6

Three out of Four Programmers Surveyed Prefer Modules

Contents

Related Topics

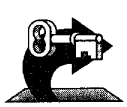
Characteristics of high-quality routines: Chapter 5
High-level design: Chapter 7
Abstract data types: Section 12.3

"YOU'VE GOT YOUR ROUTINE IN MY MODULE."

"No, you've got your module around my routine."

The distinction between routines and modules is one that some people don't make very carefully, but you should understand the difference so that you can use modules to their full advantage.

"Routine" and "module" are flexible words that have different meanings in different contexts. In this book's context, a routine is a function or procedure invocable for a single purpose, as described in Chapter 5.

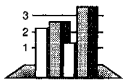


KEY POINT

A module is a collection of data and the routines that act on the data. A module might also be a collection of routines that provides a cohesive set of services even if no common data is involved. Examples of modules include a source file in C, a package in Ada, and a unit in some versions of Pascal. If your language doesn't support modules directly, you can still get many of their benefits by emulating them with disciplined programming practices.

6.1 Modularity: Cohesion and Coupling

"Modularity" has one foot in routine design, one foot in module design. The basic idea is powerful and worth exploring.



HARD DATA

In the *Software Maintenance Guidebook,* Glass and Noiseux argue that modularity contributes far more to maintainability than structure does and is the most important factor in preventing corrective maintenance (1981). In *Software Maintenance Management,* Lientz and Swanson cite a study finding that 89 percent of code users reported improved maintainability with modular programming (1980). On a test of comprehension, Shneiderman and Mayer found that programmers scored 15 percent higher on a modular program than on a nonmodular one (1979).

The modularity goal for individual routines is to make each routine like a "black box": You know what goes in, and you know what comes out, but you don't know what happens inside. A black box has such a simple interface and such well-defined functionality that for any specific input you can accurately predict the corresponding output. If your routines are like black boxes, they're perfectly modular, perform well-defined functions, and have simple interfaces.

The goal of perfect modularity is often hard to achieve with individual routines, however, and that's where modules come in. A group of routines often needs to share a set of common data. In such cases, the individual routines aren't perfectly modular because they share data with other routines. As individual routines, they thus don't have simple interfaces. As a group, however, the routines can present a simple interface to the rest of the world, and as a group, the routines can be perfectly modular.

Module Cohesion

CROSS-REFERENCE
For details on cohesion of routines rather than modules, see Section 5.3, "Strong Cohesion."

The criterion for module cohesion is as simple as the criterion for the cohesion of an individual routine. A module should offer a group of services that clearly belong together.

You might have a module that implements a cruise-control simulator. It would contain data describing the car's current cruise-control setting and current speed. It would offer services to set the speed, resume the former speed, and deactivate. Internally, it might have additional routines and data to support these services, but routines outside the module wouldn't need to know anything about them. The module would have great cohesion because every routine in it would work toward implementing the cruise-control simulator.

Or you might have a module of trigonometry functions. The module might include *sin()*, *cos()*, *tan()*, *arcsin()*, *arccos()*, and *arctan()*, a cohesive set of routines. If the routines were standard trig functions, they wouldn't need to share any data, but the routines would be related, so the module would still have great cohesion.

An incohesive module would be one that contained a collection of miscellaneous functions. Suppose that a module contains routines to implement a stack: *init_stack()*, *push()*, and *pop()*; that it contains routines to format report data; and that it defines all the global data used in a program. It's hard to see any connection among the stack and report routines or the module's data, so the module wouldn't be cohesive. The routines should be reorganized into more-focused modules.

The evaluations of module cohesion in these examples are based on each module's package of data and services. This cohesion is at the level of the module as a whole. The routines inside the module aren't necessarily cohesive just because the overall module is. The routines inside the module need to be designed so that they're individually cohesive too. For guidelines on that, see Section 5.3, "Strong Cohesion."

Module Coupling

CROSS-REFERENCE
For details on coupling of routines rather than modules, see Section 5.4, "Loose Coupling."

The criterion for a module's coupling to the rest of the program is similar to the criterion for the coupling of an individual routine. A module should offer a collection of services designed so that the rest of the program can interact with it cleanly.

In the cruise-control example, the module offered the services *SetSpeed()*, *GetCurrentSettings()*, *ResumeFormerSpeed()*, and *Deactivate()*. This is a complete set of services that allows the rest of the program to interact with the module entirely through its official, public interface.

If a module offers an incomplete set of services, other routines might be forced to read or write its internal data directly. That opens up the module, making it a glass box instead of a black box, and virtually eliminates the module's "modularity." Larry Constantine, the principal developer of structured design, points out that a module's services should be "fully factored" so that its clients can tailor the services to their own needs (Constantine 1990a).

In designing a module for maximum cohesion and minimal coupling, you'll find that you have to balance trade-offs between criteria for designing modules and criteria for designing individual routines. One guideline for reducing coupling between individual routines is to minimize the use of global data.

Part of the reason for creating a module, however, is so that routines can share data; you want to make data accessible to all the routines in a module without passing it through their parameter lists.

CROSS-REFERENCE
For more details on the differences between module data and global data, see "Module data mistaken for global data" in the next section.

Module data is like global data in that more than one routine can access it. It's unlike global data in that it's not accessible to all the routines in a program; it's accessible only to the routines in a single module. Consequently, one of the most important decisions in designing a module is deciding which routines require direct access to module data. If a routine can access the data just as well through the module's service routines and if it doesn't have another compelling reason to stay in the module, kick it out.

6.2 Information Hiding

If you've been reading the cross-references throughout the book, you've probably noticed about 400 references to information hiding. With that many references, it had better be important.

It is.

KEY POINT

The idea of information hiding comes into play at all levels of design, from the use of named constants instead of literals, to routine design, to module and program design. Because the idea is often best applied at the module level, it's discussed at length in this chapter.

HARD DATA

Information hiding is one of the few theoretical techniques that has indisputably proven its value in practice (Boehm 1987a). Large programs that use information hiding have been found to be easier to modify—by a factor of 4—than programs that don't (Korson and Vaishnavi 1986). Moreover, information hiding is part of the foundation of both structured design and object-oriented design. In structured design, the notion of black boxes comes from information hiding. In object-oriented design, information hiding gives rise to the notions of encapsulation and abstraction.

Secrets and the Right to Privacy

CROSS-REFERENCE
For a description of encapsulation in object-oriented design, see "Key Ideas" in Section 7.3.

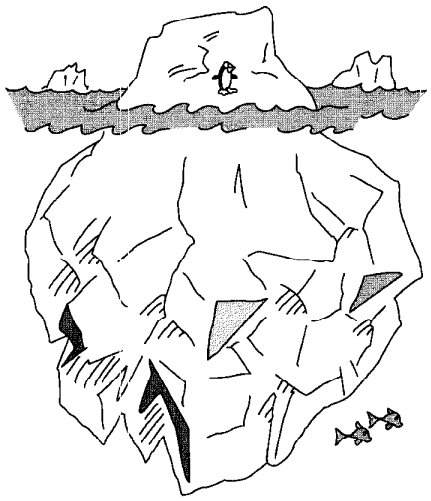
The key concept in information hiding is the idea of "secrets." Each module is characterized by design or implementation decisions that it hides from all other modules. The secret might be an area that's likely to change, the format of a file, the way a data structure is implemented, or an area that needs to be walled off from the rest of the program so that errors in it cause as little

damage as possible. The module's job is to keep this information hidden and to protect its own right to privacy. Another term for information hiding is "encapsulation," which expresses the idea of a package that has an outside that's distinct from its inside.

Regardless of what you call it, information hiding is a way of designing routines and, more important, modules. When you hide secrets, you design a collection of routines that access a common data set. Minor changes to a system might affect several routines, but they should affect only one module.

One key task in designing a module is deciding which features should be known outside the module and which should remain secret. A module might use 25 routines and expose only 5 of them, using the other 20 internally. A module might use several data structures and expose no information about them. It might or might not provide routines that give data-structure information to the rest of the program. This aspect of module design is also known as "visibility" since it has to do with which features of the module are "visible," "exposed," or "exported" outside the module.

The interface to a module should reveal as little as possible about its inner workings. A module is a lot like an iceberg: Seven-eighths is under water, and you can see only the one-eighth that's above the surface.



A good module interface is like the tip of an iceberg, leaving most of the module unexposed.

Designing the module interface is an iterative process just like any other aspect of design. If the interface isn't right the first time, try a few more times until it stabilizes. If it doesn't stabilize, it needs to be redesigned.

Various diagramming techniques are available for representing modules. The key aspect of a module diagram is that it differentiates between services that are available within a module and services that are exposed to the outside world. The kind of diagram generally known as a "module diagram" was originally developed by Grady Booch for use in conjunction with Ada development. Figure 6-1 shows an adaptation:

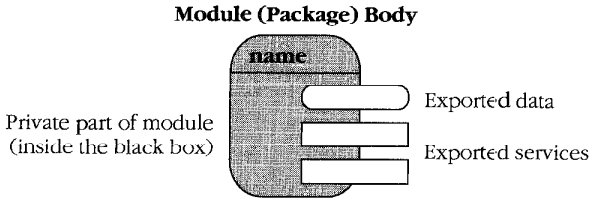


Figure 6-1. *A module diagram differentiates between the public and private parts of a module. The public parts are shown as the rounded and rectangular bars coming out of the bigger rectangle. The private part is shown as a "black box."*

Information hiding doesn't necessarily imply a system shape: A system might have a hierarchical structure, or it might have a network structure like the one sketched in Figure 6-2.

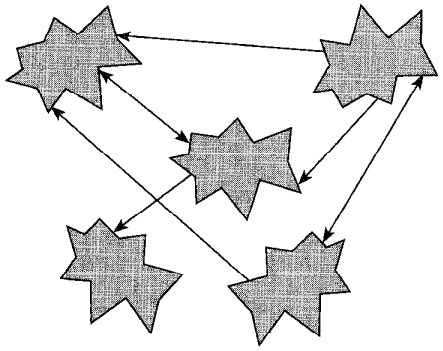


Figure 6-2. *Some systems are shaped like networks rather than hierarchies.*

In a network structure, you simply specify which modules can communicate with which other modules and how the specific communication occurs, and then you make the connections. As Figure 6-3 shows, the module diagram works in the network configuration too.

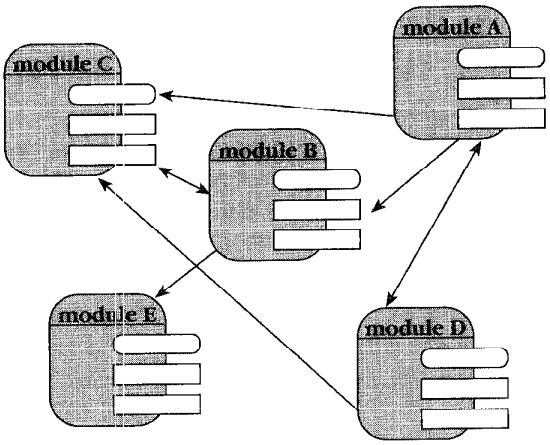


Figure 6-3. *Network-shaped systems are represented naturally with a notation that includes the idea of information hiding.*

Example of Information Hiding

A few years ago I wrote a medium-sized (20K lines of code) system that made extensive use of a linked-list structure. The problem area was made up of data nodes, each of which was connected to subordinate, superordinate, and peer nodes. I chose to use a linked-list data structure, so throughout the code, I had statements like

```
node = node.next
```

and

```
phone = node.data.phone
```

These statements directly manipulated the linked-list data structure. Although the linked list modeled the problem naturally, taking that approach turned out to be an inefficient use of memory. I wanted to replace the memory pointers with integer array indexes, which would have halved the memory used and created opportunities for performance optimizations in other areas.

CROSS-REFERENCE
A way to handle complex changes like this is to use a multi-file search-and-replace tool that handles regular expressions. For details on such tools, see "Editing" in Section 20.2.

I couldn't make the change easily because I had hard-coded statements like the ones above throughout the program, and I wasn't going to wade through 20,000 lines of code looking for them. If, instead, I had created a module with access routines like

```
node  = NearestNeighbor( node )
phone = EmergencyContact( node )
```

I would have had to change the code in only one place—in the access routines.

Exactly how much memory would I have gained? How much speed would I have gained or lost? I don't know. But if I had hidden the details of the data structure and used access routines, I could easily have found out, and I could have tried several other approaches too. I could have selected the best alternative from a group of alternatives rather than using the one I was stuck with because I had exposed the details of the data structure to the whole program.

CROSS-REFERENCE
When information hiding is applied to data structures, as it is in this example, you can treat the data as an abstract data type. For more details, see Section 12.3, "Abstract Data Types (ADTs)."

The second big reason, after ease of modification, to hide details of complex data structures is that hiding the details clarifies the intent of your code. In the example above, experienced programmers would have no problem reading

```
node = node.next
```

Clearly, the statement refers to a linked-list structure, but it tells you little else. On the other hand, an access routine such as *node = NearestNeighbor(node)* describes what the linked list represents, which is useful. And that makes you realize that a name such as *node* leaves room for improvement. (What does *node* have to do with neighbors?) Computer-sciencey statements such as *node = node.next* are so completely removed from the real-world problem that you wouldn't even think about improving them to convey information about the real-world problem.

The final big reason to hide complex data structures is reliability. If you access data structures through specialized routines, you can build in safeguards that would be awkward to build in everywhere that the routines reference a variable. For example, if you have a linked list and you want to get the next element in the list, being careful not to run past the end of the list, you could code something like

```
if ( node.next <> null ) then
   node = node.next
```

Depending on the specific circumstances, if you wanted to be really careful, you could code something like

```
if ( node <> null ) then
   if ( node.next <> null ) then
      node = node.next
```

If you have a lot of *node = node.next* code spread throughout your program, you might make the test in some places, skip it in others. But if the operation is isolated in a routine call,

```
node = NearestNeighbor( node )
```

you make the test once, in the routine, and it works for the whole program. Moreover, sometimes, even if you intend to make the test everywhere *node* is used, it's easy to forget the test in a few places. If the linked-list manipulation is isolated in a routine, however, it's not possible to forget the tests. They are done automatically.

Another advantage of hiding a data structure's implementation details is easier debugging. Suppose you know that the *node* variable is getting bad values somewhere, and you don't know where. If the code that accesses *node* is spread throughout your program, good luck in ever finding the source of the bad data. But if it's isolated in a single routine, you can add a debugging test to check *node* in that one place, every time it's accessed.

A last advantage pertaining to reliability is that access routines provide a way of making all accesses to data follow a parallel organization: Either manipulate a data structure directly or use access routines—don't do both. Of course, within the module responsible for the data structure, the accesses are direct, and lack of parallelism at that level is unavoidable. The goal is to avoid airing dirty laundry in public, which is accomplished by hiding the ugly secrets of direct data manipulation in access routines.

Common Kinds of Secrets

You'll deal with many kinds of secrets specific to the project you're working on, but you'll deal with a few general classes of secrets again and again:

- Areas likely to change
- Complex data
- Complex logic
- Operations at the programming-language level

Each of these classes of secrets is discussed in more detail below.

Areas likely to change

Accommodating changes is one of the most challenging aspects of good program design. The goal is to isolate unstable areas so that the effect of a change will be limited to one module. Here are the steps you should follow in preparing for such perturbations.

FURTHER READING
These steps are adapted from "Designing Software for Ease of Extension and Contraction" (Parnas 1979).

1. Identify items that seem likely to change. If the requirements have been done well, they include a list of potential changes and the likelihood of each change. In such a case, identifying the likely changes is easy. If the requirements don't cover potential changes, see the discussion that follows of areas that are likely to change in any project.
2. Separate items that are likely to change. Compartmentalize each volatile component identified in step 1 into its own module, or into a module with other volatile components that are likely to change at the same time.
3. Isolate items that seem likely to change. Design the intermodule interfaces to be insensitive to the potential changes. Design the interfaces so that changes are limited to the inside of the module and the outside remains unaffected. Any other module using the changed module should be unaware that the change has occurred. The module's interface should protect its secrets.

Here are a few areas that are likely to change:

Hardware dependencies. For screens, printers, and plotters, be aware of possible changes in dimensions, resolution, number of colors, available fonts, memory, control codes, and graphics capabilities. Other hardware dependencies include interfaces with disks, tapes, communications ports, sound facilities, and so on.

Input and output. At a slightly higher level of design than raw hardware interfaces, input/output is a volatile area. If your application creates its own data files, the file format will probably change as your application becomes more sophisticated. User-level input and output formats will also change—the positioning of fields on the page, the number of fields on each page, the sequence of fields, and so on. In general, it's a good idea to examine all external interfaces for possible changes.

Nonstandard language features. If you use nonstandard extensions to your programming language, hide those extensions in a module of their own so that you can replace them with your own code when you move to a different environment. Likewise, if you use library routines that aren't available in all environments, hide the actual library routines behind an interface that works just as well in another environment.

Difficult design and implementation areas. It's a good idea to hide difficult design and implementation areas because they might be done poorly and you might need to do them again. Compartmentalize them and minimize the impact their bad design or implementation might have on the rest of the system.

Status variables. Status variables indicate the state of a program and tend to be changed more frequently than most other data. In a typical scenario, you

might originally define an error-status variable as a boolean variable and decide later that it would be better implemented as an enumerated type with the values *NoError*, *WarningError*, and *FatalError*.

You can add at least two levels of flexibility and readability to your use of status variables:

CROSS-REFERENCE
For details on simulating enumerated types using integers, see "If Your Language Doesn't Have Enumerated Types" in Section 11.6.

- Don't use a boolean variable as a status variable. Use an enumerated type instead. It's common to add a new state to a status variable, and adding a new type to an enumerated type requires a mere recompilation rather than a major revision of every line of code that checks the variable.
- Use access routines rather than checking the variable directly. By checking the access routine rather than the variable, you allow for the possibility of more sophisticated state detection. For example, if you wanted to check combinations of an error-state variable and a current-function-state variable, it would be easy to do if the test were hidden in a routine and hard to do if it were a complicated test hard-coded throughout the program.

Data-size constraints. When you declare an array of size *15*, you're exposing information to the world that the world does not need to see. Defend your right to privacy! Information hiding isn't always as complicated as a collection of functions packaged into a module. Sometimes it's as simple as using a constant such as *MAX_EMPLOYEES* to hide a *15*.

Business rules. Business rules are the laws, regulations, policies, and procedures that you encode into a computer system. If you're writing a payroll system, you might encode rules from the IRS about the number of allowable withholdings and the estimated tax rate. Additional rules for a payroll system might come from a union contract specifying overtime rates, vacation and holiday pay, and so on. If you're writing a program to quote auto insurance rates, rules might come from state regulations on required liability coverages, actuarial rate tables, or underwriting restrictions.

Such rules tend to be the source of frequent changes in data-processing systems. Congress changes the tax structure, or an insurance company changes its rate tables. If you follow the principle of information hiding, logic based on these rules won't be strewn throughout your program. The logic will stay hidden in a single dark corner of the system until it needs to be changed.

Anticipating changes. When thinking about potential changes to a system, design the system so that the effect or scope of the change is proportional to the chance that the change will occur. If a change is likely, make sure that the system can accommodate it easily. Only extremely unlikely changes should be allowed to have drastic consequences for more than one module in a system.

FURTHER READING
This point is adapted from "Designing Software for Ease of Extension and Contraction" (Parnas 1979).

A good technique for identifying areas likely to change is first to identify the minimal subset of the program that might be of use to the user. The subset makes up the core of the system and is unlikely to change. Next, define minimal increments to the system. They can be so small that they seem trivial. These areas of potential improvement constitute potential changes to the system; design these areas using the principles of information hiding. By identifying the core first, you can see which components are really add-ons and then extrapolate and hide improvements from there.

Complex data

CROSS-REFERENCE
Additional techniques for hiding complex data are presented in Section 12.3, "Abstract Data Types (ADTs)."

All complex data is likely to change; if it's complicated and you use it much, you'll discover better ways to implement it after you've worked with it at the implementation level and are well into coding. If you use information-hiding principles to hide the implementation of the data, you can change to a better implementation with little fuss. If not, every time you work with the data, you'll think of how you want to change the implementation and how much easier it would be to change if you'd used information hiding.

The extent to which you'll hide complex data depends somewhat on the size of the program you're writing. If you have a small program of a few hundred lines of code and want to manipulate a variable directly, go ahead. It might hurt the program, but then again, it might not. Consider the small program's future before committing yourself to the maintenance headache of direct data manipulation. If you're working on a larger program or using global data, consider using access routines as a matter of course.

Complex logic

CROSS-REFERENCE
Techniques for hiding complex boolean tests are discussed in "Making Complicated Expressions Simple" in Section 17.1.

Hiding complex logic improves program readability. The complex logic isn't always the most important aspect of a routine, and hiding it makes the main activity of the routine clearer. Complex logic is about as likely to change as complex data, so it's a good idea to insulate the rest of the program from the effects of such changes. In some instances, you can hide the kind of logic used—for example, you can code certain tests as a big *if* statement, *case* statement, or table lookup. No code other than the code that performs the logic needs to know the ugly details. If the rest of the code just needs to know the result, just give it the result.

Operations at the programming-language level

The more you make the program read like a solution to the real-world problem and the less like a collection of programming-language constructs, the better your program will be. Hide the information about computer-science entities. For example, the statement

```
EmployeeID = EmployeeID + 1
CurrentEmployee = EmployeeList[ EmployeeID ]
```

is not bad programming, but it represents the problem in computer-science terms. Instead, deal with the operation at a higher level of abstraction:

```
CurrentEmployee = NextAvailableEmployee()
```

or possibly

```
CurrentEmployee = NextAvailableEmployee( EmployeeList, EmployeeID )
```

By adding a routine that hides the computer-science interpretation of what's happening, you deal with the problem at a higher level of abstraction. This makes your intent clearer and makes the code easier to understand, work with, and modify.

Suppose you use a linked list to implement a scheduling queue. The functions *HighestPriorityEvent()*, *LowestPriorityEvent()*, and *NextEvent()* are abstract functions that would hide the implementation details. The functions *FrontOfQueue()*, *BackOfQueue()*, and *NextInQueue()* don't hide much because they refer to the implementation—exposing secrets that should be hidden.

Generally, design one level of routines that deal with data at the level of programming-language statements. Hide direct data manipulations inside that group of routines. The rest of your program can then work at the more abstract level of the problem area, as the abstract functions above do.

Barriers to Information Hiding

FURTHER READING
Parts of this section are adapted from "Designing Software for Ease of Extension and Contraction" (Parnas 1979).

Most of the barriers to information hiding are mental blocks built up from the habitual use of other techniques. In some instances, information hiding is truly impossible. Certain practices that seem to create barriers to information hiding, however, might only be excuses.

Excessive distribution of information

One common barrier to information hiding is an excessive distribution of information throughout a system. You might have hard-coded the literal *100* throughout a system. Using *100* as a literal decentralizes references to it. It's better to hide the information in one place, in a constant *MaxEmployees* perhaps, whose value is changed in only one place.

Another example of excessive information distribution is threading interaction with human users throughout a system. If the mode of interaction changes—say, from a command-line to a forms-driven interface—virtually all the code will have to be modified. It's better to concentrate user interaction in a single module you can change without affecting the whole system.

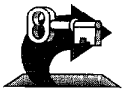
Yet another example would be a global data structure—perhaps an array of employee data with 1000 elements maximum that's accessed throughout a

program. If the program uses the global data directly, information about the data structure's implementation—such as the fact that it's an array and has a maximum of 1000 elements—will be spread throughout the program. If the program uses the data structure only through access routines, only the access routines will know the implementation details.

Circular dependencies

A more subtle barrier to information hiding is circular dependencies, as when a routine in module *A* calls a routine in module *B*, and a routine in module *B* calls a routine in module *A*. Avoid such dependency loops. They make it hard to test a system because you can't test either module *A* or module *B* until at least part of the other is ready. If the program will be overlaid, module *A* and module *B* will always need to be in memory at the same time to prevent thrashing. Eliminating circularities is nearly always possible by identifying the parts of modules *A* and *B* used by the other module, factoring those parts into new modules *A'* and *B'*, and calling *A'* and *B'* with whatever is left of modules *A* and *B*.

Module data mistaken for global data



KEY POINT

If you're a conscientious programmer, one of the barriers to effective information hiding might be thinking of module data as global data and avoiding it because you want to avoid the problems associated with global data. As was mentioned in Section 6.1, "Modularity: Cohesion and Coupling," the two kinds of data are different. Accessible only to the routines in a single module, module data entails far fewer risks than global data.

CROSS-REFERENCE
For several examples of cases in which you need to use module data to program effectively, see Section 12.3, "Abstract Data Types (ADTs)."

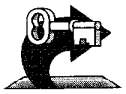
If you don't use module data, you don't realize all of the power that comes from creating modules. A module can't be truly responsible for a collection of data and operations on the data if other routines pass it the data that only it can manipulate. For instance, an earlier example advised providing a level of abstraction by having an assignment such as

```
CurrentEmployee = NextAvailableEmployee()
```

or possibly

```
CurrentEmployee = NextAvailableEmployee( EmployeeList, EmployeeID )
```

The difference between the two assignments is that in the first case, *NextAvailableEmployee()* owns the information about the employee list and about which entry is the current entry in the list. In the second example, *NextAvailableEmployee()* is only borrowing the information from the routine that passes it in. In order to provide a full level of abstraction when you use *NextAvailableEmployee()*, you shouldn't need to worry about the data it needs. Make it take responsibility for its own problems!



KEY POINT

Global data is generally subject to two problems: (1) Routines operate on global data without knowing that other routines are operating on it; and (2) routines are aware that other routines are operating on the global data, but they don't know exactly what they're doing to it. Module data isn't subject to either of these problems. Direct access to the data is restricted to a few routines packaged into a single module. The routines are aware that other routines operate on the data, and they know exactly which other routines they are. If you're still not convinced, try it. You'll be happy with the results.

Perceived performance penalties

CROSS REFERENCE
Code-level performance optimizations are discussed in Chapter 28, "Code-Tuning Strategies," and Chapter 29, "Code-Tuning Techniques."

A final barrier to information hiding can be an attempt to avoid performance penalties at both the architectural and the coding levels. You don't need to worry at either level. At the architectural level, the worry is unnecessary because architecting a system for information hiding doesn't conflict with architecting it for performance. If you keep both information hiding and performance in mind, you can achieve both objectives.

The more common worry is at the coding level. The concern is that accessing data structures indirectly incurs run-time performance penalties for additional levels of routine calls and so on. This concern is premature. Until you can measure the system's performance and pinpoint the bottlenecks, the best way to prepare for code-level performance work is to create a highly modular design. When you detect hot spots later, you can optimize individual routines without affecting the rest of the system.

Further Reading

I'm not aware of any comprehensive treatment of information hiding. Most sofware-engineering textbooks discuss it briefly, frequently in the context of object-oriented techniques. The three Parnas papers listed below are the seminal presentations of the idea and are probably still the best resources on information hiding.

Parnas, David L. "On the Criteria to Be Used in Decomposing Systems into Modules." *Communications of the ACM* 5, no. 12 (December 1972): 1053–58.

Parnas, David L. "Designing Software for Ease of Extension and Contraction." *IEEE Transactions on Software Engineering* SE-5, no. 2 (March 1979): 128–38.

Parnas, David L., Paul C. Clements, and D. M. Weiss. "The Modular Structure of Complex Systems." *IEEE Transactions on Software Engineering* SE-11, no. 3 (March 1985): 259–66.

Freeman, Peter, and Anthony I. Wasserman, eds. *Tutorial on Software Design Techniques*, 4th ed. Silver Spring, Md.: IEEE Computer Society Press, 1983. This collection includes the first two Parnas papers above and about 45 others.

6.3 Good Reasons to Create a Module

Even if you aren't using modules much yet, you probably have an intuitive idea of the kinds of routines and data you can collect in a module. In a sense, modules are poor folks' objects. They're collections of data and operations on data, and they support the object-oriented concepts of abstraction and encapsulation. They don't support inheritance, so they don't fully support object-oriented programming. The term that has emerged to describe this limited kind of object orientation is "object-based" programming (Booch 1991).

Here are some good areas in which to create modules:

User interface. Create a module to isolate user-interface components so that the user interface can evolve without damaging the rest of the program. In many cases, a user-interface module uses several subordinate modules for menu operations, window management, the help system, and so forth.

Hardware dependencies. Isolate hardware dependencies in their own module or modules. Examples of hardware dependencies include interfaces to screens, printers, keyboards, mice, disk drives, and communications devices. Isolating such dependencies helps when you move the program to a new hardware environment. It also helps initially when you're developing a program for volatile hardware. You can write software that simulates interaction with specific hardware, have the hardware-interface routines use the simulator as long as the hardware is unstable or unavailable, and then unplug the hardware-interface routines from the simulator and plug the routines into the hardware when it's ready to use.

Input and output. Encapsulate input/output operations to protect the rest of the program from volatile file and report formats. Having a module for input/output also makes a program easier to adapt to other input/output devices.

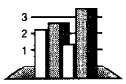
System dependencies. Package operating-system dependencies into a module for the same reason you package hardware dependencies. If you're developing a program for Microsoft Windows, for example, why limit yourself to the Windows environment? Isolate the Windows calls in a Windows-interface module. If you later want to move your program to a Macintosh or the OS/2 Presentation Manager, all you'll have to rewrite is the interface module.

Data management. Hide data management in its own module. Have routines inside the module worry about messy implementation-language details. Have routines outside the module work with the data in abstract terms, terms that are as close to the real-world problem as possible. Referring to this data-management module as a single module is probably misleading. You'll normally need individual modules to handle each major abstract data type.

CROSS-REFERENCE
For details on abstract data types, see Section 12.3, "Abstract Data Types (ADTs)."

Real-world objects and abstract data types. Create a module for each real-world object that your program models. Put the data needed for the object into the module, and then build service routines that model the behavior of the object. This is known as creating an abstract data type.

Reusable code. Modularize parts of the program that you plan to reuse in other programs. One of the huge advantages of creating modules is that reuse is more practical with them than with functionally oriented programs. NASA's Software Engineering Laboratory studied ten projects that pursued reuse aggressively (McGarry, Waligora, and McDermott 1989). In both the object-based and the functionally oriented approaches, the initial projects weren't able to take much of their code from previous projects because previous projects hadn't established a sufficient code base. Subsequently, the projects that used functional design were able to take about 35 percent of their code from previous projects. Projects that used an object-based approach were able to take more than 70 percent of their code from previous projects. If you can avoid writing 70 percent of your code by planning ahead, do it!



HARD DATA

Related operations that are likely to change. Build a partition around groups of related operations that are likely to change. This is a damage-containment policy that prevents changes in specific areas from affecting the rest of the program. Section 6.2 on information hiding described several common sources of change.

Related operations. Finally, put related operations together. In most cases, you'll be able to find a stronger organizing principle than grouping routines and data that seem related. In cases in which you can't hide information, share data, or plan for flexibility, you can still package sets of operations into sensible groups such as trig functions, statistical functions, string-manipulation routines, bit-manipulation routines, graphics routines, and so on. Who knows—if you group related operations carefully, you might even be able to reuse them on your next project.

6.4 Implementing Modules in Any Language

Some languages provide direct support for modularity. Others need to be supplemented with programming standards.

Language Support Needed for Modularity

A module consists of data, data types, operations on data, and a distinction between public and private operations. To support modularity, a language needs to support multiple modules. Without multiple modules, any other requirement is moot.

Data needs to be accessible and hideable at three levels: local, module, and global. Most languages support local data and global data. Language support for module data, data accessible to some but not all routines, is required if some data is to be private to a collection of routines in a module.

The demand for data type accessibility and hideability in modules is similar to the requirement for data. Some types should be hidden within a specific module; others should be made available to other modules. The module needs to be able to control which other modules know about its types.

The demands on module-level routines are similar. Some routines should be accessible only within a specific module, and the module should be able to control whether routines are public or private. Outside the module, no other modules or routines should know that the private routines exist. If the module is designed well, other modules and routines shouldn't have any reason to care about the existence of the private routines.

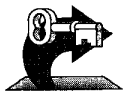
Language-Support Summary

The following table summarizes the support of several languages for constructs necessary to provide information hiding:

Language	Multiple Modules	Data			Data Types			Routines	
		Local	Module	Global	Local	Module	Global	Private	Module/Global
Ada	•	•	•	•	•	•	•	•	•
C	•	•	•	•	•	†	•	•	•
C++	•	•	•	•	•	•	•	•	•
Fortran 77	†	•	†	•	–	–	–	–	•
Generic Basic	–	–	–	•	–	–	–	–	•
Generic Pascal	–	•	–	•	•	–	•	–	•
Turbo Pascal	•	•	•	•	•	•	•	•	•
QuickBasic	•	•	•	•	•	–	–	–	•

† Possible if the language is used with discipline; the capability is not entirely dependent on programming standards.

Generic Basic and generic Pascal don't support multiple modules, so they don't get to first base in supporting modularity. Fortran and QuickBasic don't have the control over data types required to support modularity. Only Ada, C, C++, and Turbo Pascal allow their modules to restrict access to specified routines so that the routines are truly private.



KEY POINT

In short, unless you're using Ada, C, C++, or Turbo Pascal, you'll have to supplement your language's capabilities with naming and other conventions to simulate the use of modules. The following sections take a look at the languages that support modularity directly and tell you how to simulate modularity in other languages.

Ada and Modula-2 support

Ada supports modularity through the notion of "packages." If you program in Ada, you already know how to create packages. Modula-2 supports it through the notion of modules. Even though Modula-2 isn't featured in this book, its support for modularity is so direct that an example of a Modula-2 implementation of a module for an event queue is shown below:

Modula-2 Example of a Module Declaration

```
definition module Events;
   export
      EVENT,
      EventAvailable,
      HighestPriorityEvent,
      LowestPriorityEvent;

   type
      EVENT = integer;
   var
      EventAvailable: boolean;          { true if an event is available }

   function HighestPriorityEvent: Event;
   function LowestPriorityEvent: Event;

end Events;
```

These items are public.

Object-oriented language support

Object-oriented languages such as C++ also support modularity directly. Modularity is at the core of object-oriented programming. Here's how an event-queue module (object) looks in C++.

C++ Example of a Module (Object) Declaration

```
class Buffer
   {
   public:

   typedef int EVENT;

   BOOL EventAvailable;          /* true if an event is available */

   EVENT HighestPriorityEvent( void );
   EVENT LowestPriorityEvent( void );

   private:
   ...
   };
```

Pascal support

Some versions of Pascal, notably versions 4 and later of Turbo Pascal, support modularization through the notion of units. A "unit" is a data file that can contain data types, data, and routines. A unit has an interface section that declares the data and the routines that can be used outside the module. The rest of the data and routines are available only inside the unit. Data can also be declared to be available to all functions or procedures within a file and only within that file. That gives Turbo Pascal the local, module, and global levels of data accessibility necessary to support modularity. Here's how the event-queue module would look in Turbo Pascal version 5:

Turbo Pascal Example of a Module Declaration

```
unit Events;

INTERFACE

   type
      EVENT = integer;
   var
      EventAvailable: boolean;          { true if an event is available }

   function HighestPriorityEvent: Event;
   function LowestPriorityEvent: Event;

IMPLEMENTATION
...

end. { unit Events }
```

Routines and data in this part of the file are hidden from routines in other files unless they are declared in the INTERFACE *section above.*

Implementations of Pascal that adhere to the generic Pascal standard don't support modularity directly, but you can supplement them to achieve modularity, as discussed later.

C support

C also supports modules, although many C programmers aren't accustomed to using modules in C. Each C source file can contain both data and functions. You can declare the data and functions to be *static,* which makes them available only inside the source file. If they're not declared *static,* they're available outside the source file. When each source file is thought of as a module, C fully supports modularity.

Because a source file isn't exactly the same as a module, you'll need to create two header files for each source file—one that acts as a public, module header file and one that acts as a private, source-file header file. In the source file's public header file, put only the public data and function declarations. Here's an example:

C Example of a Public, Module Header File

This file contains only the publicly available type, data, and function declarations.

```
/*   File: Event.h
     Contains public declarations for the "Event" module. */

typedef int EVENT;

extern BOOL EventAvailable;     /* true if an event is available */

EVENT HighestPriorityEvent( void );
EVENT LowestPriorityEvent( void );
```

In the source file's private header file, put all the private data and function declarations used internally by the routine. *#include* the header file only in the source file that makes up a module; don't allow other files to *#include* it. Here's an example:

C Example of a Private, Source-File Header File

Here are the private types, data, and functions.

```
/*   File: _Event.h
     Contains private declarations for the "Event" module. */

/* private declarations */
...
```

Use *#include* for both header files in the *.c* source file. In other modules that use the public routines from the *Event* module, *#include* only the public, module header file.

If you need more than one source file to make a single module, you might need a private header file for each source file, but you should create only one public, module header file for the group of files that make up the module. It's all right for the source files within the module to *#include* private header files from the other source files in the same module.

Fortran support

Fortran 90 provides full support for modules. Fortran 77, used with discipline, provides limited support for modules. It has a mechanism for constructing a group of routines that have exclusive access to data. The mechanism is multiple entry points. Although indiscreet use of multiple entry points begs for problems, careful use provides a way to give a group of routines access to common data without making the data public.

You can declare variables as local, global, or *COMMON*. Such distinctions are another means of providing restricted access to data among routines. Define a set of routines that make up a module. The routines won't constitute a module in any sense that the Fortran compiler recognizes, but they will in a logical sense. Have each routine in the module use a named *COMMON* to access the variables it has permission to manipulate directly. Don't allow routines outside the module to use that particular named *COMMON*. Programming standards to make up for language deficiencies are discussed next.

Faking Modularity

What about modularity in generic Basic, generic Pascal, and other languages that don't support modules directly or indirectly? The germ of the answer was mentioned earlier: Use programming standards as a substitute for direct language support. Even if your compiler doesn't enforce good programming practices, you can adopt coding standards that do. The following discussions suggest standards for each of the required aspects of modularity.

Packaging data and routines into modules

Packaging can be simple or complicated depending on whether your language allows you to use multiple source files or requires that all code be in one source file. If it allows you to use multiple source files, put all the data and code used by a single module into a single source file. If you have 10 modules, create 10 source files. If your environment requires everything to be in one file, separate the file into sections for each module. Clearly identify the beginning and end of each module section with comments.

Keeping a module's internal routines private

If your language doesn't support restrictions on the visibility of internal routines, all routines are publicly available to all other routines. Use coding stan-

dards to emphasize your intention that only the routines designated as public be used publicly, regardless of what the compiler allows. Here's what to do:

Identify public and private routines prominently in comments where they're declared. Clearly identify the module with which each routine is associated. If someone has to look up the routine declaration in order to use the routine, make sure it's marked noticeably as public or private.

Require that no routines use internal routines from other modules. Use comments to associate all routines with the modules they belong to.

CROSS-REFERENCE
For details on naming conventions, see Section 9.3, "The Power of Naming Conventions," and the sections following it.

Adopt a naming convention that indicates whether routines are internal or external. You could have the names of all internal routines begin with an underscore (_). This convention wouldn't differentiate between the internal routines of one module and the internal routines of another, but most people should be able to determine whether the internal routine is from their own module. If it isn't, it's obviously from some other module.

Adopt a naming convention that indicates the source of the routine and whether it's internal or external. Details of the specific convention you adopt depend a lot on how much flexibility your language gives you in creating routine names. For example, an internal routine from the *DataRetrieval* module might start with a *dr_* prefix. An internal *UserInterface* routine might start with a *ui_* prefix. External routines from the same modules might start with *DR_* and *UI_* prefixes. If you're limited to a few characters (for example, only six characters in ANSI Fortran 77), naming-convention characters will cost dearly and you'll have to develop a standard with that restriction in mind.

Keeping a module's internal data private

The guidelines for keeping module-level data private are about the same as those for keeping module-level routines private. In general, adopt coding standards that make it clear that only specific data is to be used outside the module, regardless of what the compiler allows. Here are the steps to take:

First, use comments to document whether data is public or private. Clearly identify the module with which each piece of data is associated.

Second, require that no routines use private, module-level data from other modules, even if the compiler treats the data as global.

Third, adopt a naming convention that calls attention to the distinction between public and private data. For consistency's sake, handle this convention similarly to the naming convention for routines.

Fourth, adopt a naming convention that indicates which module owns the data and whether the data is internal or external. Handle this similarly to the routine convention.

CHECKLIST

Module Quality

- ❑ Does the module have a central purpose?
- ❑ Is the module organized around a common set of data?
- ❑ Does the module offer a cohesive set of services?
- ❑ Are the module's services complete enough that other modules don't have to meddle with its internal data?
- ❑ Is the module independent of other modules? Is it loosely coupled?
- ❑ Does the module hide implementation details from other modules?
- ❑ Are the module's interfaces abstract enough that you don't have to think about how its services are implemented? Can you treat the module as a black box?
- ❑ Have you thought about subdividing the module into component modules, and have you subdivided it as much as you can?
- ❑ If you're working in a language that doesn't fully support modules, have you implemented programming conventions to support them?

Key Points

- The difference between a routine and a module is important, regardless of what you call either one. Think of design in terms of both routines and modules.
- Module data is similar to global data in that it's used by more than one routine. It's different from global data in that the number of routines that use it is strictly limited and you know exactly which routines they are. Consequently, you can use module data without the risks associated with global data.
- Information hiding is always appropriate. It results in reliable, easily modifiable systems. It is also at the core of popular design methods.
- You create modules for many of the same reasons you create routines. The module is a more powerful concept than the routine because it can offer more than one service. Consequently, it's a higher-level design tool than the routine.
- You can implement modules in any language. If your language doesn't support them directly, you can supplement your language with programming conventions to help you achieve a kind of modularity.