

Chapter 7

From Modules to Objects

Chapter Objectives

After studying this chapter, you should be able to

- Design modules and classes with high cohesion and low coupling.
- Understand the need for information hiding.
- Describe the software engineering implications of inheritance, polymorphism, and dynamic binding.
- Distinguish among generalization, aggregation, and association.
- Discuss the object-oriented paradigm in greater depth than before.

Some of the more lurid computer magazines seem to suggest that the object-oriented paradigm was a sudden, dramatic new discovery of the mid-1980s, a revolutionary alternative to the then-popular classical paradigm. That is not the case. Instead, the theory of modularity underwent steady progress during the 1970s and the 1980s, and objects were simply an evolutionary development within the theory of modularity (but see Just in Case You Wanted to Know Box 7.1). This chapter describes objects within the context of modularity.

This approach is taken because it is extremely difficult to use objects correctly without understanding why the object-oriented paradigm is superior to the classical paradigm. And, to do that, it is necessary to appreciate that an object is merely the next logical step in the body of knowledge that begins with the concept of a module.

7.1 What Is a Module?

When a large product consists of a single monolithic block of code, maintenance is a nightmare. Even for the author of such a monstrosity, attempting to debug the code is extremely difficult; for another programmer to understand it is virtually impossible. The solution is to

Just in Case You Wanted to Know

Box 7.1

Object-oriented concepts were introduced as early as 1966 in the simulation language Simula 67 [Dahl and Nygaard, 1966]. However, at that time, the technology was too radical for practical use, so it lay dormant until the early 1980s, when it essentially was reinvented within the context of the theory of modularity.

This chapter includes other examples of the way leading-edge technology lies dormant until the world is ready for it. For example, information hiding (Section 7.6) was first proposed in 1971 within the software context by Parnas [1971], but the technology was not widely adopted until about 10 years later, when encapsulation and abstract data types had become part of software engineering.

We humans seem to adopt new ideas only when we are ready to use them, not necessarily when they are first presented.

break the product into smaller pieces, called *modules*. What is a module? Is the way a product is broken into modules important in itself or is it important only to break a large product into smaller pieces of code?

Stevens, Myers, and Constantine [1974] made an early attempt to describe modules. They defined a **module** as, “A set of one or more contiguous program statements having a name by which other parts of the system can invoke it, and preferably having its own distinct set of variable names.” In other words, a module consists of a single block of code that can be invoked in the way that a procedure, function, or method is invoked. This definition seems to be extremely broad. It includes procedures and functions of all kinds, whether internal or separately compiled. It includes COBOL paragraphs and sections, even though they cannot have their own variables, because the definition states that the property of possessing a distinct set of variable names is merely “preferable.” It also includes modules nested inside other modules. But, broad as it is, the definition does not go far enough. For example, an assembler macro is not invoked and therefore, by the preceding definition, is not a module. In C and C++, a header file of declarations that is **#included** in a product similarly is not invoked. In short, this definition is too restrictive.

Yourdon and Constantine [1979] give a broader definition: “A module is a lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier.” Examples of boundary elements are `begin . . . end` pairs in a block-structured language like Pascal or `{ . . }` pairs in C++ or Java. This definition not only includes all the cases excluded by the previous definition but is broad enough to be used throughout this book. In particular, procedures and functions of the classical paradigm are modules. In the object-oriented paradigm, an object is a module and so is a method within an object.

To understand the importance of modularization, consider the following somewhat fanciful example. John Fence is a highly incompetent computer architect. He still has not discovered that both NAND gates and NOR gates are complete; that is, every circuit can be built with only NAND gates or with only NOR gates. John therefore decides to build an ALU, shifter, and 16 registers using AND, OR, and NOT gates. The resulting computer is shown in Figure 7.1. The three components are connected in a simple fashion. Now, our architect friend decides that the circuit should be fabricated on three silicon chips, so he designs the three chips shown in Figure 7.2. One chip has all the gates of the ALU, a second contains the shifter, and the third is for the registers. At this point John vaguely recalls that someone in a bar told him that it is best to build chips so that they have only one kind

FIGURE 7.1 The design of a computer.

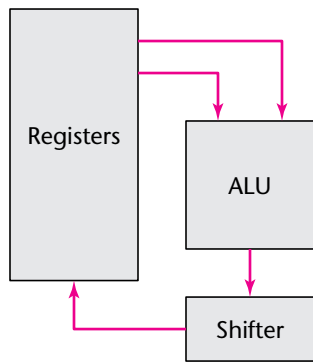


FIGURE 7.2 The computer of Figure 7.1 fabricated on three chips.

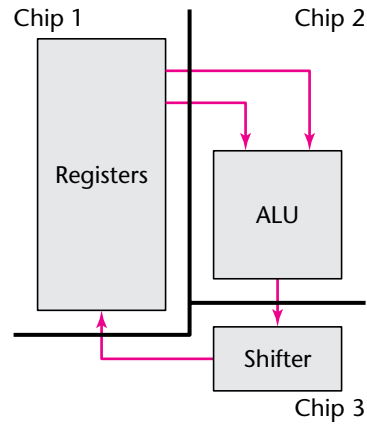
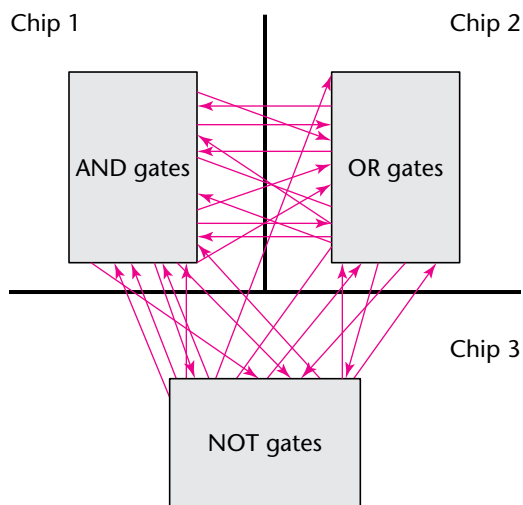


FIGURE 7.3 The computer of Figure 7.1 fabricated on three other chips.



of gate, so he redesigns his chips. On chip 1 he puts all the AND gates, on chip 2 all the OR gates, and all the NOT gates go onto chip 3. The resulting “work of art” is shown schematically in Figure 7.3.

Figures 7.2 and 7.3 are functionally equivalent; that is, they do exactly the same thing. But the two designs have markedly different properties:

1. Figure 7.3 is considerably harder to *understand* than Figure 7.2. Almost anyone with a knowledge of digital logic immediately knows that the chips in Figure 7.2 form an ALU, a shifter, and a set of registers. However, even a leading hardware expert would have trouble understanding the function of the various AND, OR, and NOT gates in Figure 7.3.

2. *Corrective maintenance* of the circuits shown in Figure 7.3 is difficult. Should the computer have a design fault—and anyone capable of coming up with Figure 7.3 is undoubtedly going to make lots and lots of mistakes—it would be difficult to determine where the fault is located. On the other hand, if the design of the computer in Figure 7.2 has a fault, it can be localized by determining whether it appears to be in the way the ALU works, the way the shifter works, or the way the registers work. Similarly, if the computer of Figure 7.2 breaks down, it is relatively easy to determine which chip to replace; if the computer in Figure 7.3 breaks down, it is probably best to replace all three chips.
3. The computer of Figure 7.3 is difficult to *extend* or *enhance*. If a new type of ALU is needed or faster registers are required, it is back to the drawing board. But the design of the computer of Figure 7.2 makes it easy to replace the appropriate chip. Perhaps worst of all, the chips of Figure 7.3 cannot be *reused* in any new product. There is no way that those three specific combinations of AND, OR, and NOT gates can be utilized for any product other than the one for which they were designed. In all probability, the three chips of Figure 7.2 can be reused in other products that require an ALU, a shifter, or registers.

The point here is that software products have to be designed to look like Figure 7.2, where there is a maximal relationship within each chip and a minimal relationship between chips. A module can be likened to a chip, in that it performs an operation or series of operations and is connected to other modules. The functionality of the product as a whole is fixed; what has to be determined is how to break the product into modules. Composite/structured design [Stevens, Myers, and Constantine, 1974] provides a rationale for breaking a product into modules as a way to reduce the cost of maintenance, the major component of the total software budget, as pointed out in Chapter 1. The maintenance effort, whether corrective, perfective, or adaptive, is reduced when there is maximal interaction within each module and minimal interaction between modules. In other words, the aim of composite/structured design (C/SD) is to ensure that the module decomposition of the product resembles Figure 7.2 rather than Figure 7.3.

Myers [1978b] quantified the ideas of module **cohesion**, the degree of interaction within a module, and module **coupling**, the degree of interaction between two modules. To be more precise, Myers used the term **strength** rather than *cohesion*. However, *cohesion* is preferable because modules can have high strength or low strength, and something is inherently contradictory in the expression *low strength*—something that is not strong is weak. To prevent terminological inexactitude, C/SD now uses the term *cohesion*. Some authors have used the term **binding** in place of *coupling*. Unfortunately, *binding* also is used in other contexts in computer science, such as binding values to variables. But *coupling* has none of these overtones and therefore is preferable.

It is necessary at this point to distinguish between the operation of a module, the logic of a module, and the context of a module. The **operation** of a module is what it does, that is, its behavior. For example, the operation of module *m* is to compute the square root of its argument. The **logic** of a module is how the module performs its operation; in the case of module *m*, the specific way of computing the square root is Newton's method [Gerald and Wheatley, 1999]. The **context** of a module is the specific use of that module. For example,

module `m` is used to compute the square root of a double precision integer. A key point in C/SD is that the name assigned a module is its operation and not its logic or its context. Therefore, in C/SD, module `m` should be named `compute_square_root`,¹ its logic and its context are irrelevant from the viewpoint of its name.

7.2 Cohesion

Myers [1978b] defined seven categories or levels of cohesion. In the light of modern theoretical computer science, Myers's first two levels need to be interchanged because, as will be shown, informational cohesion supports reuse more strongly than functional cohesion. The resulting ranking is shown in Figure 7.4. This is not a linear scale of any sort. It is merely a relative ranking, a way of determining which types of cohesion are high (good) and which are low (bad).

To understand what constitutes a module with high cohesion, it is necessary to start at the other end and consider the lower cohesion levels.

7.2.1 Coincidental Cohesion

A module has **coincidental cohesion** if it performs multiple, completely unrelated operations. An example of a module with coincidental cohesion is a module named `print_the_next_line`, `reverse_the_string_of_characters_comprising_the_second_argument`, `add_7_to_the_fifth_argument`, `convert_the_fourth_argument_to_floating_point`. An obvious question is, How can such modules possibly arise in practice? The most common cause is as a consequence of rigidly enforcing rules such as “every module shall consist of between 35 and 50 executable statements.” If a software organization insists that modules must be neither too big nor too small, then two undesirable things happen. First, two or more otherwise ideal smaller modules are lumped together to create a larger module with coincidental cohesion. Second, pieces hacked from well-designed modules that management considers too large are combined, again resulting in modules with coincidental cohesion.

FIGURE 7.4
Levels of cohesion.

7.	Informational cohesion	(Good)
6.	Functional cohesion	
5.	Communicational cohesion	
4.	Procedural cohesion	
3.	Temporal cohesion	
2.	Logical cohesion	
1.	Coincidental cohesion	(Bad)

¹ For added clarity, the underscore is used in function names like `compute_square_root` to highlight that the structured paradigm is used in this and the following sections. When the object-oriented paradigm is used (from Section 7.4.2 onward), the corresponding method would be named `computeSquareRoot`.

Why is coincidental cohesion so bad? Modules with coincidental cohesion suffer from two serious drawbacks. First, such modules degrade the maintainability of the product, both corrective maintenance and enhancement. From the viewpoint of trying to understand a product, modularization with coincidental cohesion is worse than no modularization at all [Shneiderman and Mayer, 1975]. Second, these modules are not reusable. It is extremely unlikely that the module with coincidental cohesion in the first paragraph of this section could be reused in any other product.

Lack of reusability is a serious drawback. The cost of building software is so great that it is essential to try to reuse modules wherever possible. Designing, coding, documenting, and above all, testing a module are time-consuming and hence costly processes. If an existing well-designed, thoroughly tested, and properly documented module can be used in another product, then management should insist that the existing module be reused. But there is no way that a module with coincidental cohesion can be reused, and the money spent to develop it can never be recouped. (Reuse is discussed in detail in Chapter 8.)

It is generally easy to rectify a module with coincidental cohesion—because it performs multiple operations, break the module into smaller modules that each perform one operation.

7.2.2 Logical Cohesion

A module has **logical cohesion** when it performs a series of related operations, one of which is selected by the calling module. All the following are examples of modules with logical cohesion.

Example 1 Module `new_operation`, which is invoked as follows:

```
function_code = 7;
new_operation (function_code, dummy_1, dummy_2, dummy_3);
// dummy_1, dummy_2, and dummy_3 are dummy variables,
// not used if function_code is equal to 7
```

In this example, `new_operation` is called with four arguments, but as stated in the comment lines, three of them are not needed if `function_code` is equal to 7. This degrades readability, with the usual implications for maintenance, both corrective and enhancement.

Example 2 An object that performs all input and output.

Example 3 A module that edits insertions, deletions, and modifications of master file records.

Example 4 A module with logical cohesion in an early version of OS/VS2 that performed 13 different operations; its interface contained 21 pieces of data [Myers, 1978b].

Two problems occur when a module has logical cohesion. First, the interface is difficult to understand (Example 1 is a case in point), and comprehensibility of the module as a whole may suffer as a result. Second, the code for more than one operation may be intertwined, leading to severe maintenance problems. For instance, a module that performs all input and output may be structured as shown in Figure 7.5. If a new tape unit is installed, it may be necessary to modify the sections numbered 1, 2, 3, 4, 6, 9, and 10. These changes may adversely affect other forms of input–output, such as laser printer output, because the laser printer is affected by changes to sections 1 and 3. This intertwined property is characteristic of modules with logical cohesion. A further consequence of intertwining is that it is difficult to reuse such a module in other products.

FIGURE 7.5
A module that performs all input and output.

1. Code for all input and output
2. Code for input only
3. Code for output only
4. Code for disk and tape I/O
5. Code for disk I/O
6. Code for tape I/O
7. Code for disk input
8. Code for disk output
9. Code for tape input
10. Code for tape output
⋮ ⋮ ⋮
37. Code for keyboard input

7.2.3 Temporal Cohesion

A module has **temporal cohesion** when it performs a series of operations related in time. An example of a module with temporal cohesion is one named `open_old_master_file`, `new_master_file`, `transaction_file`, and `print_file`; `initialize_sales_region_table`; `read_first_transaction_record_and_first_old_master_file_record`. In the bad old days before C/SD, such a module would be called `perform_initialization`.

The operations of this module are related weakly to one another but more strongly to operations in other modules. Consider, for example, the `sales_region_table`. It is initialized in this module, but operations such as `update_sales_region_table` and `print_sales_region_table` are located in other modules. Therefore, if the structure of the `sales_region_table` is changed, perhaps because the organization is expanding into areas of the country where it previously had not done business, a number of modules have to be changed. Not only is there more chance of a regression fault (a fault caused by a change made to an apparently unrelated part of the product), but if the number of affected modules is large, one or two modules are likely to be overlooked. It is much better to have all the operations on the `sales_region_table` in one module, as described in Section 7.2.7. These operations then can be invoked, when needed, by other modules. In addition, a module with temporal cohesion is unlikely to be reusable in a different product.

7.2.4 Procedural Cohesion

A module has **procedural cohesion** if it performs a series of operations related by the sequence of steps to be followed by the product. An example of a module with procedural cohesion is `read_part_number_from_database_and_update_repair_record_on_maintenance_file`.

This clearly is better than temporal cohesion—at least the operations are related procedurally to one another. Even so, the operations are still weakly connected, and again the module is unlikely to be reusable in another product. The solution is to break a module with procedural cohesion into separate modules, each performing one operation.

7.2.5 Communicational Cohesion

A module has **communicational cohesion** if it performs a series of operations related by the sequence of steps to be followed by the product and if all the operations are performed on the same data. Two examples of modules with communicational cohesion are `update_record_in_database_and_write_it_to_the_audit_trail`, and `calculate_new_trajectory_and_send_it_to_the_printer`. This is better than procedural cohesion because the operations of the module are more closely connected, but it still has the same drawback as coincidental, logical, temporal, and procedural cohesion. The module cannot be reused. Again the solution is to break such a module into separate modules, each performing one operation.

In passing, it is interesting to note that Dan Berry [personal communication, 1978] uses the term **flowchart cohesion** to refer to temporal, procedural, and communicational cohesion, because the operations performed by such modules are adjacent in the product flowchart. The operations are adjacent in the case of temporal cohesion because they are performed at the same time. They are adjacent in procedural cohesion because the algorithm requires the operations to be performed in series. They are adjacent in communicational cohesion because, in addition to being performed in series, the operations are performed on the same data, and therefore it is natural that these operations should be adjacent in the flowchart.

7.2.6 Functional Cohesion

A module that performs exactly one operation or achieves a single goal has **functional cohesion**. Examples of such modules are `get_temperature_of_furnace`; `compute_orbital_of_electron`; `write_to_diskette`; and `calculate_sales_commission`.

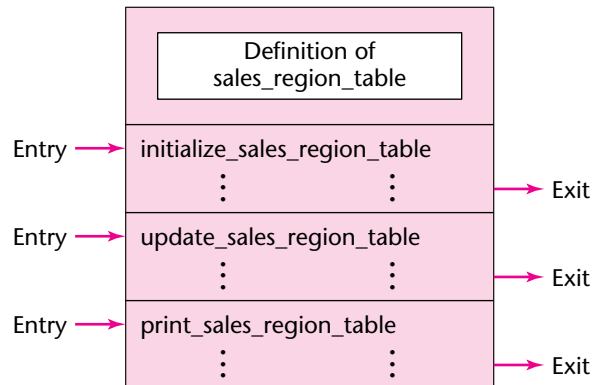
A module with functional cohesion often can be reused because the one operation it performs often needs to be performed in other products. A properly designed, thoroughly tested, and well-documented module with functional cohesion is a valuable (economic and technical) asset to any software organization and should be reused as often as possible (but see Section 8.4).

Maintenance is easier to perform on a module with functional cohesion. First, functional cohesion leads to fault isolation. If it is clear that the temperature of the furnace is not being read correctly, then the fault almost certainly is in module `get_temperature_of_furnace`. Similarly, if the orbital of an electron is computed incorrectly, then the first place to look is in `compute_orbital_of_electron`.

Once the fault has been localized to a single module, the next step is to make the required changes. Because a module with functional cohesion performs only one operation, such a module generally is easier to understand than a module with lower cohesion. This ease in understanding also simplifies the maintenance. Finally, when the change is made, the chance of that change affecting other modules is slight, especially if the coupling between modules is low (Section 7.3).

Functional cohesion also is valuable when a product has to be extended. For example, suppose that a personal computer has a 10 gigabyte hard drive but the manufacturer now wishes to market a more powerful model of the computer with a 30 gigabyte hard drive instead. Reading through the list of modules, the maintenance programmer finds a module named `write_to_hard_drive`. The obvious thing to do is to replace that module with a new one called `write_to_larger_hard_drive`.

FIGURE 7.6
A module with
informational
cohesion.



In passing, it should be pointed out that the three “modules” of Figure 7.2 have functional cohesion, and the arguments made in Section 7.1 for favoring the design of Figure 7.2 over that of Figure 7.3 are precisely those made in the preceding discussion for favoring functional cohesion.

7.2.7 Informational Cohesion

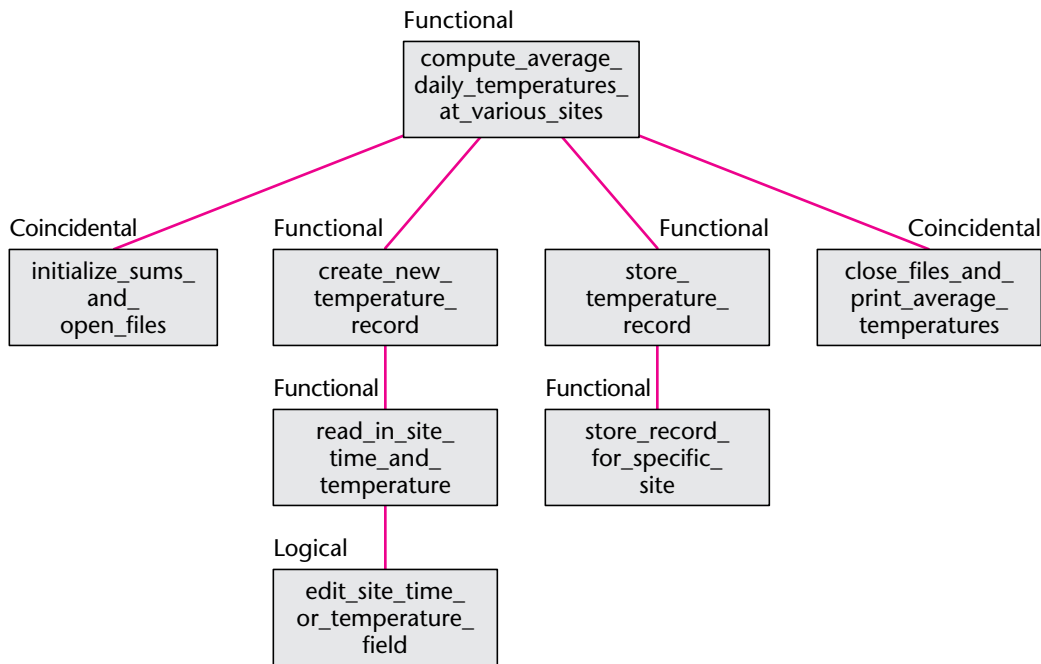
A module has **informational cohesion** if it performs a number of operations, each with its own entry point, with independent code for each operation, all performed on the same data structure. An example is given in Figure 7.6. This does not violate the tenets of structured programming; each piece of code has exactly one entry point and one exit point. A major difference between logical cohesion and informational cohesion is that the various operations of a module with logical cohesion are intertwined, whereas in a module with informational cohesion the code for each operation is completely independent.

A module with informational cohesion essentially is an implementation of an abstract data type, as explained in Section 7.5, and all the advantages of using an abstract data type are gained when a module with informational cohesion is used. Because an object essentially is an instantiation (instance) of an abstract data type (Section 7.7), an object, too, is a module with informational cohesion.²

7.2.8 Cohesion Example

For further insight into cohesion, consider the example shown in Figure 7.7. Two modules in particular merit comment. It may seem somewhat surprising that the modules `initialize_sums_and_open_files` and `close_files_and_print_average_temperatures` have been labeled as having coincidental cohesion rather than temporal cohesion. First, consider module `initialize_sums_and_open_files`. It performs two operations related in time, in that both have to be done before any calculations can be performed, and therefore it seems that the module has temporal cohesion. Although the two operations of `initialize_sums_and_open_files` indeed are performed at the beginning of the calculation, another factor is involved. Initializing the sums is related to the problem, but opening files

² The discussion in this paragraph assumes that the abstract data type or object is well designed. If the methods of an object perform completely unrelated operations, then the object has coincidental cohesion.

FIGURE 7.7 A module interconnection diagram showing the cohesion of each module.

is a hardware issue that has nothing to do with the problem itself. The rule when two or more levels of cohesion can be assigned to a module is to assign the lowest possible level. Thus, because `initialize_sums_and_open_files` could have either temporal or coincidental cohesion, the lower of the two levels of cohesion (coincidental) is assigned that module. That also is the reason why `close_files_and_print_average_temperatures` has coincidental cohesion.

7.3 Coupling

Recall that cohesion is the degree of interaction within a module. Coupling is the degree of interaction between two modules. As before, a number of levels can be distinguished, as shown in Figure 7.8. To highlight good coupling, the various levels are described in order from the worst to the best.

FIGURE 7.8

Levels of coupling.

5.	Data coupling	(Good)
4.	Stamp coupling	
3.	Control coupling	
2.	Common coupling	
1.	Content coupling	(Bad)

7.3.1 Content Coupling

Two modules are **content coupled** if one directly references the contents of the other. All the following are examples of content coupling:

Example 1. Module *p* modifies a statement of module *q*. This practice is not restricted to assembly language programming. The **alter** verb, now mercifully removed from COBOL, did precisely that: It modified another statement.

Example 2. Module *p* refers to local data of module *q* in terms of some numerical displacement within *q*.

Example 3. Module *p* branches to a local label of module *q*.

Suppose that module *p* and module *q* are content coupled. One of the many dangers is that almost any change to *q*, even recompiling *q* with a new compiler or assembler, requires a change to *p*. Furthermore, it is impossible to reuse module *p* in some new product without reusing module *q* as well. When two modules are content coupled, they are inextricably interlinked.

7.3.2 Common Coupling

Two modules are **common coupled** if both have access to the same global data. The situation is depicted in Figure 7.9. Instead of communicating with one another by passing arguments, modules *cca* and *ccb* can access and change the value of *global_variable*. The most common situation in which this arises is when both *cca* and *ccb* have access to the same database and can read and write the same record. For common coupling, it is necessary that both modules can read *and* write to the database; if the database access mode is read-only, then this is not common coupling. But there are other ways of implementing common coupling, including use of the C++ or Java modifier **public**.

This form of coupling is undesirable for a number of reasons:

1. It contradicts the spirit of structured programming in that the resulting code is virtually unreadable. Consider the pseudocode fragment shown in Figure 7.10. If *global_variable* is a global variable, then its value may be changed by *module_3*, *module_4*, or any module called by them. Determining under what conditions the loop terminates is a nontrivial question; if a run-time failure occurs, it may be difficult to reconstruct what happened, because any of a number of modules could have changed the value of *global_variable*.

FIGURE 7.9 An example of common coupling.

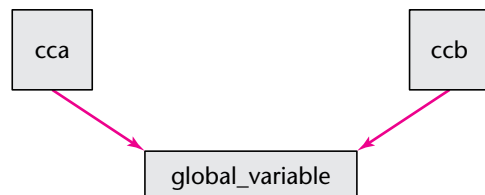


FIGURE 7.10 A pseudocode fragment reflecting common coupling.

```

while (global_variable == 0)
{
    if (argument_xyz > 25)
        module_3 ();
    else
        module_4 ();
}
  
```

2. Consider the call `edit_this_transaction(record_7)`. If there is common coupling, this call could change not just the value of `record_7` but any global variable that can be accessed by that module. In short, the entire module must be read to find out precisely what it does.
3. If a maintenance change is made in one module to the declaration of a global variable, then every module that can access that global variable has to be changed. Furthermore, all changes must be consistent.
4. Another problem is that a common-coupled module is difficult to reuse because the identical list of global variables has to be supplied each time the module is reused.
5. Common coupling possesses the unfortunate property that the number of instances of common coupling between a module *p* and the other modules in a product can change drastically, even if module *p* itself never changes; this is termed **clandestine common coupling** [Schach et al., 2003a]. For example, if both module *p* and module *q* can modify global variable *g_v*, then there is one instance of common coupling between module *p* and the other modules in the software product. But if 10 new modules are designed and implemented, all of which can modify global variable *g_v*, then the number of instances of common coupling between module *p* and the other modules increases to 11, even though module *p* itself has not been changed in any way. Clandestine common coupling can have surprising consequences. For example, between 1993 and 2000, there were nearly 400 releases of Linux; 5332 versions of the 17 Linux kernel modules were unchanged between successive releases. In more than half of the 5332 versions, the number of instances of common coupling between each kernel module and the rest of Linux increased or decreased, even though the kernel module itself did not change. Considerably more modules exhibited clandestine common coupling in an upward direction (2482) than downward (379) [Schach et al., 2003a]. The number of lines of code in the Linux kernel grew linearly with version number, but the number of instances of common coupling grew exponentially [Schach et al., 2002]. It seems inevitable that, at some future date, the dependencies between modules induced by clandestine common coupling will render Linux extremely hard to maintain. It will then be exceedingly hard to change one part of Linux without inducing a regression fault (an apparently unrelated fault) elsewhere in the product.
6. This problem is potentially the most dangerous. As a consequence of common coupling, a module may be exposed to more data than it needs. This defeats any attempts to control data access and ultimately may lead to computer crime. Many types of computer crime need some form of collusion. Properly designed software should not allow any one programmer access to all the data and modules needed to commit a crime. For example, a programmer writing the check printing part of a payroll product needs to have access to employee records; but, in a well-designed product, such access is exclusively in read-only mode, preventing the programmer from making unauthorized changes to his or her monthly salary. To make such changes, the programmer has to find another dishonest employee, one with access to the relevant records in update mode. But, if the product has been badly designed and every module can access the payroll database in update mode, then an unscrupulous programmer acting alone can make unauthorized changes to any record in the database.

Although we hope that these arguments will dissuade all but the most daring of readers from using common coupling, in some situations, common coupling might seem to be preferable to the alternatives. Consider, for example, a product that performs computer-aided design of petroleum storage tanks [Schach and Stevens-Guille, 1979]. A tank is specified by a large number of descriptors such as height, diameter, maximum wind speed to which the tank will be subjected, and insulation thickness. The descriptors have to be initialized but do not change in value thereafter, and most of the modules in the product need access to the values of the descriptors. Suppose that there are 55 tank descriptors. If all these descriptors are passed as arguments to every module, then the interface to each module will consist of at least 55 arguments and the potential for faults is huge. Even in a language like Ada, which requires strict type checking of arguments, two arguments of the same type still can be interchanged, a fault that would not be detected by a type checker.

One solution is to put all the tank descriptors in a database and design the product in such a way that one module initializes the values of all the descriptors, whereas all the other modules access the database exclusively in read-only mode. However, if the database solution is impractical, perhaps because the specified implementation language cannot be interfaced with the available database management system, then an alternative is to use common coupling but in a controlled way. That is, the product should be designed so that the 55 descriptors are initialized by one module, but none of the other modules changes the value of a descriptor. This programming style has to be enforced by management, unlike the database solution, where enforcement is imposed by the software. Therefore, in situations where there is no good alternative to the use of common coupling, close supervision by management can reduce some of the risks. A better solution, however, is to obviate common coupling by using information hiding, as described in Section 7.6.

7.3.3 Control Coupling

Two modules are **control coupled** if one passes an element of control to the other module; that is, one module explicitly controls the logic of the other. For example, control is passed when a function code is passed to a module with logical cohesion (Section 7.2.2). Another example of control coupling is when a control switch is passed as an argument.

If module *p* calls module *q* and *q* passes back a flag to *p* that says, “I am unable to complete my task,” then *q* is passing *data*. But if the flag means, “I am unable to complete my task; accordingly, write error message ABC123,” then *p* and *q* are control coupled. In other words, if *q* passes information back to *p* and *p* decides what action to take as a consequence of receiving that information, then *q* is passing data. But, if *q* not only passes back information but also informs module *p* as to what action *p* must take, then control coupling is present.

The major difficulty that arises as a consequence of control coupling is that the two modules are not independent; module *q*, the called module, has to be aware of the internal structure and logic of module *p*. As a result, the possibility of reuse is reduced. In addition, control coupling generally is associated with modules that have logical cohesion and includes the difficulties associated with logical cohesion.

7.3.4 Stamp Coupling

In some programming languages, only simple variables, such as `part_number`, `satellite_altitude`, or `degree_of_multiprogramming`, can be passed as arguments. But

Just in Case You Wanted to Know

Box 7.2

Passing four or five different fields to a module may be slower than passing a complete record. This situation leads to a larger issue: What should be done when optimization issues (such as response time or space constraints) clash with what is generally considered to be good software engineering practice?

In my experience, this question frequently turns out to be irrelevant. The recommended approach may slow down the response time, but by only a millisecond or so, far too small to be detected by users. Therefore, in accordance with Knuth's [1974] First Law of Optimization: *Don't!*—rarely is there a need for optimization of any kind, including for performance reasons.

But what if optimization really is required? In this case, Knuth's Second Law of Optimization applies. The Second Law (labeled *for experts only*) is *Not yet!* In other words, first complete the entire product using appropriate software engineering techniques. Then, if optimization really is required, make only the necessary changes, meticulously documenting what is being changed and why. If at all possible, this optimization should be done by an experienced software engineer.

many languages also support passing data structures, such as records or arrays, as arguments. In such languages, valid arguments include `part_record`, `satellite_coordinates`, or `segment_table`. Two modules are **stamp coupled** if a data structure is passed as an argument, but the called module operates on only some of the individual components of that data structure.

Consider, for example, the call `calculate_withholding (employee_record)`. It is not clear, without reading the entire `calculate_withholding` module, which fields of the `employee_record` the module accesses or changes. Passing the employee's salary obviously is essential for computing the withholding, but it is difficult to see how the employee's home telephone number is needed for this purpose. Instead, only those fields that it actually needs for computing the withholding should be passed to module `calculate_withholding`. Not only is the resulting module, and particularly its interface, easier to understand, it is likely to be reusable in a variety of other products that also need to compute withholding. (See Just in Case You Wanted to Know Box 7.2 for another perspective on this.)

Perhaps even more important, because the call `calculate_withholding (employee_record)` passes even more data than strictly necessary, the problems of uncontrolled data access, and conceivably computer crime, once again arise. This issue is discussed in Section 7.3.2.

Nothing is at all wrong with passing a data structure as an argument, provided all the components of the data structure are used by the called module. For example, calls like `invert_matrix (original_matrix, inverted_matrix)` or `print_inventory_record (warehouse_record)` pass a data structure as an argument, but the called modules operate on all the components of that data structure. Stamp coupling is present when a data structure is passed as an argument but only some of the components are used by the called module.

A subtle form of stamp coupling can occur in languages like C or C++ when a pointer to a record is passed as an argument. Consider the call `check_altitude (pointer_to_position_record)`. At first sight, what is being passed is a simple variable. But the called module has access to all of the fields in the `position_record` pointed to by `pointer_to_position_record`. Because of the potential problems, it is a good idea to examine the coupling closely whenever a pointer is passed as an argument.

7.3.5 Data Coupling

Two modules are **data coupled** if all arguments are homogeneous data items. That is, every argument is either a simple argument or a data structure in which all elements are used by the called module. Examples include `display_time_of_arrival` (`flight_number`), `compute_product` (`first_number`, `second_number`, `result`), and `determine_job_with_highest_priority` (`job_queue`).

Data coupling is a desirable goal. To put it in a negative way, if a product exhibits data coupling exclusively, then the difficulties of content, common, control, and stamp coupling are not present. From a more positive viewpoint, if two modules are data coupled, then maintenance is easier, because a change to one module is less likely to cause a regression fault in the other. The following example clarifies certain aspects of coupling.

7.3.6 Coupling Example

Consider the example shown in Figure 7.11. The numbers on the arcs represent interfaces that are defined in greater detail in Figure 7.12. For example, when module `p` calls module `q` (interface 1), it passes one argument, the type of the aircraft. When `q` returns control to `p`, it passes back a status flag. Using the information in Figures 7.11 and 7.12, the coupling between every pair of modules can be deduced. The results are shown in Figure 7.13.

Some of the entries in Figure 7.13 are obvious. For instance, the data coupling between `p` and `q` (interface 1 in Figure 7.11), between `r` and `t` (interface 5), and between `s` and `u` (interface 6) are a direct consequence of the fact that a simple variable is passed in each direction. The coupling between `p` and `s` (interface 2) is data coupling if all the elements of

FIGURE 7.11
Module interconnection diagram for a coupling example.

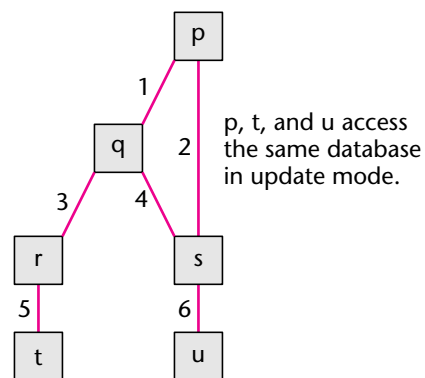


FIGURE 7.12
Interface description for Figure 7.11.

Number	In	Out
1	aircraft_type	status_flag
2	list_of_aircraft_parts	—
3	function_code	—
4	list_of_aircraft_parts	—
5	part_number	part_manufacturer
6	part_number	part_name

FIGURE 7.13
Coupling
between pairs of
modules of
Figure 7.11.

	q	r	s	t	u
p	Data	—	{ Data or stamp	Common	Common
q		Control	{ Data or stamp	—	—
r			—	Data	—
s				—	Data
t					Common

the list of parts passed from *p* to *s* are used or updated, but it is stamp coupling if *s* operates on only certain elements of the list. The coupling between *q* and *s* (interface 4) is similar. Because the information in Figures 7.11 and 7.12 does not completely describe the function of the various modules, there is no way of determining whether the coupling is data or stamp. The coupling between *q* and *r* (interface 3) is control coupling, because a function code is passed from *q* to *r*.

Perhaps somewhat surprising are the three entries marked common coupling in Figure 7.13. The three module pairs that are farthest apart in Figure 7.11—*p* and *t*, *p* and *u*, and *t* and *u*—at first appear not to be coupled in any way. After all, no interface connects them, so the very idea of coupling between them, let alone common coupling, requires some explanation. The answer lies in the annotation on the right-hand side of Figure 7.11, namely, that *p*, *t*, and *u* all access the same database in update mode. The result is that a number of global variables can be changed by all three modules, and hence they are pairwise common coupled.

7.3.7 The Importance of Coupling

Coupling is an important metric. If module *p* is tightly coupled to module *q*, then a change to module *p* may require a corresponding change to module *q*. If this change is made, as required, during integration or postdelivery maintenance, then the resulting product functions correctly; however, progress at that stage is slower than would have been the case had the coupling been looser. On the other hand, if the required change is not made to module *q* at that time, then the fault manifests itself later. In the best case, the compiler or linker informs the team right away that something is amiss or a failure will occur while testing the change to module *p*. What usually happens, however, is that the product fails either during subsequent integration testing or after the product has been installed on the client's computer. In both cases, the failure occurs after the change to module *p* has been completed. There no longer is any apparent link between the change to module *p* and the overlooked corresponding change to module *q*. The fault therefore may be hard to find.

Given that a design in which modules have high cohesion and low coupling is a good design, the obvious question is, How can such a design be achieved? Because this chapter is devoted to theoretical concepts surrounding design, the answer to the question is presented in Chapter 13. In the meantime, those qualities that identify a good design are examined further and refined. For convenience, the key definitions in this chapter appear in Figure 7.14, together with the section in which each definition appears.

FIGURE 7.14
Key definitions
of this chapter,
and the sections
in which they
appear.

Abstract data type: a data type together with the operations performed on instantiations of that data type (Section 7.5)

Abstraction: a means of achieving stepwise refinement by suppressing unnecessary details and accentuating relevant details (Section 7.4.1)

Class: an abstract data type that supports inheritance (Section 7.7)

Cohesion: the degree of interaction within a module (Section 7.1)

Coupling: the degree of interaction between two modules (Section 7.1)

Data encapsulation: a data structure together with the operations performed on that data structure (Section 7.4)

Encapsulation: the gathering together into one unit of all aspects of the real-world entity modeled by that unit (Section 7.4.1)

Information hiding: structuring the design so that the resulting implementation details are hidden from other modules (Section 7.6)

Object: an instantiation of a class (Section 7.7)

7.4 Data Encapsulation

Consider the problem of designing an operating system for a large mainframe computer. According to the specifications, any job submitted to the computer is classified as high priority, medium priority, or low priority. The task of the operating system is to decide which job to load into memory next, which of the jobs in memory gets the next time slice and how long that time slice should be, and which of the jobs that require disk access has highest priority. In performing this scheduling, the operating system must consider the priority of each job; the higher is the priority, the sooner that job should be assigned the resources of the computer. One way of achieving this is to maintain separate job queues for each job-priority level. The job queues have to be initialized, and facilities must exist for adding a job to a job queue when the job requires memory, CPU time, or disk access as well as for removing a job from a queue when the operating system decides to allocate the required resource to that job.

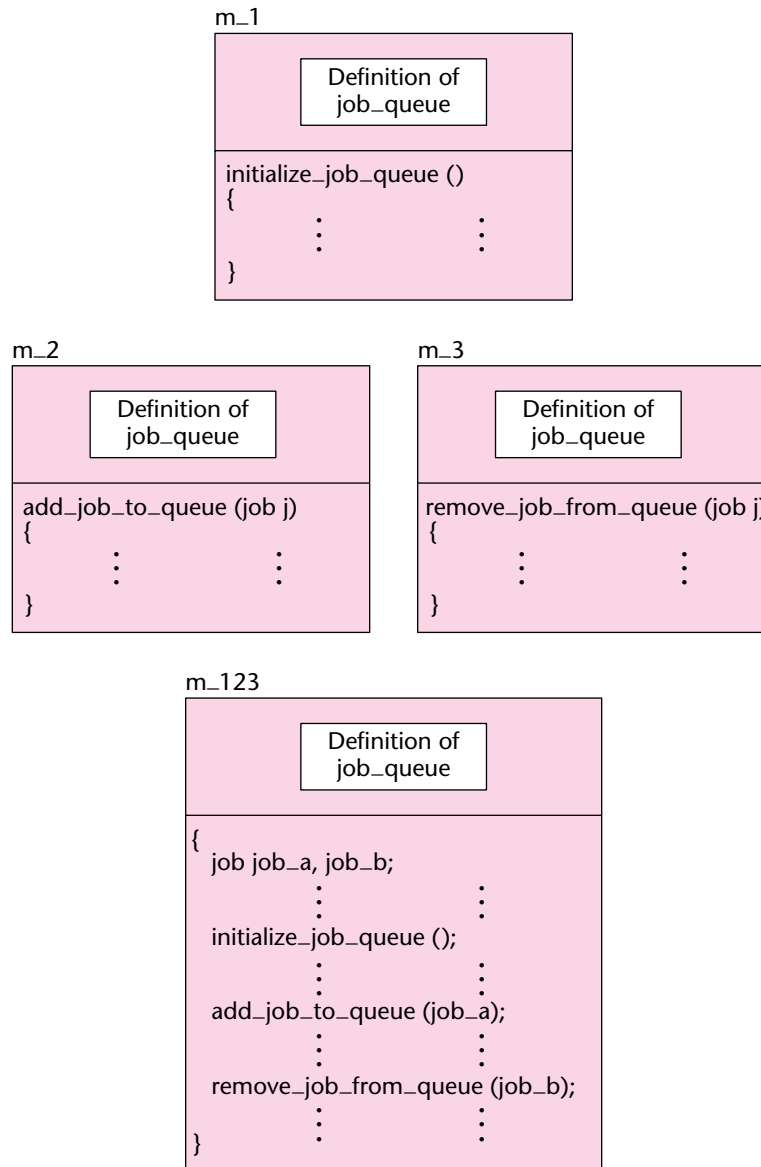
To simplify matters, consider the restricted problem of batch jobs queuing up for memory access. There are three queues for incoming batch jobs, one for each priority level. When submitted by a user, a job is added to the appropriate queue; and when the operating system decides that a job is ready to be run, it is removed from its queue and memory is allocated to it.

This portion of the product can be built in a number of different ways. One possible design, shown in Figure 7.15, depicts modules for manipulating one of the three job queues. A C-like pseudocode is used to highlight some of the problems that can arise in the classical paradigm. Later in this chapter, these problems are solved using the object-oriented paradigm.

Consider Figure 7.15. Function `initialize_job_queue` in module `m_1` is responsible for the initialization of the job queue, and functions `add_job_to_queue` and `remove_job_from_queue` in modules `m_2` and `m_3`, respectively, are responsible for the addition and deletion of jobs. Module `m_123` contains invocations of all three functions in order to manipulate the job queue. To concentrate on data encapsulation, issues such as underflow

FIGURE 7.15

One possible design of the job queue portion of the operating system.



(trying to remove a job from an empty queue) and overflow (trying to add a job to a full queue) have been suppressed here, as well as in the remainder of this chapter.

The modules of the design of Figure 7.15 have low cohesion, because operations on the job queue are spread all over the product. If a decision is made to change the way `job_queue` is implemented (for example, as a linked list of records instead of as a linear list), then modules `m_1`, `m_2`, and `m_3` have to be drastically revised. Module `m_123` also has to be changed; at the very least, the data structure definition has to be changed.

FIGURE 7.16

A design of the job queue portion of the operating system using data encapsulation.

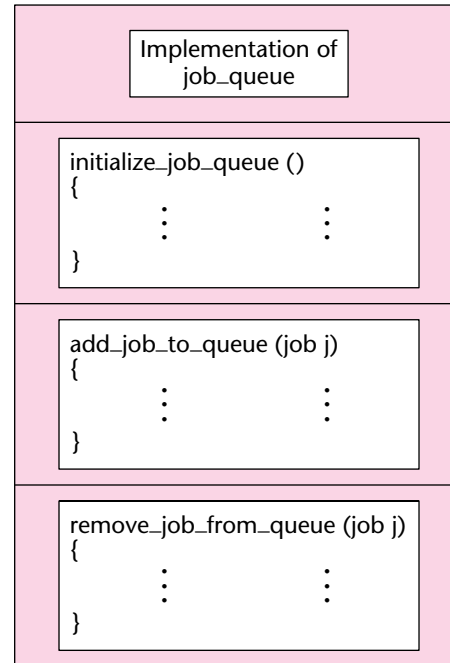
m_123

```

{
    job job_a, job_b;
        :           :
    initialize_job_queue ();
        :           :
    add_job_to_queue (job_a);
        :           :
    remove_job_from_queue (job_b);
        :           :
}

```

m_encapsulation



Now suppose that the design of Figure 7.16 is chosen instead. The module on the right-hand side of the figure has informational cohesion (Section 7.2.7), in that it performs a number of operations on the same data structure. Each operation has its own entry point and exit point and independent code. Module `m_encapsulation` in Figure 7.16 is an implementation of **data encapsulation**; that is, a data structure, in this case the job queue, together with the operations to be performed on that data structure.

An obvious question to ask at this point is, What is the advantage of designing a product using data encapsulation? This will be answered in two ways, from the viewpoint of development and from the viewpoint of maintenance.

7.4.1 Data Encapsulation and Development

Data encapsulation is an example of **abstraction**. Returning to the job queue example, a data structure (the job queue) has been defined, together with three associated operations (initialize the job queue, add a job to the queue, and delete a job from the queue). The developer can conceptualize the problem at a higher level, the level of jobs and job queues, rather than at the lower level of records or arrays.

The basic theoretical concept behind abstraction, once again, is stepwise refinement. First, a design for the product is produced in terms of high-level concepts such as jobs, job queues, and the operations performed on job queues. At this stage, it is entirely irrelevant how the job queue is implemented. Once a complete high-level design has been obtained, the second step is to design the lower-level components in terms of which the data structure

and operations on the data structure are implemented. In C, for example, the data structure (the job queue) is implemented in terms of records (structures) or arrays; the three operations (initialize the job queue, add a job to the queue, and remove a job from the queue) are implemented as functions. The key point is that, while this lower level is being designed, the designer totally ignores the intended use of the jobs, job queue, and operations. Therefore, during the first step, the existence of the lower level is assumed, even though at this stage no thought has been given to that level; during the second step (the design of the lower level), the existence of the higher level is ignored. At the higher level, the concern is with the behavior of the data structure, the job queue; at the lower level, the implementation of that behavior is the primary concern. Of course, a larger product has many levels of abstraction.

Different types of abstraction exist. Consider Figure 7.16. The figure has two types of abstraction. Data encapsulation (that is, a data structure together with the operations to be performed on that data structure) is an example of **data abstraction**; the C functions themselves are an example of **procedural abstraction**. Abstraction, to summarize, simply is a means of achieving stepwise refinement by suppressing unnecessary details and accentuating relevant details. **Encapsulation** now can be defined as the gathering into one unit of all aspects of the real-world entity modeled by that unit; this was termed *conceptual independence* in Section 1.9.

Data abstraction allows the designer to think at the level of the data structure and the operations performed on it and only later be concerned with the details of how the data structure and operations are implemented. Turning now to procedural abstraction, consider the result of defining a C function, `initialize_job_queue`. The effect is to extend the language by supplying the developer with another function, one that is not part of the language as originally defined. The developer can use `initialize_job_queue` in the same way as `sqrt` or `abs`.

The implications of procedural abstraction for design are as powerful as those of data abstraction. The designer can conceptualize the product in terms of high-level operations. These operations can be defined in terms of lower-level operations, until the lowest level is reached. At this level, the operations are expressed in terms of the predefined constructs of the programming language. At each level, the designer is concerned only with expressing the product in terms of operations appropriate to that level. The designer can ignore the level below, which is handled at the next level of abstraction, that is, the next refinement step. The designer also can ignore the level above, a level irrelevant from the viewpoint of designing the current level.

7.4.2 Data Encapsulation and Maintenance

Approaching data encapsulation from the viewpoint of maintenance, a basic issue is to identify the aspects of a product likely to change and design the product to minimize the effects of future changes. Data structures as such are unlikely to change; if a product includes job queues, for instance, then future versions are likely to incorporate them. At the same time, the specific way that job queues are implemented may well change, and data encapsulation provides a means of coping with that change.

Figure 7.17 depicts an implementation in C++ of the job queue data structure as **class `JobQueue`**; Figure 7.18 is the corresponding Java implementation. (Just in Case You

FIGURE 7.17

A C++ implementation of **class JobQueue**. (Problems caused by **public** attributes will be solved in Section 7.7.)

```
//
// Warning:
// This code has been written in such a way as to be accessible to readers
// who are not C++ experts, as opposed to using good C++ style. Also, vital
// features such as checks for overflow and underflow have been omitted for simplicity.
// See Just in Case You Wanted to Know Box 7.3 for details.
//
class JobQueue
{
    // attributes
public:
    int queueLength;    // length of job queue
    int queue[25];    // queue can contain up to 25 jobs

    // methods
public:
    void initializeJobQueue ()
    /*
     * an empty job queue has length 0
     */
    {
        queueLength = 0;
    }

    void addJobToQueue (int jobNumber)
    /*
     * add the job to the end of the job queue
     */
    {
        queue[queueLength] = jobNumber;
        queueLength = queueLength + 1;
    }

    int removeJobFromQueue ()
    /*
     * set jobNumber equal to the number of the job stored at the head of the queue,
     * remove the job at the head of the job queue, move up the remaining jobs,
     * and return jobNumber
     */
    {
        int jobNumber = queue[0];
        queueLength = queueLength - 1;
        for (int k = 0; k < queueLength; k++)
            queue[k] = queue[k + 1];
        return jobNumber;
    }
} // class JobQueue
```

FIGURE 7.18

A Java implementation of **class JobQueue**. (Problems caused by **public** attributes will be solved in Section 7.7.)

```
//
// Warning:
// This code has been written in such a way as to be accessible to readers
// who are not Java experts, as opposed to using good Java style.
// Also, vital features such as checks for overflow and underflow
// have been omitted for simplicity.
// See Just in Case You Wanted to Know Box 7.3 for details.
//
class JobQueue
{
    // attributes
    public int    queueLength;           // length of job queue
    public int    queue[ ] = new int[25]; // queue can contain up to 25 jobs

    // methods
    public void initializeJobQueue ()
    /*
     * an empty job queue has length 0
     */
    {
        queueLength = 0;
    }

    public void addJobToQueue (int jobNumber)
    /*
     * add the job to the end of the job queue
     */
    {
        queue[queueLength] = jobNumber;
        queueLength = queueLength + 1;
    }

    public int removeJobFromQueue ()
    /*
     * set jobNumber equal to the number of the job stored at the head of the queue,
     * remove the job at the head of the job queue, move up the remaining jobs,
     * and return jobNumber
     */
    {
        int jobNumber = queue[0];
        queueLength = queueLength - 1;
        for (int k = 0; k < queueLength; k++)
            queue[k] = queue[k + 1];
        return jobNumber;
    }
} // class JobQueue
```

Just in Case You Wanted to Know

Box 7.3

I deliberately wrote the code examples of Figure 7.17 and Figure 7.18 as well as the subsequent code examples in this chapter in such a way as to highlight data abstraction issues at the cost of good programming practice. For example, the number 25 in the definition of `JobQueue` in Figures 7.17 and 7.18 certainly should be coded as a parameter, that is, as a `const` in C++ or a `public static final` variable in Java. Also, for simplicity, I omitted checks for conditions such as underflow (trying to remove an item from an empty queue) or overflow (trying to add an item to a full queue). In any real product, it is absolutely essential to include such checks.

In addition, language-specific features have been minimized. For instance, a C++ programmer usually writes

```
queueLength++;
```

to increment the value of `queueLength` by 1, rather than

```
queueLength = queueLength + 1;
```

Similarly, use of constructors and destructors has been minimized.

In summary, I wrote the code in this chapter for pedagogic purposes only. It should not be utilized for any other purpose.

Wanted to Know Box 7.3 has comments on the programming style in Figures 7.17 and 7.18, as well as in the subsequent code examples in this chapter.) In Figure 7.17 or Figure 7.18, the queue is implemented as an array of up to 25 job numbers; the first element is `queue[0]` and the 25th is `queue[24]`. Each job number is represented as an integer. The reserved word **public** allows `queueLength` and `queue` to be visible everywhere in the operating system. The resulting common coupling is extremely poor practice and is corrected in Section 7.6.

Because they are **public**, the methods in **class JobQueue** may be invoked from anywhere in the operating system. In particular, Figure 7.19 shows how **class JobQueue** may be used by method `queueHandler` using C++, and Figure 7.20 is the corresponding

FIGURE 7.19
A C++
implementation
of
`queueHandler`.

```
class Scheduler
{
    ...
    public:
    void queueHandler ()
    {
        int          jobA, jobB;
        JobQueue    jobQueue;

        // various statements
        jobQueue.initializeJobQueue ();
        // more statements
        jobQueue.addJobToQueue (jobA);
        // still more statements
        jobB = jobQueue.removeJobFromQueue ();
        // further statements
    } // queueHandler
    ...
} // class Scheduler
```

FIGURE 7.20
A Java
implementation
of
queueHandler.

```

class Scheduler
{
    ...
    public void queueHandler ()
    {
        int          jobA, jobB;
        JobQueue    jobQueue = new JobQueue ();

        // various statements
        jobQueue.initializeJobQueue ();
        // more statements
        jobQueue.addJobToQueue (jobA);
        // still more statements
        jobB = jobQueue.removeJobFromQueue ();
        // further statements
    } // queueHandler
    ...
} // class Scheduler

```

FIGURE 7.21 A C++ implementation of a two-way linked **class JobRecord**. (Problems caused by **public** attributes will be solved in Section 7.6.)

```

class JobRecord
{
    public:
        int          jobNo;          // number of the job (integer)
        JobRecord  *inFront;       // pointer to the job record in front
        JobRecord  *inRear;        // pointer to the job record behind
} // class JobRecord

```

FIGURE 7.22 A Java implementation of a two-way linked **class JobRecord**. (Problems caused by **public** attributes will be solved in Section 7.6.)

```

class JobRecord
{
    public int          jobNo;          // number of the job (integer)
    public JobRecord  inFront;       // reference to the job record in front
    public JobRecord  inRear;        // reference to the job record behind
} // class JobRecord

```

Java implementation. Method `queueHandler` invokes methods `initializeJobQueue`, `addJobToQueue`, and `removeJobFromQueue` of **JobQueue** without having any knowledge as to how the job queue is implemented; the only information needed to use **class JobQueue** is interface information regarding the three methods.

Now suppose that the job queue currently is implemented as a linear list of job numbers, but a decision has been made to reimplement it as a two-way, linked list of job records. Each job record will have three components: the job number as before, a pointer to the job record in front of it in the linked list, and a pointer to the job record behind it. This is specified in C++ as shown in Figure 7.21 and in Java as shown in Figure 7.22. What changes must be made to the software product as a whole as a consequence of this modification to the way

FIGURE 7.23
Outline of a
C++
implementation
of **class**
JobQueue
using a two-way
linked list.

```

class JobQueue
{
    public:
    JobRecord *frontOfQueue; // pointer to the front of the queue
    JobRecord *rearOfQueue; // pointer to the rear of the queue

    void initializeJobQueue ()
    {
        /*
         * initialize the job queue by setting frontOfQueue and rearOfQueue to NULL
         */
    }

    void addJobToQueue (int JobNumber)
    {
        /*
         * Create a new job record,
         * place jobNumber in its jobNo field,
         * set its inFront field to point to the current rearOfQueue
         * (thereby linking the new record to the rear of the queue),
         * and set its inRear field to NULL.
         * Set the inRear field of the record pointed to by the current rearOfQueue
         * to point to the new record (thereby setting up a two-way link), and
         * finally, set rearOfQueue to point to this new record.
         */
    }

    int removeJobFromQueue ()
    {
        /*
         * set jobNumber equal to the jobNo field of the record at the front of the queue,
         * update frontOfQueue to point to the next item in the queue,
         * set the inFront field of the record that is now the head of the queue to NULL,
         * and return jobNumber.
         */
    }
} // class JobQueue

```

the job queue is implemented? In fact, only **JobQueue** itself has to be changed. Figure 7.23 shows the outline of a C++ implementation of **JobQueue** using the two-way linked list of Figure 7.21. Implementation details have been suppressed to highlight that the interface between **JobQueue** and the rest of the product (including method `queueHandler`) has not changed (but see Problem 7.12). That is, the three methods `initializeJobQueue`, `addJobToQueue`, and `removeJobFromQueue` are invoked in exactly the same way as before. Specifically, when method `addJobToQueue` is invoked, it still passes an integer value to **JobQueue**, and `removeJobFromQueue` still returns an integer value from **JobQueue**, even though the job queue itself has been implemented in an entirely different way. Consequently, the source code of method `queueHandler` (Figure 7.19) need not be changed at all. Thus, data encapsulation supports the implementation of data abstraction in a way that simplifies maintenance and reduces the chance of a regression fault.

Comparing Figures 7.17 and 7.18 and Figures 7.19 and 7.20, it is clear that, in these instances, the differences between the C++ and Java implementations essentially are syntactic. In the remainder of this chapter, we give only one implementation, together with a description of the syntactic differences in the other implementation. Specifically, the rest of the job queue code is in C++ and all the other code examples are in Java.

7.5 Abstract Data Types

Figure 7.17 (equivalently, Figure 7.18) is an implementation of a job queue **class**, that is, a data type together with the operations to be performed on instantiations of that data type. Such a construct is called an **abstract data type**.

Figure 7.24 shows how this abstract data type may be utilized in C++ for the three job queues of the operating system. Three job queues are instantiated: `highPriorityQueue`, `mediumPriorityQueue`, and `lowPriorityQueue`. (The Java version differs only in the syntax of the data declarations of the three job queues.) The statement `highPriorityQueue.initializeJobQueue ()` means “apply method `initializeJobQueue` to data structure `highPriorityQueue`,” and similarly for the other two statements.

Abstract data types are a widely applicable design tool. For example, suppose that a product is to be written in which a large number of operations have to be performed on rational numbers, that is, numbers that can be represented in the form n/d , where n and d are integers, $d \neq 0$. Rational numbers can be represented in a variety of ways, such as two elements of a one-dimensional array of integers or two attributes of a class. To implement rational numbers in terms of an abstract data type, a suitable representation for the data structure is chosen. In Java, it could be defined as shown in Figure 7.25, together with the various operations performed on rational numbers, such as constructing a rational number from two integers, adding two rational numbers, or multiplying two rational numbers.

FIGURE 7.24
C++ method `queueHandler` implemented using the abstract data type of Figure 7.17.

```
class Scheduler
{
    ...
    public:
        void queueHandler ()
        {
            int          job1, job2;
            JobQueue     highPriorityQueue;
            JobQueue     mediumPriorityQueue;
            JobQueue     lowPriorityQueue;

            // some statements
            highPriorityQueue.initializeJobQueue ();
            // some more statements
            mediumPriorityQueue.addJobToQueue (job1);
            // still more statements
            job2 = lowPriorityQueue.removeJobFromQueue ();
            // even more statements
        } // queueHandler
    ...
} // class Scheduler
```

FIGURE 7.25

Java abstract data type implementation of a rational number. (Problems caused by **public** attributes will be solved in Section 7.6.)

```
class Rational
{
    public int    numerator;
    public int    denominator;

    public void sameDenominator (Rational r, Rational s)
    {
        // code to reduce r and s to the same denominator
    }

    public boolean equal (Rational t, Rational u)
    {
        Rational    v, w;
        v = t;
        w = u;
        sameDenominator (v, w);
        return (v.numerator == w.numerator);
    }

    // methods to add, subtract, multiply, and divide two rational numbers
}

// class Rational
```

(The problems induced by **public** attributes such as `numerator` and `denominator` in Figure 7.25 will be fixed in Section 7.6.) The corresponding C++ implementation differs in the placement of the reserved word **public**. Also, an ampersand is needed when an argument is passed by reference.

Abstract data types support both data abstraction and procedural abstraction (Section 7.4.1). In addition, when a product is modified, it is unlikely that the abstract data types will be changed; at worst, additional operations may have to be added to an abstract data type. Therefore, from both the development and the maintenance viewpoints, abstract data types are an attractive tool for software producers.

7.6 Information Hiding

The two types of abstraction discussed in Section 7.4.1 (data abstraction and procedural abstraction) are in turn instances of a more general design concept put forward by Parnas, **information hiding** [Parnas, 1971, 1972a, 1972b]. Parnas's ideas are directed toward future maintenance. Before a product is designed, a list should be made of implementation decisions likely to change in the future. Modules then should be designed so that the implementation details of the resulting design are hidden from other modules. Thus, each future change is localized to one specific module. Because the details of the original implementation decision are not visible to other modules, changing the design clearly cannot affect any other module. (See Just in Case You Wanted to Know Box 7.4 for a further insight into information hiding.)

To see how these ideas can be used in practice, consider Figure 7.24, which uses the abstract data type implementation of Figure 7.17. A primary reason for using an abstract data type is to ensure that the contents of a job queue can be changed only by invoking one

Just in Case You Wanted to Know

Box 7.4

The term *information hiding* is somewhat of a misnomer. A more accurate description would be “details hiding,” because what is hidden is not information but implementation details.

of the three methods of Figure 7.17. Unfortunately, the nature of that implementation is such that job queues can be changed in other ways as well. Attributes `queueLength` and `queue` are both declared **public** in Figure 7.17 and therefore accessible inside `queueHandler`. As a result, in Figure 7.24, it is perfectly legal C++ (or Java) to use an assignment statement such as

```
highPriorityQueue.queue[7] = -5678;
```

anywhere in `queueHandler` to change `highPriorityQueue`. In other words, the contents of a job queue can be changed without using any of the three operations of the abstract data type. In addition to the implications this might have with regard to lowering cohesion and increasing coupling, management must recognize that the product may be vulnerable to computer crime as described in Section 7.3.2.

Fortunately, there is a way out. The designers of both C++ and Java provided for information hiding within a class specification. This is shown in Figure 7.26 for C++ (the Java syntactic differences are as before). Other than changing the visibility modifier for the attributes from **public** to **private**, Figure 7.26 is identical to Figure 7.17. Now the only information visible to other modules is that **JobQueue** is a **class** and that three operations with specified interfaces can operate on the resulting job queues. But the exact way job

FIGURE 7.26

A C++ abstract data type implementation with information hiding, correcting the problem of Figures 7.17, 7.18, 7.21, 7.22, and 7.25.

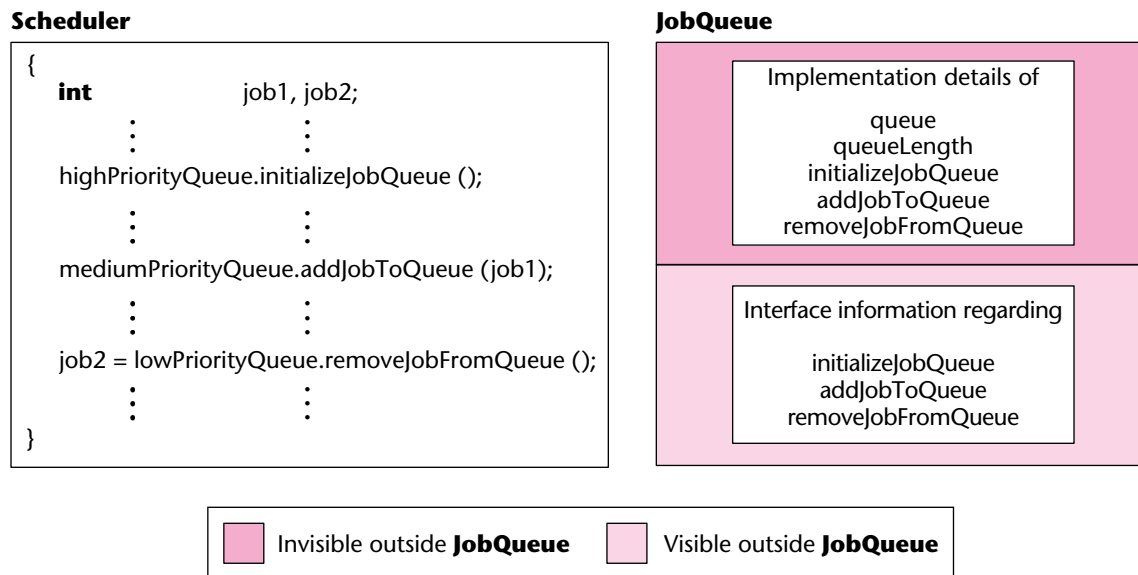
```
class JobQueue
{
    // attributes
    private:
    int   queueLength;    // length of job queue
    int   queue[25];     // queue can contain up to 25 jobs

    // methods
    public:
    void initializeJobQueue ()
    {
        // body of method unchanged from Figure 7.17
    }

    void addJobToQueue (int jobNumber)
    {
        // body of method unchanged from Figure 7.17
    }

    int removeJobFromQueue ()
    {
        // body of method unchanged from Figure 7.17
    }
} // class JobQueue
```

FIGURE 7.27 Representation of an abstract data type with information hiding achieved via **private** attributes (Figure 7.26 with Figure 7.24).



queues are implemented is **private**, that is, invisible to the outside. The diagram in Figure 7.27 shows how a class with **private** attributes enables a C++ or Java user to implement an abstract data type with full information hiding.

Information hiding techniques also can be used to obviate common coupling, as mentioned at the end of Section 7.3.2. Consider again the product described in that section, a computer-aided design tool for petroleum storage tanks specified by 55 descriptors. If the product is implemented with **private** operations for initializing a descriptor and **public** operations for obtaining the value of a descriptor, then there is no common coupling. This type of solution is characteristic of the object-oriented paradigm, because as described in the next section, objects support information hiding. This is another strength of object technology.

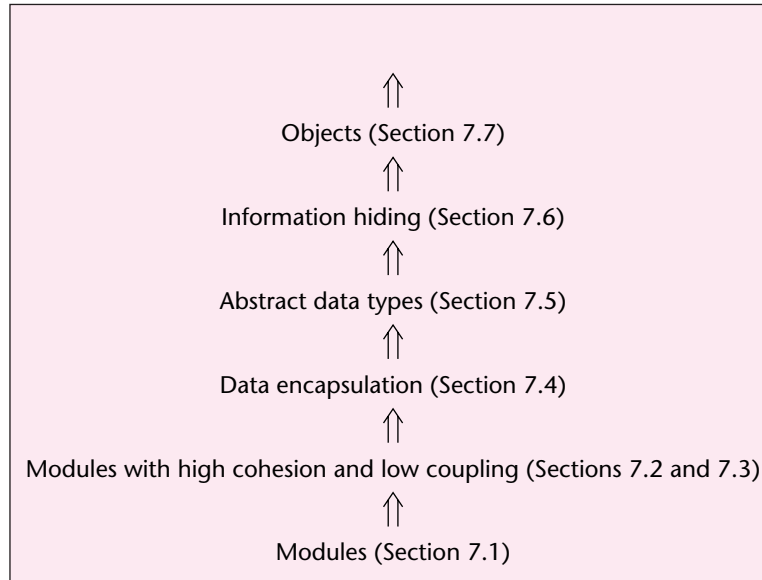
7.7 Objects

As stated at the beginning of this chapter, objects simply are the next step in the progression shown in Figure 7.28. Nothing is special about objects; they are as ordinary as abstract data types or modules with informational cohesion. The importance of objects is that they have all the properties possessed by their predecessors in Figure 7.28, as well as additional properties of their own.

An incomplete definition of an object is that an object is an instantiation (instance) of an abstract data type. That is, a product is designed in terms of abstract data types, and the variables (objects) of the product are instantiations of the abstract data types. But defining an object as an instantiation of an abstract data type is too simplistic. Something more is needed; namely, **inheritance**, a concept first introduced in Simula 67 [Dahl and Nygaard,

FIGURE 7.28

The major concepts of Chapter 7 and the section in which each is described.



1966]. Inheritance is supported by all object-oriented programming languages, such as Smalltalk [Goldberg and Robson 1989], C++ [Stroustrup, 2000], and Java [Flanagan, 2002]. The basic idea behind inheritance is that new data types can be defined as extensions of previously defined types, rather than having to be defined from scratch [Meyer, 1986].

In an object-oriented language, a **class** can be defined as an abstract data type that supports inheritance. An **object** then is an instantiation of a class. To see how classes are used, consider the following example. Define **HumanBeing** to be a class and Joe to be an object, an instance of that class. Every **HumanBeing** has certain attributes such as age and height, and values can be assigned to those attributes when describing the object Joe. Now suppose that **Parent** is defined to be a **subclass** (or derived class) of **HumanBeing**. This means that an instance of subclass **Parent** has all the attributes of a **HumanBeing** and, in addition, may have attributes of his or her own such as name of oldest child and number of children. This is depicted in Figure 7.29. In object-oriented terminology, a **Parent** *isA* **HumanBeing**. That is why the arrow in Figure 7.29 seems to be going in the wrong direction. In fact, the arrow depicts the *isA* relation and therefore points from the derived class to the base class. (The use of the open arrowhead to denote inheritance is a UML convention; another is that class names appear in boldface with the first letter of each word capitalized. UML is discussed in more detail in Part 2, especially in Chapter 16.)

Class **Parent** *inherits* all the attributes of **HumanBeing**, because class **Parent** is a derived class (or subclass) of base class **HumanBeing**. If Fred is an object and an instance of class **Parent**, then Fred has all the attributes of a **Parent** and also inherits all the attributes of a **HumanBeing**. A Java implementation is shown in Figure 7.30. The C++ version differs in the placement of the **private** and **public** modifiers. Also, the Java syntax **extends** is replaced in C++ by **: public** in this example.

The property of inheritance is an essential feature of all object-oriented programming languages. However, neither inheritance nor the concept of a class is supported by classical

FIGURE 7.29
Derived types
and inheritance.

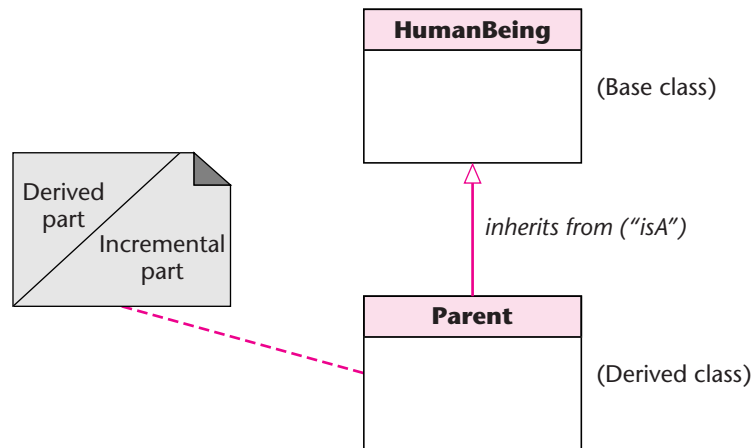


FIGURE 7.30
Java
implementation
of Figure 7.29.

```

class HumanBeing
{
    private int    age;
    private float height;

    // public declarations of operations on HumanBeing
} // class HumanBeing

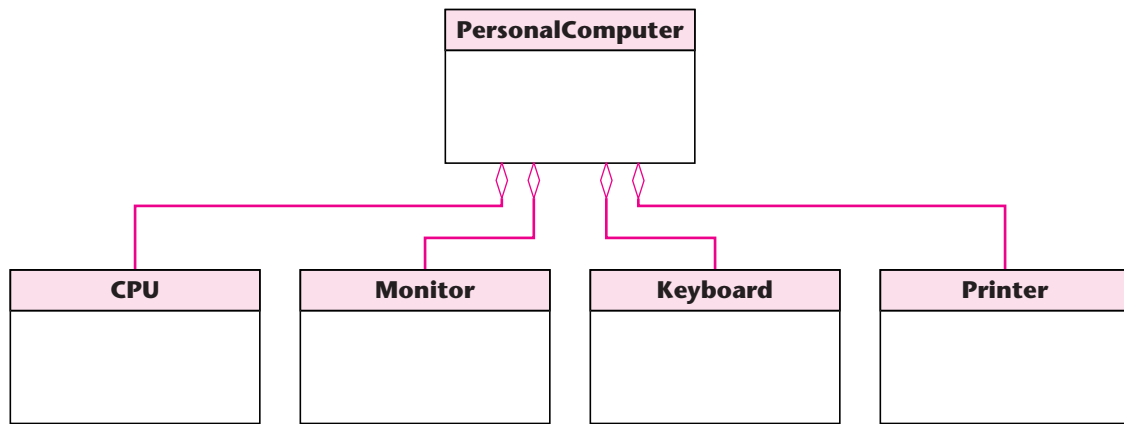
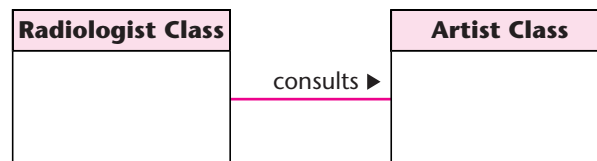
class Parent extends HumanBeing
{
    private String nameOfOldestChild;
    private int    numberOfChildren;

    // public declarations of operations on Parent
} // class Parent
  
```

languages such as C, COBOL, or FORTRAN. Therefore, the object-oriented paradigm cannot be directly implemented in these languages (but see Section 8.7.4).

In the terminology of the object-oriented paradigm, there are two other ways of looking at the relationship between **Parent** and **HumanBeing** in Figure 7.29. We can say that **Parent** is a **specialization** of **HumanBeing** or that **HumanBeing** is a **generalization** of **Parent**. In addition to specialization and generalization, classes have two other basic relationships [Blaha, Premerlani, and Rumbaugh, 1988]: aggregation and association. **Aggregation** refers to the components of a class. For example, class **PersonalComputer** might consist of components **CPU**, **Monitor**, **Keyboard**, and **Printer**. This is depicted in Figure 7.31 (the use of a diamond to denote aggregation is another UML convention). Nothing is new about this; it occurs whenever a language supports records, such as a **struct** in C. Within the object-oriented context, however, it is used to group related items, resulting in a reusable class (Section 8.1).

FIGURE 7.31 An aggregation example.

FIGURE 7.32 An association example.
An association example.

Association refers to a relationship of some kind between two apparently unrelated classes. For example, there seems to be no connection between a radiologist and an artist, but a radiologist may consult an artist in regard to drawing the diagrams for a book describing how an MRI machine works. Association is depicted in Figure 7.32. The nature of the association in this instance is indicated by the word *consults*. In addition, the solid triangle (termed a **navigation triangle** in UML) indicates the direction of the association; after all, an artist with a broken ankle might consult a radiologist.

In passing, one aspect of Java and C++ notation, like that of other object-oriented languages, explicitly reflects the equivalence of operation and data. First, consider a classical language that supports records; C, for example. Suppose that `record_1` is a **struct** (record) and `field_2` is a field within the class. Then, the field is referred to as `record_1.field_2`. That is, the period `.` denotes membership within the record. If `function_3` is a function within a C module, then `function_3 ()` denotes an invocation of that function.

In contrast, suppose that **ClassA** is a **class**, with attribute `attributeB` and method `methodC`. Suppose further that `ourObject` is an instance of **ClassA**. Then the field is referred to as `ourObject.attributeB`. Furthermore, `ourObject.methodC ()` denotes an invocation of the method. Here, the period is used to denote membership within an object, whether the member is an attribute or a method.

The advantages of using objects (or, rather, classes) are precisely those of using abstract data types, including data abstraction and procedural abstraction. In addition, the inheritance aspects of classes provide a further layer of data abstraction, leading to easier and less fault-prone product development. Yet another strength follows from combining inheritance with polymorphism and dynamic binding, the subject of the next section.

7.8 Inheritance, Polymorphism, and Dynamic Binding

Suppose that the operating system of a computer is called on to open a file. That file could be stored on a number of different media. For example, it could be a disk file, a tape file, or a diskette file. Using the classical paradigm, there would be three differently named functions, `open_disk_file`, `open_tape_file`, and `open_diskette_file`; this is shown in Figure 7.33(a). If `my_file` is declared to be a file, then at run time, it is necessary to test whether it is a disk file, a tape file, or a diskette file to determine which function to invoke. The corresponding classical code is shown in Figure 7.34(a).

In contrast, when the object-oriented paradigm is used, a class named **FileClass** is defined, with three derived classes **DiskFileClass**, **TapeFileClass**, and **DisketteFileClass**. This is shown in Figure 7.33(b); recall that the open arrowhead denotes inheritance.

Now, suppose that method `open` were defined in parent class **FileClass** and inherited by the three derived classes. Unfortunately, this would not work, because different operations need to be carried out to open the three different types of files.

The solution is as follows. In parent class **FileClass**, a dummy method `open` is declared. In Java, such a method is declared to be **abstract**; in C++, the reserved word **virtual** is used instead. A specific implementation of the method appears in each of the three derived classes and each method is given an identical name, that is, `open`, as shown in Figure 7.33(b). Again, suppose that `myFile` is declared to be a file. At run time, the message

`myFile.open ()`

FIGURE 7.33 Operations needed to open a file. (a) Classical implementation. (b) Object-oriented file class hierarchy using Java notation.

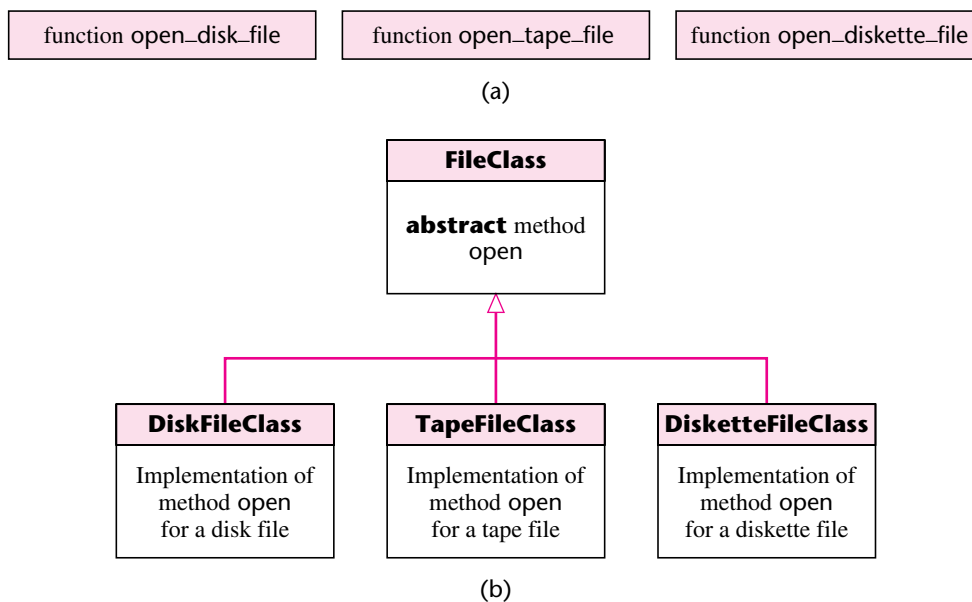


FIGURE 7.34

(a) Classical code to open a file, corresponding to Figure 7.33(a).
 (b) Object-oriented code to open a file, corresponding to Figure 7.33(b).

```

switch (file_type)
{
    case 1:
        open_disk_file ();           // file_type 1 corresponds to a disk file
        break;
    case 2:
        open_tape_file ();           // file_type 2 corresponds to a tape file
        break;
    case 3:
        open_diskette_file ();       // file_type 3 corresponds to a diskette file
        break;
}
  
```

(a)

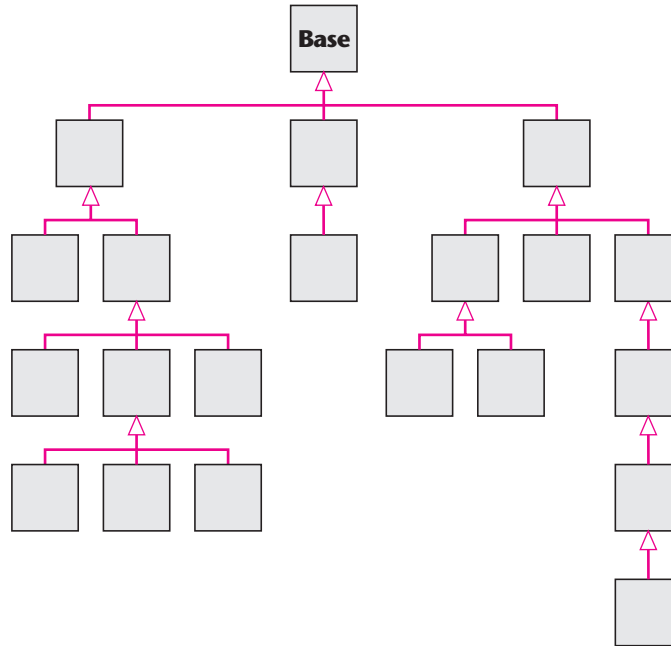
```
myFile.open ( )
```

(b)

is sent. The object-oriented system now determines whether `myFile` is a disk file, a tape file, or a diskette file and invokes the appropriate version of `open`. That is, the system determines at run time whether object `myFile` is an instance of class **DiskFileClass**, class **TapeFileClass**, or class **DisketteFileClass** and automatically invokes the correct method. Because this has to be done at run time (dynamically) and not at compile time (statically), the act of connecting an object to the appropriate method is termed **dynamic binding**. Furthermore, because the method `open` can be applied to objects of different classes, it is termed **polymorphic**, which means “of many shapes.” Just as carbon crystals come in many different shapes, including hard diamonds and soft graphite, so the method `open` comes in three different versions. In Java, these versions are denoted `DiskFileClass.open`, `TapeFileClass.open`, and `DisketteFileClass.open`. (In C++, the period is replaced by two colons, and the files are denoted `DiskFileClass::open`, `TapeFileClass::open`, and `DisketteFileClass::open`.) However, because of dynamic binding, it is not necessary to determine which method to invoke to open a specific file. Instead, at run time, it is necessary to send only the message `myFile.open ()` and the system will determine the type (class) of `myFile` and invoke the correct method; this is shown in Figure 7.34(b).

These ideas are applicable to more than just **abstract (virtual)** methods. Consider a hierarchy of classes, as shown in Figure 7.35. All classes are derived by inheritance from the **Base** class. Suppose method `checkOrder (b : Base)` takes as argument an instance of class **Base**. Then, as a consequence of inheritance, polymorphism, and dynamic binding, it is valid to invoke `checkOrder` with an argument not just of class **Base** but also of any subclass of class **Base**, that is, any class derived from **Base**. All that is needed is to invoke `checkOrder` and everything is taken care of at run time. This technique is extremely powerful, in that the software professional need not be concerned about the precise type of an argument at the time that a message is sent.

FIGURE 7.35
A hierarchy of classes.



However, polymorphism and dynamic binding also have major weaknesses.

1. It generally is not possible to determine at compilation time which version of a specific polymorphic method will be invoked at run time. Accordingly, the cause of a failure can be extremely difficult to determine.
2. Polymorphism and dynamic binding can have a negative impact on maintenance. The first task of a maintenance programmer usually is to try to understand the product (as explained in Chapter 15, the maintainer rarely is the person who developed that code). However, this can be laborious if there are multiple possibilities for a specific method. The programmer has to consider all the possible methods that could be invoked dynamically at a specific place in the code, a time-consuming task.

Thus, polymorphism and dynamic binding add both strengths and weaknesses to the object-oriented paradigm.

We conclude this chapter with a discussion of the object-oriented paradigm.

7.9 The Object-Oriented Paradigm

There are two ways of looking at every software product. One way is to consider just the data, including local and global variables, arguments, dynamic data structures, files, and so on. Another way of viewing a product is to consider just the operations performed on the data, that is, the procedures and the functions. In terms of this division of software into data and operations, the classical techniques essentially fall into two groups. Operation-oriented

techniques primarily consider the operations of the product. The data are of secondary importance, considered only after the operations of the product have been analyzed in depth. Conversely, data-oriented techniques stress the data of the product; the operations are examined only within the framework of the data.

A fundamental weakness of both the data- and operation-oriented approaches is that data and operation are two sides of the same coin; a data item cannot change unless an operation is performed on it, and operations without associated data are equally meaningless. Therefore, techniques that give equal weight to data and operations are needed. It should not come as a surprise that the object-oriented techniques do this. After all, an object comprises both data and operations. Recall that an object is an instance of an abstract data type (more precisely, of a class). It therefore incorporates both data and the operations performed on those data, and the data and the operations are present in objects as equal partners. Similarly, in all the object-oriented techniques, data and operations are considered of the same importance; neither takes precedence over the other.

It is inaccurate to claim that data and operations are considered simultaneously in the techniques of the object-oriented paradigm. From the material on stepwise refinement (Section 5.1), it is clear that sometimes data have to be stressed and other times operations are more critical. Overall, however, data and operations are given equal importance during the workflows of the object-oriented paradigm.

Many reasons are given in Chapter 1 and this chapter as to why the object-oriented paradigm is superior to the classical paradigm. Underlying all these reasons is that a well-designed object, that is, an object with high cohesion and low coupling, models all the aspects of one physical entity. That is, there is a clear mapping between a real-world entity and the object that models it.

The details of how this is implemented are hidden; the only communication with an object is via messages sent to that object. As a result, objects essentially are independent units with a well-defined interface. Consequently, they can be maintained easily and safely; the chance of a regression fault is reduced. Furthermore, as will be explained in Chapter 8, objects are reusable, and this reusability is enhanced by the property of inheritance. Turning now to development using objects, it is safer to construct a large-scale product by combining these fundamental building blocks of software than to use the classical paradigm. Because objects essentially are independent components of a product, development of the product, as well as management of that development, is easier and hence less likely to induce faults.

All these aspects of the superiority of the object-oriented paradigm raise a question: If the classical paradigm is so inferior to the object-oriented paradigm, why has the classical paradigm had so much success? This can be explained by realizing that the classical paradigm was adopted at a time when software engineering was not widely practiced. Instead, software was simply “written.” For managers, the most important thing was for programmers to churn out lines of code. Little more than lip service was paid to the requirements and analysis (systems analysis) of a product, and design was almost never performed. The code-and-fix model (Section 3.1) was typical of the techniques of the 1970s. Therefore, use of the classical paradigm exposed the majority of software developers to methodical techniques for the first time. Small wonder, then, that the so-called structured techniques of the classical paradigm led to major improvements in the software industry worldwide. However, as software products grew in size, inadequacies of the structured techniques

started to become apparent, and the object-oriented paradigm was proposed as a better alternative.

This, in turn, leads to another question: How do we know for certain that the object-oriented paradigm is superior to all other present-day techniques? No data are available that prove beyond all doubt that object-oriented technology is better than anything else currently available, and it is hard to imagine how such data could be obtained. The best we can do is to rely on the experiences of organizations that have adopted the object-oriented paradigm. Although not all reports are favorable, the majority (if not the overwhelming majority) attest that using the object-oriented paradigm is a wise decision.

For example, IBM has reported on three totally different projects that were developed using object-oriented technology [Capper, Colgate, Hunter, and James, 1994]. In almost every respect, the object-oriented paradigm greatly outperformed the classical paradigm. Specifically, there were major decreases in the number of faults detected, far fewer change requests during both development and postdelivery maintenance that were not the result of unforeseeable business changes, and significant increases in both adaptive and perfective maintainability. Also improvement in usability was found, although not as large as the previous four improvements, and no meaningful difference in performance.

A survey of 150 experienced U.S. software developers was undertaken to determine their attitudes toward the object-oriented paradigm [Johnson, 2000]. The sample consisted of 96 developers who used the object-oriented paradigm and 54 who still use the classical paradigm to develop software. Both groups felt that the object-oriented paradigm is superior, although the positive attitude of the object-oriented group was significantly stronger. Both groups essentially discounted the various weaknesses of the object-oriented paradigm.

Notwithstanding the many strengths of the object-oriented paradigm, some difficulties and problems indeed have been reported. A frequently reported problem concerns development effort and size. The first time anything new is done, it takes longer than on subsequent occasions; this initial period is sometimes referred to as the **learning curve**. But when the object-oriented paradigm is used for the first time by an organization, it often takes longer than anticipated, even allowing for the learning curve, because the size of the product is larger than when structured techniques are used. This is particularly noticeable when the product has a graphical user interface (GUI) (see Section 10.13). Thereafter, things improve greatly. First, postdelivery maintenance costs are lower, reducing the overall lifetime cost of the product. Second, the next time that a new product is developed, some of the classes from the previous project can be reused, further reducing software costs. This has been especially significant when a GUI has been used for the first time; much of the effort that went into the GUI can be recouped in subsequent products.

Problems of inheritance are harder to solve. A major reason for using inheritance is to create a new subclass that differs slightly from its parent class without affecting the parent class or any other ancestor class in the inheritance hierarchy. Conversely, however, once a product has been implemented, any change to an existing class directly affects all its descendants in the inheritance hierarchy; this often is referred to as the **fragile base class problem**. At the very least, the affected units have to be recompiled. In some cases, the methods of the relevant objects (instantiations of the affected subclasses) have to be recoded; this can be a nontrivial task. To minimize this problem, it is important that all classes be meticulously designed during the development process. This will reduce the ripple effect induced by a change to an existing class.

A second problem can result from a cavalier use of inheritance. Unless explicitly prevented, a subclass inherits all the attributes of its parent class(es). Usually, subclasses have additional attributes of their own. As a consequence, objects lower in the inheritance hierarchy quickly can get large, with resulting storage problems [Bruegge, Blythe, Jackson, and Shufelt, 1992]. One way to prevent this is to change the dictum “use inheritance wherever possible” to “use inheritance wherever appropriate.” In addition, if a descendent class does not need an attribute of an ancestor, then that attribute should be explicitly excluded.

A third group of problems stem from polymorphism and dynamic binding. These were described in Section 7.8.

Fourth, it is possible to write bad code in any language. However, it is easier to write bad code in an object-oriented language than in a classical language because object-oriented languages support a variety of constructs that, when misused, add unnecessary complexity to a software product. Therefore, when using the object-oriented paradigm, extra care needs to be taken to ensure that the code is always of the highest quality.

One final question is this: Someday might there be something better than the object-oriented paradigm? That is, in the future will a new technology appear in the space above the topmost arrow in Figure 7.28? Even its strongest proponents do not claim that the object-oriented paradigm is the ultimate answer to all software engineering problems. Furthermore, today’s software engineers are looking beyond objects to the next major breakthrough. After all, in few fields of human endeavor are the discoveries of the past superior to anything that is being put forward today. The object-oriented paradigm is sure to be superseded by the methodologies of the future. It has been suggested that **aspect-oriented programming** (AOP) may play a role [Murphy et al., 2001]. It remains to be seen whether AOP will indeed be the next major concept in future versions of Figure 7.28 or whether some other technology will be widely adopted as the successor to the object-oriented paradigm. The important lesson is that, based on today’s knowledge, the object-oriented paradigm appears to be better than the alternatives.

Chapter Review

The chapter begins with a description of a module (Section 7.1). The next two sections analyze what constitutes a well-designed module in terms of module cohesion and module coupling (Sections 7.2 and 7.3). Specifically, a module should have high cohesion and low coupling. A description is given of the different types of cohesion and coupling. Various types of abstraction are presented in Sections 7.4 through 7.7. In data encapsulation (Section 7.4), a module comprises a data *structure* and the actions performed on that data structure. An abstract data type (Section 7.5) is a data *type*, together with the actions performed on instances of that type. Information hiding (Section 7.6) consists of designing a module in such a way that implementation details are hidden from other modules. The progression of increasing abstraction culminates in the description of a class, an abstract data type that supports inheritance (Section 7.7). An object is an instance of a class. Inheritance, polymorphism, and dynamic binding are the subjects of Section 7.8. The chapter concludes with a discussion of the object-oriented paradigm (Section 7.9).

For Further Reading

Objects were first described in Dahl and Nygaard [1966]. Many of the ideas in this chapter originally were put forward by Parnas [1971, 1972a, 1972b]. The use of abstract data types in software development was put forward in Liskov and Zilles [1974]; another important early paper is [Guttag, 1977].

The primary source on cohesion and coupling is [Stevens, Myers, and Constantine, 1974]. The ideas of composite/structured design have been extended to objects [Binkley and Schach, 1997].

Introductory material on objects can be found in Meyer [1997]. Different types of inheritance are described in Meyer [1996b]. A number of short articles on the object-oriented paradigm can be found in El-Rewini et al. [1995]. The proceedings of the annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) include a wide selection of research papers as well as reports describing successful object-oriented projects. The successful use of the object-oriented paradigm in three IBM projects is described in Capper, Colgate, Hunter, and James [1994]. A survey of attitudes toward the object-oriented paradigm appears in Johnson [2000]. Fayad, Tsai, and Fulghum [1996] describe how to make the transition to object-oriented technology; a number of recommendations for managers are included.

The October 1992 issue of *IEEE Computer* contains a number of important articles on objects, including [Meyer, 1992], which describes “design by contract.” A variety of articles on objects can be found in the January 1993 issue of *IEEE Software*; Snyder’s [1993] paper precisely defining key terms in the field is particularly useful. Possible drawbacks of polymorphism are described in Ponder and Bush [1994]. The October 1995 issue of the *Communications of the ACM* contains articles on object technology, as does issue no. 2, 1996, of the *IBM Systems Journal*.

Eleven articles on aspect-oriented programming appear in the October 2001 issue of the *Communications of the ACM*; [Elrad et al., 2001] and [Murphy et al., 2001] are of particular interest. An investigation of the impact of inheritance on fault densities appears in Cartwright and Shepperd [2000].

Key Terms

abstract data type, 191	context, 169	learning curve, 202
abstraction, 184	control coupling, 178	logic, 169
aggregation, 196	coupling, 169	logical cohesion, 171
aspect-oriented programming (AOP), 203	data abstraction, 185	module, 167
association, 197	data coupling, 180	navigation triangle, 197
binding, 169	data encapsulation, 184	object, 195
clandestine common coupling, 177	dynamic binding, 199	operation, 169
class, 175	encapsulation, 185	polymorphism, 199
cohesion, 169	flowchart cohesion, 173	procedural abstraction, 185
coincidental cohesion, 170	fragile base class problem, 202	procedural cohesion, 172
common coupling, 176	functional cohesion, 173	specialization, 196
communicational cohesion, 173	generalization, 196	stamp coupling, 179
content coupling, 176	information hiding, 192	strength, 169
	informational cohesion, 174	subclass, 195
	inheritance, 194	temporal cohesion, 172
	isA, 195	

Problems

- 7.1 Choose any programming language with which you are familiar. Consider the two definitions of modularity given in Section 7.1. Determine which of the two definitions includes what you intuitively understand to constitute a module in the language you have chosen.
- 7.2 Determine the cohesion of the following modules:
 - edit_profit_and_tax_record
 - edit_profit_record_and_tax_record

read_delivery_record_and_check_salary_payments
compute_the_optimal_cost_using_Aksen's_algorithm
measure_vapor_pressure_and_sound_alarm_if_necessary

- 7.3 You are a software engineer involved in product development. Your manager asks you to investigate ways of ensuring that modules designed by your group will be as reusable as possible. What do you tell her?
- 7.4 Your manager now asks you to determine how existing modules can be reused. Your first suggestion is to break each module with coincidental cohesion into separate modules with functional cohesion. Your manager correctly points out that the separate modules have not been tested nor have they been documented. What do you say now?
- 7.5 What is the influence of cohesion on maintenance?
- 7.6 What is the influence of coupling on maintenance?
- 7.7 Distinguish between data encapsulation and abstract data types.
- 7.8 Distinguish between abstraction and information hiding.
- 7.9 Distinguish between polymorphism and dynamic binding.
- 7.10 What happens if we use polymorphism without dynamic binding?
- 7.11 What happens if we use dynamic binding without polymorphism?
- 7.12 Convert the comments in Figure 7.23 to C++ or Java, as specified by your instructor. Make sure that the resulting module executes correctly.
- 7.13 It has been suggested that C++ and Java support implementation of abstract data types but only at the cost of giving up information hiding. Discuss this claim.
- 7.14 As pointed out in Just in Case You Wanted to Know Box 7.1, objects first were put forward in 1966. Only after essentially being reinvented nearly 20 years later did objects begin to receive widespread acceptance. Can you explain this phenomenon?
- 7.15 Your instructor will distribute a classical software product. Analyze the modules from the viewpoints of information hiding, levels of abstraction, coupling, and cohesion.
- 7.16 Your instructor will distribute an object-oriented software product. Analyze the modules from the viewpoints of information hiding, levels of abstraction, coupling, and cohesion. Compare your answer with that of Problem 7.15.
- 7.17 (Term Project) Suppose that the Ophelia's Oasis product of Appendix A were developed using the classical paradigm. Give examples of modules of functional cohesion that you would expect to find. Now suppose that the product was developed using the object-oriented paradigm. Give examples of classes that you would expect to find.
- 7.18 (Readings in Software Engineering) Your instructor will distribute copies of [Johnson, 2000]. Why do you think that the respondents viewed the drawbacks to the object-oriented paradigm as essentially irrelevant?

References

- [Binkley and Schach, 1997] A. B. BINKLEY AND S. R. SCHACH, "Toward a Unified Approach to Object-Oriented Coupling," *Proceedings of the 35th Annual ACM Southeast Conference*, Murfreesboro, TN, April 2–4, 1997, pp. 91–97.
- [Blaaha, Premerlani, and Rumbaugh, 1988] M. R. BLAHA, W. J. PREMERLANI, AND J. E. RUMBAUGH, "Relational Database Design Using an Object-Oriented Methodology," *Communications of the ACM* **31** (April 1988), pp. 414–27.
- [Bruegge, Blythe, Jackson, and Shufelt, 1992] B. BRUEGGE, J. BLYTHE, J. JACKSON, AND J. SHUFELT, "Object-Oriented Modeling with OMT," *Proceedings of the Conference on Object-Oriented*

- Programming, Languages, and Systems, OOPSLA '92, *ACM SIGPLAN Notices* **27** (October 1992), pp. 359–76.
- [Capper, Colgate, Hunter, and James, 1994] N. P. CAPPER, R. J. COLGATE, J. C. HUNTER, AND M. F. JAMES, “The Impact of Object-Oriented Technology on Software Quality: Three Case Histories,” *IBM Systems Journal* **33** (No. 1, 1994), pp. 131–57.
- [Cartwright and Shepperd, 2000] M. CARTWRIGHT AND M. SHEPPERD, “An Empirical Investigation of an Object-Oriented Software System,” *IEEE Transactions on Software Engineering* **26** (August 2000), pp. 786–95.
- [Dahl and Nygaard, 1966] O.-J. DAHL AND K. NYGAARD, “SIMULA—An ALGOL-Based Simulation Language,” *Communications of the ACM* **9** (September 1966), pp. 671–78.
- [El-Rewini et al., 1995] H. EL-REWINI, S. HAMILTON, Y.-P. SHAN, R. EARLE, S. MCGAUGHEY, A. HELAL, R. BADRACHALAM, A. CHIEN, A. GRIMSHAW, B. LEE, A. WADE, D. MORSE, A. ELMAGRAMID, E. PITOURA, R. BINDER, AND P. WEGNER, “Object Technology,” *IEEE Computer* **28** (October 1995), pp. 58–72.
- [Elrad et al., 2001] T. ELRAD, M. AKSIT, G. KICZALES, K. LIEBERHERR, AND H. OSSHER, “Discussing Aspects of AOP,” *Communications of the ACM* **44** (October 2001), pp. 33–38.
- [Fayad, Tsai, and Fulghum, 1996] M. E. FAYAD, W.-T. TSAI, AND M. L. FULGHUM, “Transition to Object-Oriented Software Development,” *Communications of the ACM* **39** (February 1996), pp. 108–21.
- [Flanagan, 2002] D. FLANAGAN, *Java in a Nutshell: A Desktop Quick Reference*, 4th ed., O’Reilly and Associates, Sebastopol, CA, 2002.
- [Gerald and Wheatley, 1999] C. F. GERALD AND P. O. WHEATLEY, *Applied Numerical Analysis*, 6th ed., Addison Wesley, Reading, MA, 1999.
- [Goldberg and Robson, 1989] A. GOLDBERG AND D. ROBSON, *Smalltalk-80: The Language*, Addison Wesley, Reading, MA, 1989.
- [Gutttag, 1977] J. GUTTAG, “Abstract Data Types and the Development of Data Structures,” *Communications of the ACM* **20** (June 1977), pp. 396–404.
- [Johnson, 2000] R. A. JOHNSON, “The Ups and Downs of Object-Oriented System Development,” *Communications of the ACM* **43** (October 2000), pp. 69–73.
- [Knuth, 1974] D. E. KNUTH, “Structured Programming with `go to` Statements,” *ACM Computing Surveys* **6** (December 1974), pp. 261–301.
- [Liskov and Zilles, 1974] B. LISKOV AND S. ZILLES, “Programming with Abstract Data Types,” *ACM SIGPLAN Notices* **9** (April 1974), pp. 50–59.
- [Meyer, 1986] B. MEYER, “Genericity versus Inheritance,” Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications, *ACM SIGPLAN Notices* **21** (November 1986), pp. 391–405.
- [Meyer, 1992] B. MEYER, “Applying ‘Design by Contract’,” *IEEE Computer* **25** (October 1992), pp. 40–51.
- [Meyer, 1996b] B. MEYER, “The Many Faces of Inheritance: A Taxonomy of Taxonomy,” *IEEE Computer* **29** (May 1996), pp. 105–8.
- [Meyer, 1997] B. MEYER, *Object-Oriented Software Construction*, 2nd ed., Prentice-Hall, Upper Saddle River, NJ, 1997.
- [Murphy et al., 2001] G. C. MURPHY, R. J. WALKER, E. L. A. BANNIASSAD, M. P. ROBILLARD, A. LIA, AND M. A. KERSTEN, “Does Aspect-Oriented Programming Work?” *Communications of the ACM* **44** (October 2001), pp. 75–78.
- [Myers, 1978b] G. J. MYERS, *Composite/Structured Design*, Van Nostrand Reinhold, New York, 1978.

- [Parnas, 1971] D. L. PARNAS, “Information Distribution Aspects of Design Methodology,” *Proceedings of the IFIP Congress*, Ljubljana, Yugoslavia, 1971, pp. 339–44.
- [Parnas, 1972a] D. L. PARNAS, “A Technique for Software Module Specification with Examples,” *Communications of the ACM* **15** (May 1972), pp. 330–36.
- [Parnas, 1972b] D. L. PARNAS, “On the Criteria to Be Used in Decomposing Systems into Modules,” *Communications of the ACM* **15** (December 1972), pp. 1053–58.
- [Ponder and Bush, 1994] C. PONDER AND B. BUSH, “Polymorphism Considered Harmful,” *ACM SIGSOFT Software Engineering Notes* **19** (April, 1994), pp. 35–38.
- [Schach et al., 2002] S. R. SCHACH, B. JIN, D. R. WRIGHT, G. Z. HELLER, AND A. J. OFFUTT, “Maintainability of the Linux Kernel,” *IEE Proceedings—Software* **149** (February 2002), pp. 18–23.
- [Schach et al., 2003a] S. R. SCHACH, B. JIN, DAVID R. WRIGHT, G. Z. HELLER, AND J. OFFUTT, “Quality Impacts of Clandestine Common Coupling,” *Software Quality Journal* **11** (July 2003), pp. 211–18.
- [Schach and Stevens-Guille, 1979] S. R. SCHACH AND P. D. STEVENS-GUILLE, “Two Aspects of Computer-Aided Design,” *Transactions of the Royal Society of South Africa* **44** (Part 1, 1979), pp. 123–26.
- [Shneiderman and Mayer, 1975] B. SHNEIDERMAN AND R. MAYER, “Towards a Cognitive Model of Programmer Behavior,” Technical Report TR-37, Indiana University, Bloomington, 1975.
- [Snyder, 1993] A. SNYDER, “The Essence of Objects: Concepts and Terms,” *IEEE Software* **10** (January 1993), pp. 31–42.
- [Stevens, Myers, and Constantine, 1974] W. P. STEVENS, G. J. MYERS, AND L. L. CONSTANTINE, “Structured Design,” *IBM Systems Journal* **13** (No. 2, 1974), pp. 115–39.
- [Stroustrup, 2000] B. STROUSTRUP, *The C++ Programming Language*, Special 3rd ed., Addison Wesley, Reading, MA, 2000.
- [Yourdon and Constantine, 1979] E. YOURDON AND L. L. CONSTANTINE, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, Englewood Cliffs, NJ, 1979.