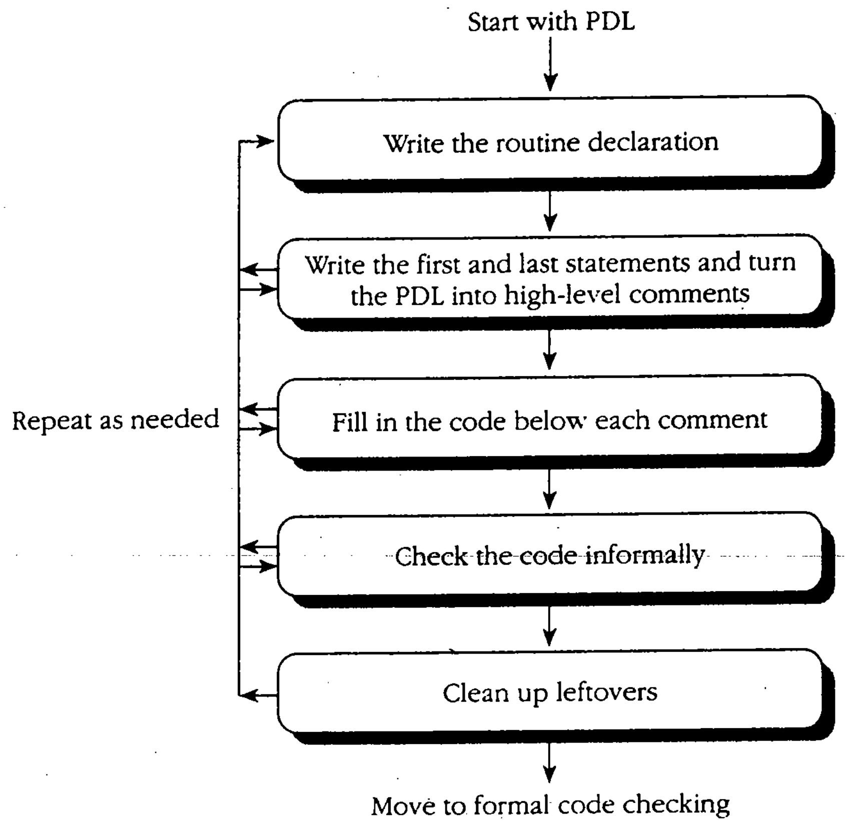
Start with PDL
Write the routine declaration
Write the first and last statements and turn the PDL into high-level comments
Repeat as needed
Fill in the code below each comment
Check the code informally
Clean up leftovers
Move to formal code checking

Figure 4-3. *The steps taken to implement a routine.*

Write the routine declaration. Write the routine interface statement—the procedure or function declaration in Pascal, function declaration in C, subroutine definition in Fortran, or whatever your language calls for. Turn the original header comment into a programming-language comment. Leave it in position above the PDL you've already written. Here are the example routine's interface statement and header in Pascal:

Pascal Example of a Routine Interface and Header Added to PDL

Here's the interface statement.

```
procedure RecordErrorMessage
   (
   ErrorCode: ErrorCode_t;
   var Status: Status_t
   );
```

Here's the header comment that's been turned into a Pascal-style comment.

```
{ This routine outputs an error message based on an error code
  supplied by the calling routine. The way it outputs the message
  depends on the current processing state, which it retrieves
  on its own. It returns a variable indicating success or failure. }
```

(continued)

```
set the default status
look up the message based on the error code
if the error code is valid
   determine the processing method
   if doing interactive processing
      print the error message interactively and declare success
   else doing batch processing
      if the batch message file opens properly
         log the error message to the batch file,
         close the file, and declare success
else the message code is not valid
   notify the user that an internal error has been detected
```

This is a good time to make notes about any interface assumptions. In this case, the interface variables *ErrorCode* and *Status* are straightforward and typed for their specific purposes; they don't contain any hidden assumptions.

Turn the PDL into high-level comments. Keep the ball rolling by writing the first and last statements—*begin* and *end* in Pascal, *{* and *}* in C. Then turn the PDL into comments. Here's how it would look in the example:

Pascal Example of Writing the First and Last Statements Around PDL

The PDL statements — from here down have been turned into Pascal comments.

You can also document the end of the routine. —

```
procedure RecordErrorMessage
   (
   ErrorCode: ErrorCode_t;
   var Status: Status_t
   );

{ This routine outputs an error message based on an error code
  supplied by the calling routine. The way it outputs the message
  depends on the current processing state, which it retrieves
  on its own. It returns a variable indicating success or failure. }

begin
   { set the default status }
   { look up the message based on the error code }
   { if the error code is valid }
      { determine the processing method }
      { if doing interactive processing }
         { print the error message interactively and declare success }
      { else doing batch processing }
         { if the batch message file opens properly }
            { log the error message to the batch file,
              close the file, and declare success }
   { else the message code is not valid }
      { notify the user that an internal error has been detected }
end; { RecordErrorMessage() }
```

At this point, the character of the routine is evident. The design work is complete, and you can sense how the routine works even without seeing any code. You should feel that converting the PDL to programming-language code will be mechanical, natural, and easy. If you don't, continue designing in PDL until the design feels solid.

Fill in the code below each comment. Fill in the code below each line of PDL comment. The process is a lot like writing a term paper. First you write an outline, and then you write a paragraph for each point in the outline. Each PDL comment describes a block or paragraph of code. Like the lengths of literary paragraphs, the lengths of code paragraphs vary according to the thought being expressed, and the quality of the paragraphs depends on the vividness and focus of the thoughts in them.

In the example, the first two PDL comments give rise to two lines of code:

Pascal Example of Expressing PDL Comments as Code

```
procedure RecordErrorMessage
   (
   ErrorCode: ErrorCode_t;
   var Status: Status_t
   );

{ This routine outputs an error message based on an error code
  supplied by the calling routine. The way it outputs the message
  depends on the current processing state, which it retrieves
  on its own. It returns a variable indicating success or failure. }

begin
   { set the default status }
   Status := Failure;

   { look up the message based on the error code }
   LookupErrorMessage( ErrorCode, ErrorMessage );

   { if the error code is valid }
      { determine the processing method }
      { if doing interactive processing }
         { print the error message interactively and declare success }
      { else doing batch processing }
         { if the batch message file opens properly }
            { log the error message to the batch file,
              close the file, and declare success }
   { else the message code is not valid }
      { notify the user that an internal error has been detected }
end; { RecordErrorMessage() }
```

Here's the code that's been filled in.

Here's the new variable ErrorMessage.

This is a start on the code. The variable *ErrorMessage* is used, so it needs to be declared. If you were commenting after the fact, two lines of comments for two lines of code would nearly always be overkill. In this approach, however, it's the semantic content of the comments that's important, not how many lines of code they comment. The comments are already there, and they explain the intent of the code, so leave them in.

The code needs variable declarations, and the code below each of the remaining comments needs to be filled in. Here's the completed routine:

Pascal Example of a Complete Routine Created with the PDL-to-Code Process

Here's where the variables were declared as they were needed.

The code for each comment has been filled in from here down.

```
procedure RecordErrorMessage
   (
   ErrorCode: ErrorCode_t;
   var Status: Status_t
   );

{ This routine outputs an error message based on an error code
  supplied by the calling routine. The way it outputs the message
  depends on the current processing state, which it retrieves
  on its own. It returns a variable indicating success or failure. }

var
   ProcessingMethod: ProcessingMethod_t;
   ErrorMessage:     Message_t;
   FileStatus:       Status_t;

begin
   { set the default status }
   Status := Failure;

   { look up the message based on the error code }
   LookupErrorMessage( ErrorCode, ErrorMessage );

   { if the error code is valid }
   if ( ErrorMessage.ValidCode ) then begin

      { determine the processing method }
      ProcessingMethod := CurrentProcessingMethod;

      { if doing interactive processing }
      if ( ProcessingMethod = Interactive ) then begin

         { print the error message interactively and declare success }
         PrintInteractiveMessage( ErrorMessage.Text );
         Status := Success
      end
```

(continued)

```
      { else doing batch processing }
      else if ( ProcessingMethod = Batch ) then begin

         { if the batch message file opens properly }
         FileStatus := OpenMessageFile;
         if ( FileStatus = Success ) then begin

            { log the error message to the batch file, close the file,
              and declare success }
            LogBatchMessage( ErrorMessage.Text );
            CloseMessageFile;
            Status := Success
         end { if }
      end { else }
   end

   { else the message code is not valid }
   else begin

      { notify the user that an internal error has been detected }
      PrintInteractiveMessage( 'Internal Error: Invalid error code',
         ' in RecordErrorMessage()' )
   end

end; { RecordErrorMessage() }
```

Each comment has given rise to one or more lines of code. Each block of code forms a complete thought based on the comment. The comments have been retained to provide a higher-level explanation of the code. All the variables that have been used have been declared at the top of the routine.

Now look again at the spec on page 57 and the initial PDL on page 60. The original 5-sentence spec expanded to 12 lines of PDL, which in turn expanded into a page-long routine. Even though the spec was detailed, creation of the routine required substantial design work in PDL and code. That low-level design is one reason why "coding" is a nontrivial task and why the subject of this book is important.

Check the code informally. Mentally test each block of code as you fill it in below its comment. Try to think of what it would take to break that block, and then prove to yourself that it won't happen.

Once you have an implementation of the routine, stop to check it for mistakes. You should already have checked it when you wrote it in PDL, but sometimes an important problem doesn't appear until the routine is implemented.

CROSS-REFERENCE For details on checking for errors in architecture and requirements, see Chapter 3, "Prerequisites to Construction."

A problem might not appear until coding for several reasons. An error in the PDL might become more apparent in the detailed implementation logic. A design that looks elegant in PDL might become clumsy in the implementation language. Working with the detailed implementation might disclose an error in the architecture or the requirements analysis. Finally, the code might have an old-fashioned, mongrel coding error—nobody's perfect! For all these reasons, review the code before you move on.

Clean up the leftovers. When you've finished checking your code for problems, check it for the general characteristics described throughout this book. You can take several cleanup steps to make sure that the routine's quality is up to your standards:

- Check the routine's interface. Make sure that all input and output data is accounted for and that all parameters are used. For more details, see Section 5.7, "How to Use Routine Parameters."
- Check for general design quality. Make sure the routine does one thing and does it well, that it's loosely coupled to other routines, and that it's designed defensively. For details, see Chapter 5, "Characteristics of High-Quality Routines."
- Check the routine's data. Check for inaccurate variable names, unused data, undeclared data, and so on. For details, see the chapters on using data, Chapters 8 through 12.
- Check the routine's control structures. Check for off-by-one errors, infinite loops, and improper nesting. For details, see the chapters on using control structures, Chapters 13 through 17.
- Check the routine's layout. Make sure you've used white space to clarify the logical structure of the routine, expressions, and parameter lists. For details, see Chapter 18, "Layout and Style."
- Check the routine's documentation. Make sure the PDL that was translated into comments is still accurate. Check for algorithm descriptions, for documentation on interface assumptions and nonobvious dependencies, for justification of unclear coding practices, and so on. For details, see Chapter 19, "Self-Documenting Code."

Repeat steps as needed. If the quality of the routine is poor, back up to the PDL. High-quality programming is an iterative process, so don't hesitate to loop through the design and implementation activities again.

4.5 Check the Code Formally

KEY POINT

After designing and implementing the routine, the third big step in constructing it is checking to be sure that what you've constructed is correct. Didn't the informal checking and cleaning up of leftovers verify the correctness of the code? Yes, partially, but not completely, and any errors you miss at this stage won't be found until later testing. They're more expensive to find and correct then, so you should find all that you can at this stage.

CROSS-REFERENCE For details on review techniques, see Chapter 24, "Reviews."

Mentally check the routine for errors. The first formal check of a routine is mental. The clean-up and informal-checking steps mentioned earlier are two kinds of mental checks. Another is executing each path mentally. Mentally executing a routine is difficult, and that difficulty is one reason to keep your routines small. Make sure that you check nominal paths and endpoints and all exception conditions. Do this both by yourself, which is called "desk checking," and with one or more peers, which is called a "peer review," a "walkthrough," or an "inspection," depending on how you do it.

One of the biggest differences between hobbyists and professional programmers is the difference that grows out of moving from superstition into understanding. The word "superstition" in this context doesn't refer to a program that gives you the creeps or generates extra errors when the moon is full. It means substituting feelings about the code for understanding. If you often find yourself suspecting that the compiler or the hardware made an error, you're still in the realm of superstition. Only about 5 percent of all errors are hardware, compiler, or operating-system errors (Brown and Sampson 1973, Ostrand and Weyuker 1984). Programmers who have moved into the realm of understanding always suspect their own work first because they know that they cause 95 percent of errors. Understand the role of each line of code and why it's needed. Nothing is ever right just because it seems to work. If you don't know why it works, it probably doesn't—you just don't know it yet.

HARD DATA

Bottom line: A working routine isn't enough. If you don't know why it works, study it, discuss it, and experiment with alternative designs until you do.

Compile the routine. After reviewing the routine, compile it. It might seem inefficient to wait this long to compile since the code was completed several pages ago. Admittedly, you might have saved some work by compiling the routine earlier and letting the computer check for undeclared variables, naming conflicts, and so on.

You'll benefit in several ways, however, by not compiling until late in the process. The main reason is that when you compile new code, an internal stopwatch starts ticking. After the first compile, you step up the pressure: Get it right with Just One More Compile. The "Just One More Compile" syndrome leads to hasty, error-prone changes that take more time in the long run. Avoid the rush to completion by not compiling until you've convinced yourself that the routine is right.

The point of this book is to show how to rise above the cycle of hacking something together and running it to see if it works. Compiling before you're sure your program works is often a symptom of the hacker mind-set. If you're not caught in the hacking-and-compiling cycle, compile when you feel it's appropriate to.

Here are some guidelines for getting the most out of compiling your routine:

- Set the compiler's warning level to the pickiest level possible. You can catch an amazing number of subtle errors simply by allowing the compiler to detect them.
- Eliminate the causes of all compiler errors and warnings. Pay attention to what the compiler tells you about your code. A lot of warnings often indicates low-quality code, and you should try to understand each warning you get. In practice, warnings you've seen again and again have one of two possible effects: You ignore them and they camouflage other, more important warnings, or they become annoying, like Chinese water torture. It's usually safer and less painful to rewrite the code to solve the underlying problem and eliminate the warnings.

Use the computer to check the routine for errors. Once the routine compiles, put it into the debugger and step through each line of code. Make sure each line executes as you expect it to. You can find many errors by following this simple practice.

CROSS-REFERENCE For details, see Chapter 25, "Unit Testing." Also see "Building Scaffolding to Test Individual Routines" in Section 25.5.

After stepping through the code in the debugger, test it using the test cases you planned while you were developing the routine. You might have to develop scaffolding to support your test cases—code that is used to support routines while they're tested and isn't included in the final product. Scaffolding can be a test-harness routine that calls your routine with test data, or it can be stubs called by your routine.

CROSS-REFERENCE For details, see Chapter 26, "Debugging."

Remove errors from the routine. Once an error has been detected, it has to be removed. If the routine you're developing is buggy at this point, chances are good that it will stay buggy. If you find that a routine is unusually buggy, start over. Don't patch it. Rewrite it. Patches usually indicate incomplete understanding and guarantee errors both now and later. Creating an entirely new design for a buggy routine pays off. Few things are more satisfying than rewriting a problematic routine and never finding another error in it.

CHECKLIST

CROSS-REFERENCE The point of this list is to check whether you followed a good set of steps to create a routine. For a checklist that focuses on the quality of the routine itself, see the "High-Quality Routines" checklist in Chapter 5, page 113.

Constructing a Routine

- ☐ Have you checked that the prerequisites have been satisfied?
- ☐ Have you defined the problem that the routine will solve?
- ☐ Is the architecture clear enough to give your routine a good name?
- ☐ Have you thought about how to test the routine?
- ☐ Have you thought about efficiency mainly in terms of good modularization or in terms of meeting space and speed budgets?
- ☐ Have you checked reference books for helpful algorithms?
- ☐ Have you designed the routine using detailed PDL?
- ☐ Have you thought about the data, before the logic if necessary?
- ☐ Have you mentally checked the PDL? Is it easy to understand?
- ☐ Have you paid attention to warnings that would send you back to architecture (use of global data, operations that seem better suited to another routine, and so on)?
- ☐ Did you use the PDL-to-code process, using PDL as a basis for coding and converting the original PDL to comments?
- ☐ Did you translate the PDL to code accurately?
- ☐ Did you document assumptions as you made them?
- ☐ Have you chosen the best of several design attempts, rather than merely stopping after your first attempt?
- ☐ Do you thoroughly understand your code? Is it easy to understand?

Key Points

- Writing good PDL calls for using understandable English, avoiding features specific to a single programming language, and writing at the level of intent—describing what the design does rather than how it will do it.
- The PDL-to-code process is a useful tool for detailed design and makes coding easy. PDL translates directly into comments, ensuring that the comments are accurate and useful.
- You should check your work at each step and encourage others to check it too. That way, you'll catch mistakes at the least expensive level, when you've invested the least amount of effort.