

# 6

## Working Classes

### Contents

- 6.1 Class Foundations: Abstract Data Types (ADTs)
- 6.2 Good Class Interfaces
- 6.3 Design and Implementation Issues
- 6.4 Reasons to Create a Class
- 6.5 Language-Specific Issues
- 6.6 Beyond Classes: Packages

### Related Topics

- Design in construction: Chapter 5
- Software architecture: Section 3.5
- Characteristics of high-quality routines: Chapter 7
- The Pseudocode Programming Process: Chapter 9
- Refactoring: Chapter 24

In the dawn of computing, programmers thought about programming in terms of statements. Throughout the 1970s and 1980s, programmers began thinking about programs in terms of routines. In the twenty-first century, programmers think about programming in terms of classes.

#### KEY POINT

A class is a collection of data and routines that share a cohesive, well-defined responsibility. A class might also be a collection of routines that provides a cohesive set of services even if no common data is involved. A key to being an effective programmer is maximizing the portion of a program that you can safely ignore while working on any one section of code. Classes are the primary tool for accomplishing that objective.

This chapter contains a distillation of advice in creating high quality classes. If you're still warming up to object-oriented concepts, this chapter might be too advanced. Make sure you've read Chapter 5. Then start with Section 6.1, "Abstract Data Types (ADTs)," and ease your way into the remaining sections.

If you're already familiar with class basics, you might skim Section 6.1 and then dive into the discussion of good class interfaces in Section 6.2. The "Additional Resources" section at the end of the chapter contains pointers to introductory reading, advanced reading, and programming-language-specific resources.

## 6.1 Class Foundations: Abstract Data Types (ADTs)

An abstract data type is a collection of data and operations that work on that data. The operations both describe the data to the rest of the program and allow the rest of the program to change the data. The word "data" in "abstract data type" is used loosely. An ADT might be a graphics window with all the operations that affect it; a file and file operations; an insurance-rates table and the operations on it; or something else.

**CROSS-REFERENCE** Thinking about ADTs first and classes second is an example of programming *into* a language vs. programming in one. Section 4.3, "Your Location on the Technology Wave" and Section 34.4, "Program Into Your Language, Not In It."

Understanding ADTs is essential to understanding object-oriented programming. Without understanding ADTs, programmers create classes that are "classes" in name only—in reality, they are little more than convenient carrying cases for loosely related collections of data and routines. With an understanding of ADTs, programmers can create classes that are easier to implement initially and easier to modify over time.

Traditionally, programming books wax mathematical when they arrive at the topic of abstract data types. They tend to make statements like "One can think of an abstract data type as a mathematical model with a collection of operations defined on it." Such books make it seem as if you'd never actually use an abstract data type except as a sleep aid.

Such dry explanations of abstract data types completely miss the point. Abstract data types are exciting because you can use them to manipulate real-world entities rather than low-level, implementation entities. Instead of inserting a node into a linked list, you can add a cell to a spreadsheet, a new type of window to a list of window types, or another passenger car to a train simulation. Tap into the power of being able to work in the problem domain rather than at the low-level implementation domain!

### Example of the Need for an ADT

To get things started, here's an example of a case in which an ADT would be useful. We'll get to the theoretical details after we have an example to talk about.

Suppose you're writing a program to control text output to the screen using a variety of typefaces, point sizes, and font attributes (such as bold and italic). Part

65 of the program manipulates the text's fonts. If you use an ADT, you'll have a  
66 group of font routines bundled with the data—the typeface names, point sizes,  
67 and font attributes—they operate on. The collection of font routines and data is  
68 an ADT.

69 If you're not using ADTs, you'll take an ad hoc approach to manipulating fonts.  
70 For example, if you need to change to a 12-point font size, which happens to be  
71 16 pixels high, you'll have code like this:

72 `currentFont.size = 16`

73 If you've built up a collection of library routines, the code might be slightly  
74 more readable:

75 `currentFont.size = PointsToPixels( 12 )`

76 Or you could provide a more specific name for the attribute, something like

77 `currentFont.sizeInPixels = PointsToPixels( 12 )`

78 But what you can't do is have both *currentFont.sizeInPixels* and  
79 *currentFont.sizeInPoints*, because, if both the data members are in play,  
80 *currentFont* won't have any way to know which of the two it should use.

81 If you change sizes in several places in the program, you'll have similar lines  
82 spread throughout your program.

83 If you need to set a font to bold, you might have code like this:

84 `currentFont.attribute = currentFont.attribute or 0x02`

85 If you're lucky, you'll have something cleaner than that, but the best you'll get  
86 with an ad hoc approach is something like this:

87 `currentFont.attribute = currentFont.attribute or BOLD`

88 Or maybe something like this:

89 `currentFont.bold = True`

90 As with the font size, the limitation is that the client code is required to control  
91 the data members directly, which limits how *currentFont* can be used.

92 If you program this way, you're likely to have similar lines in many places in  
93 your program.

## 94 Benefits of Using ADTs

95 The problem isn't that the ad hoc approach is bad programming practice. It's that  
96 you can replace the approach with a better programming practice that produces  
97 these benefits:

98                   ***You can hide implementation details***  
 99                   Hiding information about the font data type means that if the data type changes,  
 100                  you can change it in one place without affecting the whole program. For  
 101                  example, unless you hid the implementation details in an ADT, changing the  
 102                  data type from the first representation of bold to the second would entail  
 103                  changing your program in every place in which bold was set rather than in just  
 104                  one place. Hiding the information also protects the rest of the program if you  
 105                  decide to store data in external storage rather than in memory or to rewrite all the  
 106                  font-manipulation routines in another language.

107                  ***Changes don't affect the whole program***  
 108                  If fonts need to become richer and support more operations (such as switching to  
 109                  small caps, superscripts, strikethrough, and so on), you can change the program  
 110                  in one place. The change won't affect the rest of the program.

111                  ***You can make the interface more informative***  
 112                  Code like *currentFont.size = 16* is ambiguous because 16 could be a size in  
 113                  either pixels or points. The context doesn't tell you which is which. Collecting  
 114                  all similar operations into an ADT allows you to define the entire interface in  
 115                  terms of points, or in terms of pixels, or to clearly differentiate between the two,  
 116                  which helps avoid confusing them.

117                  ***It's easier to improve performance***  
 118                  If you need to improve font performance, you can recode a few well-defined  
 119                  routines rather than wading through an entire program.

120                  ***The program is more obviously correct***  
 121                  You can replace the more tedious task of verifying that statements like  
 122                  *currentFont.attribute = currentFont.attribute or 0x02* are correct with the easier  
 123                  task of verifying that calls to *currentFont.BoldOn()* are correct. With the first  
 124                  statement, you can have the wrong structure name, the wrong field name, the  
 125                  wrong logical operation (a logical *and* instead of *or*), or the wrong value for the  
 126                  attribute (*0x20* instead of *0x02*). In the second case, the only thing that could  
 127                  possibly be wrong with the call to *currentFont.BoldOn()* is that it's a call to the  
 128                  wrong routine name, so it's easier to see whether it's correct.

129                  ***The program becomes more self-documenting***  
 130                  You can improve statements like *currentFont.attribute or 0x02* by replacing  
 131                  *0x02* with *BOLD* or whatever *0x02* represents, but that doesn't compare to the  
 132                  readability of a routine call such as *currentFont.BoldOn()*.

133 **HARD DATA**                   Woodfield, Dunsmore, and Shen conducted a study in which graduate and senior  
 134   undergraduate computer-science students answered questions about two  
 135   programs—one that was divided into eight routines along functional lines and  
 136   one that was divided into eight abstract-data-type routines (1981). Students using

137 the abstract-data-type program scored over 30 percent higher than students using  
138 the functional version.

139 ***You don't have to pass data all over your program***

140 In the examples just presented, you have to change *currentFont* directly or pass  
141 it to every routine that works with fonts. If you use an abstract data type, you  
142 don't have to pass *currentFont* all over the program and you don't have to turn it  
143 into global data either. The ADT has a structure that contains *currentFont*'s data.  
144 The data is directly accessed only by routines that are part of the ADT. Routines  
145 that aren't part of the ADT don't have to worry about the data.

146 ***You're able to work with real-world entities rather than with low-level  
147 implementation structures***

148 You can define operations dealing with fonts so that most of the program  
149 operates solely in terms of fonts rather than in terms of array accesses, structure  
150 definitions, and *True* and *False* booleans.

151 In this case, to define an abstract data type, you'd define a few routines to  
152 control fonts—perhaps these:

```
153         currentFont.SetSizeInPoints( sizeInPoints )  
154         currentFont.SetSizeInPixels( sizeInPixels )  
155         currentFont.BoldOn()  
156         currentFont.BoldOff()  
157         currentFont.ItalicOn()  
158         currentFont.ItalicOff()  
159         currentFont.SetTypeFace( faceName )
```

160 **KEY POINT**

161 The code inside these routines would probably be short—it would probably be  
162 similar to the code you saw in the ad hoc approach to the font problem earlier.  
163 The difference is that you've isolated font operations in a set of routines. That  
164 provides a better level of abstraction for the rest of your program to work with  
fonts, and it gives you a layer of protection against changes in font operations.

165 **More Examples of ADTs**

166 Here are a few more examples of ADTs:

167 Suppose you're writing software that controls the cooling system for a nuclear  
168 reactor. You can treat the cooling system as an abstract data type by defining the  
169 following operations for it:

```
170         coolingSystem.Temperature()  
171         coolingSystem.SetCirculationRate( rate )  
172         coolingSystem.OpenValve( valveNumber )  
173         coolingSystem.CloseValve( valveNumber )
```

174 The specific environment would determine the code written to implement each  
 175 of these operations. The rest of the program could deal with the cooling system  
 176 through these functions and wouldn't have to worry about internal details of  
 177 data-structure implementations, data-structure limitations, changes, and so on.

178 Here are more examples of abstract data types and likely operations on them:

<i>Cruise Control</i>	<i>Blender</i>	<i>Fuel Tank</i>
Set speed	Turn on	Fill tank
Get current settings	Turn off	Drain tank
Resume former speed	Set speed	Get tank capacity
Deactivate	Start "Insta-Pulverize"	Get tank status
	Stop "Insta-Pulverize"	
<i>Set of Help Screens</i>		<i>Stack</i>
Add help topic	<i>Menu</i>	Initialize stack
Remove help topic	Start new menu	Push item onto stack
Set current help topic	Delete menu	Pop item from stack
Display help screen	Add menu item	Read top of stack
Remove help display	Remove menu item	
Display help index	Activate menu item	<i>File</i>
Back up to previous screen	Deactivate menu item	Open file
	Display menu	Read file
<i>List</i>	Hide menu	Write file
Initialize list	Get menu choice	Set current file location
Insert item in list		Close file
Remove item from list	<i>Pointer</i>	
Read next item from list	Get pointer to new memory	<i>Elevator</i>
	Dispose of memory from existing pointer	Move up one floor
<i>Light</i>	Change amount of memory allocated	Move down one floor
Turn on		Move to specific floor

Turn off

Report current floor

Return to home floor

You can derive several guidelines from a study of these examples:

***Build or use typical low-level data types as ADTs, not as low-level data types***

Most discussions of ADTs focus on representing typical low-level data types as ADTs. As you can see from the examples, you can represent a stack, a list, and a queue, as well as virtually any other typical data type, as an ADTs.

The question you need to ask is, What does this stack, list, or queue represent? If a stack represents a set of employees, treat the ADT as employees rather than as a stack. If a list represents a set of billing records, treat it as billing records rather than a list. If a queue represents cells in a spreadsheet, treat it as a collection of cells rather than a generic item in a queue. Treat yourself to the highest possible level of abstraction.

***Treat common objects such as files as ADTs***

Most languages include a few abstract data types that you're probably familiar with but might not think of as ADTs. File operations are a good example. While writing to disk, the operating system spares you the grief of positioning the read/write head at a specific physical address, allocating a new disk sector when you exhaust an old one, and checking for binary error codes. The operating system provides a first level of abstraction and the ADTs for that level. High-level languages provide a second level of abstraction and ADTs for that higher level. A high-level language protects you from the messy details of generating operating-system calls and manipulating data buffers. It allows you to treat a chunk of disk space as a "file."

You can layer ADTs similarly. If you want to use an ADT at one level that offers data-structure level operations (like pushing and popping a stack), that's fine. You can create another level on top of that one that works at the level of the real-world problem.

***Treat even simple items as ADTs***

You don't have to have a formidable data type to justify using an abstract data type. One of the ADTs in the example list is a light that supports only two operations—turning it on and turning it off. You might think that it would be a waste to isolate simple "on" and "off" operations in routines of their own, but even simple operations can benefit from the use of ADTs. Putting the light and its operations into an ADT makes the code more self-documenting and easier to change, confines the potential consequences of changes to the *TurnLightOn()*

214 and *TurnLightOff()* routines, and reduces the amount of data you have to pass  
215 around.

216 ***Refer to an ADT independently of the medium it's stored on***

217 Suppose you have an insurance-rates table that's so big that it's always stored on  
218 disk. You might be tempted to refer to it as a "rate *file*" and create access  
219 routines such as *rateFile.Read()*. When you refer to it as a file, however, you're  
220 exposing more information about the data than you need to. If you ever change  
221 the program so that the table is in memory instead of on disk, the code that refers  
222 to it as a file will be incorrect, misleading, and confusing. Try to make the names  
223 of classes and access routines independent of how the data is stored, and refer to  
224 the abstract data type, like the insurance-rates table, instead. That would give  
225 your class and access routine names like *rateTable.Read()* or simply  
226 *rates.Read()*.

227 **Handling Multiple Instances of Data with ADTs in**  
228 **Non-OO Environments**

229 Object-oriented languages provide automatic support for handling multiple  
230 instances of an ADT. If you've worked exclusively in object-oriented  
231 environments and have never had to handle the implementation details of  
232 multiple instances yourself, count your blessings! (You can also move on to the  
233 next section, "ADTs and Classes")

234 If you're working in a non-object oriented environment such as C, you will have  
235 to build support for multiple instances manually. In general, that means  
236 including services for the ADT to create and delete instances and designing the  
237 ADT's other services so that they can work with multiple instances.

238 The font ADT originally offered these services:

```
239         currentFont.SetSize( sizeInPoints )  
240         currentFont.BoldOn()  
241         currentFont.BoldOff()  
242         currentFont.ItalicOn()  
243         currentFont.ItalicOff()  
244         currentFont.SetTypeFace( faceName )
```

245 In a non-OO environment, these functions would not be attached to a class, and  
246 would look more like this:

```
247         SetCurrentFontSize( sizeInPoints )  
248         SetCurrentFontBoldOn()  
249         SetCurrentFontBoldOff()  
250         SetCurrentFontItalicOn()  
251         SetCurrentFontItalicOff()  
252         SetCurrentFontTypeFace( faceName )
```



253 If you want in work with more than one font at a time, you'll need to add  
254 services to create and delete font instances—maybe these:

```
255         CreateFont( fontId )  
256         DeleteFont( fontId )  
257         SetCurrentFont( fontId )
```

258 The notion of a *fontId* has been added as a way to keep track of multiple fonts as  
259 they're created and used. For other operations, you can choose from among three  
260 ways to handle the ADT interface:

261 ***Option 1: Use implicit instances (with great care)***

262 Design a new service to call to make a specific font instance the current one—  
263 something like *SetCurrentFont( fontId )*. Setting the current font makes all other  
264 services use the current font when they're called. If you use this approach, you  
265 don't need *fontId* as a parameter to the other services. For simple applications  
266 this can streamline use of multiple instances. For complex applications, this  
267 system-wide dependence on state means that you must keep track of the current  
268 font instance throughout code that uses the *Font* functions. Complexity tends to  
269 proliferate, and for applications of any size, there are better alternatives.

270 ***Option 2: Explicitly identify instances each time you use ADT services***

271 In this case, you don't have the notion of a "current font." You pass *fontId* to  
272 each routine that manipulates fonts. The *Font* functions keep track of any  
273 underlying data, and the client code needs to keep track only of the *fontId*. This  
274 requires adding *fontId* as a parameter to each font routine.

275 ***Option 3: Explicitly provide the data used by the ADT services***

276 In this approach, you declare the data that the ADT uses within each routine that  
277 uses an ADT service. In other words, you create a *Font* data type that you pass to  
278 each of the ADT service routines. You must design the ADT service routines so  
279 that they use the *Font* data that's passed to them each time they're called. The  
280 client code doesn't need a font ID if you use this approach because it keeps track  
281 of the font data itself. (Even though the data is available directly from the *Font*  
282 data type, you should access it only with the ADT service routines. This is called  
283 keeping the structure "closed."

284 The advantage of this approach is that the ADT service routines don't have to  
285 look up font information based on a font ID. The disadvantage is that it exposes  
286 font data to the rest of the program, which increases the likelihood that client  
287 code will make use of the ADT's implementation details that should have  
288 remained hidden within the ADT.

289 Inside the abstract data type, you'll have a wealth of options for handling  
290 multiple instances, but outside, this sums up the choices if you're working in a  
291 non-object oriented language.

## ADTs and Classes

Abstract data types form the foundation for the concept of classes. In languages that support classes, you can implement each abstract data type in its own class. Classes usually involve the additional concepts of inheritance and polymorphism. One way of thinking of a class is as an abstract data type plus inheritance and polymorphism.

## 6.2 Good Class Interfaces

The first and probably most important step in creating a high quality class is creating a good interface. This consists of creating a good abstraction for the interface to represent and ensuring the details remain hidden behind the abstraction.

### Good Abstraction

As “Form Consistent Abstractions” in Section 5.3 discussed, abstraction is the ability to view a complex operation in a simplified form. A class interface provides an abstraction of the implementation that’s hidden behind the interface. The class’s interface should offer a group of routines that clearly belong together.

You might have a class that implements an employee. It would contain data describing the employee’s name, address, phone number, and so on. It would offer services to initialize and use an employee. Here’s how that might look.

---

### C++ Example of a Class Interface that Presents a Good Abstraction

```
class Employee {
public:
    // public constructors and destructors
    Employee();
    Employee(
        FullName name,
        String address,
        String workPhone,
        String homePhone,
        TaxId taxIdNumber,
        JobClassification jobClass
    );
    virtual ~Employee();

    // public routines
    FullName Name();
```

**CROSS-REFERENCE** Code samples in this book are formatted using a coding convention that emphasizes similarity of styles across multiple languages. For details on the convention (and discussions about multiple coding styles), see “Mixed-Language Programming Considerations” in Section 11.4.

```

329     String Address();
330     String WorkPhone();
331     String HomePhone();
332     TaxId TaxIdNumber();
333     JobClassification GetJobClassification();
334     ...
335 private:
336     ...
337 }

```

Internally, this class might have additional routines and data to support these services, but users of the class don't need to know anything about them. The class interface abstraction is great because every routine in the interface is working toward a consistent end.

A class that presents a poor abstraction would be one that contained a collection of miscellaneous functions. Here's an example:

#### 344 CODING HORROR

#### C++ Example of a Class Interface that Presents a Poor Abstraction

```

345 class Program {
346 public:
347     ...
348     // public routines
349     void InitializeCommandStack();
350     void PushCommand( Command &command );
351     Command PopCommand();
352     void ShutdownCommandStack();
353     void InitializeReportFormatting();
354     void FormatReport( Report &report );
355     void PrintReport( Report &report );
356     void InitializeGlobalData();
357     void ShutdownGlobalData();
358     ...
359 private:
360     ...
361 }

```

Suppose that a class contains routines to work with a command stack, format reports, print reports, and initialize global data. It's hard to see any connection among the command stack and report routines or the global data. The class interface doesn't present a consistent abstraction, so the class has poor cohesion. The routines should be reorganized into more-focused classes, each of which provides a better abstraction in its interface.

If these routines were part of a "Program" class, they could be revised to present a consistent abstraction.

370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398

### C++ Example of a Class Interface that Presents a Better Abstraction

```
class Program {
public:
    ...
    // public routines
    void InitializeProgram();
    void ShutDownProgram();
    ...
private:
    ...
}
```

The cleanup of this interface assumes that some of these routines were moved to other, more appropriate classes and some were converted to private routines used by *InitializeProgram()* and *ShutDownProgram()*.

This evaluation of class abstraction is based on the class's collection of public routines, that is, its class interface. The routines inside the class don't necessarily present good individual abstractions just because the overall class does, but they need to be designed to present good abstractions, too. For guidelines on that, see Section 7.2, "Design at the Routine Level."

The pursuit of good, abstract interfaces gives rise to several guidelines for creating class interfaces.

#### ***Present a consistent level of abstraction in the class interface***

A good way to think about a class is as the mechanism for implementing the abstract data types (ADTs) described in Section 6.1. Each class should implement one and only one ADT. If you find a class implementing more than one ADT, or if you can't determine what ADT the class implements, it's time to reorganize the class into one or more well-defined ADTs.

Here's an example of a class the presents an interface that's inconsistent because its level of abstraction is not uniform:

### **CODING HORROR**

### C++ Example of a Class Interface with Mixed Levels of Abstraction

```
class EmployeeList: public ListContainer {
public:
    ...
    // public routines
    void AddEmployee( Employee &employee );
    void RemoveEmployee( Employee &employee );

    Employee NextItemInList( Employee &employee );
    Employee FirstItem( Employee &employee );
    Employee LastItem( Employee &employee );
```

The abstraction of these routines is at the "employee" level.

The abstraction of these routines is at the "list" level.

410  
411  
412  
413  
414  
415  
416  
417  
418  
419

```
...
private:
...
}
```

This class is presenting two ADTs: an *Employee* and a *ListContainer*. This sort of mixed abstraction commonly arises when a programmer uses a container class or other library classes for implementation and doesn't hide the fact that a library class is used. Ask yourself whether the fact that a container class is used should be part of the abstraction. Usually that's an implementation detail that should be hidden from the rest of the program, like this:

420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437

The abstraction of all these  
routines is now at the  
"employee" level.

That the class uses the  
*ListContainer* library is now  
hidden

### C++ Example of a Class Interface with Consistent Levels of Abstraction

```
class EmployeeList {
public:
...
// public routines
void AddEmployee( Employee &employee );
void RemoveEmployee( Employee &employee );
Employee NextEmployee( Employee &employee );
Employee FirstEmployee( Employee &employee );
Employee LastEmployee( Employee &employee );
...
private:
    ListContainer m_EmployeeList;
    ...
}
```

Programmers might argue that inheriting from *ListContainer* is convenient because it supports polymorphism, allowing an external search or sort function that takes a *ListContainer* object.

That argument fails the main test for inheritance, which is, Is inheritance used only for "is a" relationships? To inherit from *ListContainer* would mean that *EmployeeList* "is a" *ListContainer*, which obviously isn't true. If the abstraction of the *EmployeeList* object is that it can be searched or sorted, that should be incorporated as an explicit, consistent part of the class interface.

If you think of the class's public routines as an air lock that keeps water from getting into a submarine, inconsistent public routines are leaky panels in the class. The leaky panels might not let water in as quickly as an open air lock, but if you give them enough time, they'll still sink the boat. In practice, this is what happens when you mix levels of abstraction. As the program is modified, the mixed levels of abstraction make the program harder and harder to understand, and it gradually degrades until it becomes unmaintainable.

#### KEY POINT

***Be sure you understand what abstraction the class is implementing***

Some classes are similar enough that you must be careful to understand which abstraction the class interface should capture. I once worked on a program that needed to allow information to be edited in a table format. We wanted to use a simple grid control, but the grid controls that were available didn't allow us to color the data-entry cells, so we decided to use a spreadsheet control that did provide that capability.

The spreadsheet control was far more complicated than the grid control, providing about 150 routines to the grid control's 15. Since our goal was to use a grid control, not a spreadsheet control, we assigned a programmer to write a wrapper class to hide the fact that we were using a spreadsheet control as a grid control. The programmer grumbled quite a bit about unnecessary overhead and bureaucracy, went away, and came back a couple days later with a wrapper class that faithfully exposed all 150 routines of the spreadsheet control.

This was not what was needed. We wanted a grid-control interface that encapsulate the fact that, behind the scenes, we were using a much more complicated spreadsheet control. The programmer should have exposed just the 15 grid control routines plus a 16th routine that supported cell coloring. By exposing all 150 routines, the programmer created the possibility that, if we ever wanted to change the underlying implementation, we could find ourselves supporting 150 public routines. The programmer failed to achieve the encapsulation we were looking for, as well as creating a lot more work for himself than necessary.

Depending on specific circumstances, the right abstraction might be either a spreadsheet control or a grid control. When you have to choose between two similar abstractions, make sure you choose the right one.

***Provide services in pairs with their opposites***

Most operations have corresponding, equal, and opposite operations. If you have an operation that turns a light on, you'll probably need one to turn it off. If you have an operation to add an item to a list, you'll probably need one to delete an item from the list. If you have an operation to activate a menu item, you'll probably need one to deactivate an item. When you design a class, check each public routine to determine whether you need its complement. Don't create an opposite gratuitously, but do check to see whether you need one.

***Move unrelated information to another class***

In some cases, you'll find that half a class's routines work with half the class's data, and half the routines work with the other half of the data. In such a case, you really have two classes masquerading as one. Break them up!

488  
 489 **CROSS-REFERENCE** For  
 490 more suggestions about how  
 491 to preserve code quality as  
 492 code is modified, See  
 Chapter 24, “Refactoring.”

### 493 **CODING HORROR**

```

494 class Employee {
495 public:
496     ...
497     // public routines
498     FullName GetName();
499     Address GetAddress();
500     PhoneNumber GetWorkPhone();
501     ...
502     Boolean IsJobClassificationValid( JobClassification jobClass );
503     Boolean IsZipCodeValid( Address address );
504     Boolean IsPhoneNumberValid( PhoneNumber phoneNumber );
505
506     SqlQuery GetQueryToCreateNewEmployee();
507     SqlQuery GetQueryToModifyEmployee();
508     SqlQuery GetQueryToRetrieveEmployee();
509     ...
510 private:
511     ...
512 }
  
```

### *Beware of erosion of the interface’s abstraction under modification*

As a class is modified and extended, you often discover additional functionality that’s needed, that doesn’t quite fit with the original class interface, but that seems too hard to implement any other way. For example, in the *Employee* class, you might find that the class evolves to look like this:

### **C++ Example of a Class Interface that’s Eroding Under Maintenance**

```

class Employee {
public:
    ...
    // public routines
    FullName GetName();
    Address GetAddress();
    PhoneNumber GetWorkPhone();
    ...
    Boolean IsJobClassificationValid( JobClassification jobClass );
    Boolean IsZipCodeValid( Address address );
    Boolean IsPhoneNumberValid( PhoneNumber phoneNumber );

    SqlQuery GetQueryToCreateNewEmployee();
    SqlQuery GetQueryToModifyEmployee();
    SqlQuery GetQueryToRetrieveEmployee();
    ...
private:
    ...
}
  
```

What started out as a clean abstraction in an earlier code sample has evolved into a hodgepodge of functions that are only loosely related. There’s no logical connection between employees and routines that check zip codes, phone numbers, or job classifications. The routines that expose SQL query details are at a much lower level of abstraction than the *Employee* class, and they break the *Employee* abstraction.

### *Don’t add public members that are inconsistent with the interface abstraction*

Each time you add a routine to a class interface, ask, “Is this routine consistent with the abstraction provided by the existing interface?” If not, find a different way to make the modification, and preserve the integrity of the abstraction.

### *Consider abstraction and cohesion together*

The ideas of abstraction and cohesion are closely related—a class interface that presents a good abstraction usually has strong cohesion. Classes with strong cohesion tend to present good abstractions, although that relationship is not as strong.

I have found that focusing on the abstraction presented by the class interface tends to provide more insight into class design than focusing on class cohesion. If you see that a class has weak cohesion and aren't sure how to correct it, ask yourself whether the class presents a consistent abstraction instead.

## Good Encapsulation

**CROSS-REFERENCE** For more on encapsulation, see “Encapsulate Implementation Details” in Section 5.3.

As Section 5.3 discussed, encapsulation is a stronger concept than abstraction. Abstraction helps to manage complexity by providing models that allow you to ignore implementation details. Encapsulation is the enforcer that prevents you from looking at the details even if you want to.

The two concepts are related because, without encapsulation, abstraction tends to break down. In my experience either you have both abstraction and encapsulation, or you have neither. There is no middle ground.

*The single most important factor that distinguishes a well-designed module from a poorly designed one is the degree to which the module hides its internal data and other implementation details from other modules.*

### *Minimize accessibility of classes and members*

Minimizing accessibility is one of several rules that are designed to encourage encapsulation. If you're wondering whether a specific routine should be public, private, or protected, one school of thought is that you should favor the strictest level of privacy that's workable (Meyers 1998, Bloch 2001). I think that's a fine guideline, but I think the more important guideline is, “What best preserves the integrity of the interface abstraction?” If exposing the routine is consistent with the abstraction, it's probably fine to expose it. If you're not sure, hiding more is generally better than hiding less.

### *Don't expose member data in public*

Exposing member data is a violation of encapsulation and limits your control over the abstraction. As Arthur Riel points out, a *Point* class that exposes

```
float x;
float y;
float z;
```

is violating encapsulation because client code is free to monkey around with *Point*'s data, and *Point* won't necessarily even know when its values have been changed (Riel 1996). However, a *Point* class that exposes

```
float X();
float Y();
float Z();
void SetX( float x );
void SetY( float y );
void SetZ( float z );
```

is maintaining perfect encapsulation. You have no idea whether the underlying implementation is in terms of *floats* *x*, *y*, and *z*, whether *Point* is storing those



567 items as *doubles* and converting them to *floats*, or whether *Point* is storing them  
568 on the moon and retrieving them from a satellite in outer space.

569 ***Don't put private implementation details in a class's interface***

570 With true encapsulation, programmers would not be able to see implementation  
571 details at all. They would be hidden both figuratively and literally.

572 In popular languages like C++, however, the structure of the language requires  
573 programmers to disclose implementation details in the class interface. Here's an  
574 example:

---

575 **C++ Example of Inadvertently Exposing a Class's Implementation**  
576 **Details**

```
577 class Employee {  
578 public:  
579     ...  
580     Employee(  
581         FullName name,  
582         String address,  
583         String workPhone,  
584         String homePhone,  
585         TaxId taxIdNumber,  
586         JobClassification jobClass  
587     );  
588     ...  
589     FullName Name();  
590     String Address();  
591     ...  
592 private:  
593     String m_Name;  
594     String m_Address;  
595     int m_jobClass;  
596     ...  
597 }
```

593 *Here are the exposed*  
594 *implementation details.*

598 Including *private* declarations in the class header file might seem like a small  
599 transgression, but it encourages programmers to examine the implementation  
600 details. In this case, the client code is intended to use the *Address* type for  
601 addresses, but the header file exposes the implementation detail that addresses  
602 are stored as *Strings*.

603 As the writer of a class in C++, there isn't much you can do about this without  
604 going to great lengths that usually add more complexity than they're worth. As  
605 the *reader* of a class, however, you can resist the urge to comb through the  
606 *private* section of the class interface looking for implementation clues.

607  
608  
609  
610  
611

### ***Don't make assumptions about the class's users***

A class should be designed and implemented to adhere to the contract implied by the class interface. It shouldn't make any assumptions about how that interface will or won't be used, other than what's documented in the interface. Comments like this are an indication that a class is more aware of its users than it should be:

612  
613

```
-- initialize x, y, and z to 1.0 because DerivedClass blows
-- up if they're initialized to 0.0
```

614  
615  
616  
617  
618

### ***Avoid friend classes***

In a few circumstances such as the State pattern, friend classes can be used in a disciplined way that contributes to managing complexity (Gamma et al 1995). But, in general, friend classes violate encapsulation. They expand the amount of code you have to think about at any one time, increasing complexity.

619  
620  
621  
622  
623

### ***Don't put a routine into the public interface just because it uses only public routines***

The fact that a routine uses only public routines is not a very significant consideration. Instead, ask whether exposing the routine would be consistent with the abstraction presented by the interface.

624  
625  
626  
627  
628  
629  
630  
631  
632

### ***Favor read-time convenience to write-time convenience***

Code is read far more times than it's written, even during initial development. Favoring a technique that speeds write-time convenience at the expense of read-time convenience is a false economy. This is especially applicable to creation of class interfaces. Even if a routine doesn't quite fit the interface's abstraction, sometimes it's tempting to add a routine to an interface that would be convenient for the particular client of a class that you're working on at the time. But adding that routine is the first step down a slippery slope, and it's better not to take even the first step.

633 *It ain't abstract if you*  
634 *have to look at the*  
635 *underlying*  
636 *implementation to*  
637 *understand what's going*  
*on.*

638 —*P.J. Plauger*

639  
640  
641  
642  
643

### ***Be very, very wary of semantic violations of encapsulation***

At one time I thought that when I learned how to avoid syntax errors I would be home free. I soon discovered that learning how to avoid syntax errors had merely bought me a ticket to a whole new theater of coding errors—most of which were more difficult to diagnose and correct than the syntax errors.

The difficulty of semantic encapsulation compared to syntactic encapsulation is similar. Syntactically, it's relatively easy to avoid poking your nose into the internal workings of another class just by declaring the class's internal routines and data *private*. Achieving semantic encapsulation is another matter entirely. Here are some examples of the ways that a user of a class can break encapsulation semantically:

- 644 • Not calling Class A's *Initialize()* routine because you know that Class A's
- 645 *PerformFirstOperation()* routine calls it automatically.
- 646 • Not calling the *database.Connect()* routine before you call
- 647 *employee.Retrieve( database )* because you know that the
- 648 *employee.Retrieve()* function will connect to the database if there isn't
- 649 already a connection.
- 650 • Not calling Class A's *Terminate()* routine because you know that Class A's
- 651 *PerformFinalOperation()* routine has already called it.
- 652 • Using a pointer or reference to *ObjectB* created by *ObjectA* even after
- 653 *ObjectA* has gone out of scope, because you know that *ObjectA* keeps
- 654 *ObjectB* in *static* storage, and *ObjectB* will still be valid.
- 655 • Using ClassB's *MAXIMUM\_ELEMENTS* constant instead of using
- 656 *ClassA.MAXIMUM\_ELEMENTS*, because you know that they're both equal
- 657 to the same value.

#### 658 KEY POINT

659 The problem with each of these examples is that they make the client code

660 dependent not on the class's public interface, but on its private implementation.

661 Anytime you find yourself looking at a class's implementation to figure out how

662 to use the class, you're not programming to the interface; you're programming

663 *through* the interface *to* the implementation. If you're programming through the

664 interface, encapsulation is broken, and once encapsulation starts to break down,

665 If you can't figure out how to use a class based solely on its interface

666 documentation, the right response is *not* to pull up the source code and look at

667 the implementation. That's good initiative but bad judgment. The right response

668 is to contact the author of the class and say, "I can't figure out how to use this

669 class." The right response on the class-author's part is *not* to answer your

670 question face to face. The right response for the class author is to check out the

671 class-interface file, modify the class-interface documentation, check the file back

672 in, and then say, "See if you can understand how it works now." You want this

673 dialog to occur in the interface code itself so that it will be preserved for future

674 programmers. You don't want the dialog to occur solely in your own mind,

675 which will bake subtle semantic dependencies into the client code that uses the

676 class. And you don't want the dialog to occur interpersonally so that it benefits

677 only your code but no one else's.

#### 678 *Watch for coupling that's too tight*

679 "Coupling" refers to how tight the connection is between two classes. In general,

680 the looser the connection, the better. Several general guidelines flow from this

681 concept:

- 682 • Minimize accessibility of classes and members

- 683 • Avoid *friend* classes, because they're tightly coupled
- 684 • Avoid making data *protected* in a base class because it allows derived
- 685 classes to be more tightly coupled to the base class
- 686 • Avoid exposing member data in a class's public interface
- 687 • Be wary of semantic violations of encapsulation
- 688 • Observe the Law of Demeter (discussed later in this chapter)

689 Coupling goes hand in glove with abstraction and encapsulation. Tight coupling  
690 occurs when an abstraction is leaky, or when encapsulation is broken. If a class  
691 offers an incomplete set of services, other routines might find they need to read  
692 or write its internal data directly. That opens up the class, making it a glass box  
693 instead of a black box, and virtually eliminates the class's encapsulation.

## 694 6.3 Design and Implementation Issues

695 Defining good class interfaces goes a long way toward creating a high-quality  
696 program. The internal class design and implementation are also important. This  
697 section discusses issues related to containment, inheritance, member functions  
698 and data, class coupling, constructors, and value-vs.-reference objects.

### 699 Containment ("has a" relationships)

700 **KEY POINT**  
701 Containment is the simple idea that a class contains a primitive data element or  
702 object. A lot more is written about inheritance than about containment, but that's  
703 because inheritance is more tricky and error prone, not because it's better.  
Containment is the work-horse technique in object-oriented programming.

#### 704 *Implement "has a" through containment*

705 One way of thinking of containment is as a "has a" relationship. For example, an  
706 employee "has a" name, "has a" phone number, "has a" tax ID, and so on. You  
707 can usually accomplish this by making the name, phone number, or tax ID  
708 member data of the *Employee* class.

#### 709 *Implement "has a" through private inheritance as a last resort*

710 In some instances you might find that you can't achieve containment through  
711 making one object a member of another. In that case, some experts suggest  
712 privately inheriting from the contained object (Meyers 1998). The main reason  
713 you would do that is to set up the containing class to access protected member  
714 functions or data of the class that's contained. In practice, this approach creates  
715 an overly cozy relationship with the ancestor class and violates encapsulation. It  
716 tends to point to design errors that should be resolved some way other than  
717 through private inheritance.

***Be critical of classes that contain more than about seven members***

The number “7±2” has been found to be a number of discrete items a person can remember while performing other tasks (Miller 1956). If a class contains more than about seven data members, consider whether the class should be decomposed into multiple smaller classes (Riel 1996). You might err more toward the high end of 7±2 if the data members are primitive data types like integers and strings; more toward the lower end of 7±2 if the data members are complex objects.

**Inheritance (“is a” relationships)**

Inheritance is the complex idea that one class is a specialization of another class. Inheritance is perhaps the most distinctive attribute of object-oriented programming, and it should be used sparingly and with great caution. A great many of the problems in modern programming arise from overly enthusiastic use of inheritance.

The purpose of inheritance is to create simpler code by defining a base class that specifies common elements of two or more derived classes. The common elements can be routine interfaces, implementations, data members, or data types.

When you decide to use inheritance, you have to make several decisions:

- For each member routine, will the routine be visible to derived classes? Will it have a default implementation? Will the default implementation be overridable?
- For each data member (including variables, named constants, enumerations, and so on), will the data member be visible to derived classes?

The following subsections explain the ins and outs of making these decisions.

***Implement “is a” through public inheritance***

When a programmer decides to create a new class by inheriting from an existing class, that programmer is saying that the new class “is a” more specialized version of the older class. The base class sets expectations about how the derived class will operate (Meyers 1998).

If the derived class isn’t going to adhere *completely* to the same interface contract defined by the base class, inheritance is not the right implementation technique. Consider containment or making a change further up the inheritance hierarchy.

***The single most important rule in object-oriented programming with C++ is this: public inheritance means “isa.” Commit this rule to memory.***  
—Scott Meyers

***Design and document for inheritance or prohibit it***

Inheritance adds complexity to a program, and, as such, it is a dangerous technique. As Java guru Joshua Bloch says, “design and document for inheritance, or prohibit it.” If a class isn’t designed to be inherited from, make its members non-*virtual* in C++, *final* in Java, or non *overridable* in Visual Basic so that you can’t inherit from it.

***Adhere to the Liskov Substitution Principle***

In one of object-oriented programming’s seminal papers, Barbara Liskov argued that you shouldn’t inherit from a base class unless the derived class truly “is a” more specific version of the base class (Liskov 1988). Andy Hunt and Dave Thomas suggest a good litmus test for this: “Subclasses must be usable through the base class interface without the need for the user to know the difference” (Hunt and Thomas 2000).

In other words, all the routines defined in the base class should mean the same thing when they’re used in each of the derived classes.

If you have a base class of *Account*, and derived classes of *CheckingAccount*, *SavingsAccount*, and *AutoLoanAccount*, a programmer should be able to invoke any of the routines derived from *Account* on any of *Account*’s subtypes without caring about which subtype a specific account object is.

If a program has been written so that the Liskov Substitution Principle is true, inheritance is a powerful tool for reducing complexity because a programmer can focus on the generic attributes of an object without worrying about the details. If, a programmer must be constantly thinking about semantic differences in subclass implementations, then inheritance is increasing complexity rather than reducing it. Suppose a programmer has to think, “If I call the *InterestRate()* routine on *CheckingAccount* or *SavingsAccount*, it returns the interest the bank pays, but if I call *InterestRate()* on *AutoLoanAccount* I have to change the sign because it returns the interest the consumer pays to the bank.” According to Liskov, the *InterestRate()* routine should not be inherited because its semantics aren’t the same for all derived classes.

***Be sure to inherit only what you want to inherit***

A derived class can inherit member routine interfaces, implementations, or both. Table 6-1 shows the variations of how routines can be implemented and overridden.

Table 6-1. Variations on inherited routines

	Overridable	Not Overridable
Implementation: Default Provided	Overridable Routine	Non-Overridable Routine
Implementation: No default provided	Abstract Overridable Routine	Not used (doesn't make sense to leave a routine undefined and not allow it to be overridden)

As the table suggests, inherited routines come in three basic flavors:

- An *abstract overridable routine* means that the derived class inherits the routine's interface but not its implementation.
- An *overridable routine* means that the derived class inherits the routine's interface and a default implementation, and it is allowed to override the default implementation.
- A *non-overridable routine* means that the derived class inherits the routine's interface and its default implementation, and it is not allowed to override the routine's implementation.

When you choose to implement a new class through inheritance, think through the kind of inheritance you want for each member routine. Beware of inheriting implementation just because you're inheriting an interface, and beware of inheriting an interface just because you want to inherit an implementation.

***Don't "override" a non-overridable member function***

Both C++ and Java allow a programmer to override a non-overridable member routine—kind of. If a function is *private* in the base class, a derived class can create a function with the same name. To the programmer reading the code in the derived class, such a function can create confusion because it looks like it should be polymorphic, but it isn't; it just has the same name. Another way to state this guideline is, Don't reuse names of non-overridable base-class routines in derived classes.

***Move common interfaces, data, and behavior as high as possible in the inheritance tree***

The higher you move interfaces, data, and behavior, the more easily derived classes can use them. How high is too high? Let *abstraction* be your guide. If you find that moving a routine higher would break the higher object's abstraction, don't do it.

***Be suspicious of classes of which there is only one instance***

A single instance might indicate that the design confuses objects with classes. Consider whether you could just create an object instead of a new class. Can the

817 variation of the derived class be represented in data rather than as a distinct  
818 class?

819 ***Be suspicious of base classes of which there is only one derived class***

820 When I see a base class that has only one derived class, I suspect that some  
821 programmer has been “designing ahead”—trying to anticipate future needs,  
822 usually without fully understanding what those future needs are. The best way to  
823 prepare for future work is not to design extra layers of base classes that “might  
824 be needed someday,” it’s to make current work as clear, straightforward, and  
825 simple as possible. That means not creating any more inheritance structure than  
826 is absolutely necessary.

827 ***Be suspicious of classes that override a routine and do nothing inside the***  
828 ***derived routine***

829 This typically indicates an error in the design of the base class. For instance,  
830 suppose you have a class *Cat* and a routine *Scratch()* and suppose that you  
831 eventually find out that some cats are declawed and can’t scratch. You might be  
832 tempted to create a class derived from *Cat* named *ScratchlessCat* and override  
833 the *Scratch()* routine to do nothing. There are several problems with this  
834 approach:

- 835 • It violates the abstraction (interface contract) presented in the *Cat* class by  
836 changing the semantics of its interface.
- 837 • This approach quickly gets out of control when you extend it to other  
838 derived classes. What happens when you find a cat without a tail? Or a cat  
839 that doesn’t catch mice? Or a cat that doesn’t drink milk? Eventually you’ll  
840 end up with derived classes like *ScratchlessTaillessMicelessMilklessCat*.
- 841 • Over time, this approach gives rise to code that’s confusing to maintain  
842 because the interfaces and behavior of the ancestor classes imply little or  
843 nothing about the behavior of their descendents.

844 The place to fix this problem is not in the base class, but in the original *Cat* class.  
845 Create a *Claws* class and contain that within the *Cats* class, or build a constructor  
846 for the class that includes whether the cat scratches. The root problem was the  
847 assumption that all cats scratch, so fix that problem at the source, rather than just  
848 bandaging it at the destination.

849 ***Avoid deep inheritance trees***

850 Object oriented programming provides a large number of techniques for  
851 managing complexity. But every powerful tool has its hazards, and some object-  
852 oriented techniques have a tendency to increase complexity rather than reduce it.

853 In his excellent book *Object-Oriented Design Heuristics*, Arthur Riel suggests  
854 limiting inheritance hierarchies to a maximum of six levels (1996). Riel bases his



855 recommendation on the “magic number  $7\pm 2$ ,” but I think that’s grossly  
856 optimistic. In my experience most people have trouble juggling more than two or  
857 three levels of inheritance in their brains at once. The “magic number  $7\pm 2$ ” is  
858 probably better applied as a limit to the *total number of subclasses* of a base  
859 class rather than the number of levels in an inheritance tree.

860 Deep inheritance trees have been found to be significantly associated with  
861 increased fault rates (Basili, Briand, and Melo 1996). Anyone who has ever tried  
862 to debug a complex inheritance hierarchy knows why.

863 Deep inheritance trees increase complexity, which is exactly the opposite of  
864 what inheritance should be used to accomplish. Keep the primary technical  
865 mission in mind. Make sure you’re using inheritance to *minimize complexity*.

### 866 ***Prefer inheritance to extensive type checking***

867 Frequently repeated *case* statements sometimes suggest that inheritance might be  
868 a better design choice, although this is not always true. Here is a classic example  
869 of code that cries out for a more object-oriented approach:

---

### 870 **C++ Example of a Case Statement That Probably Should be Replaced 871 by Inheritance**

```
872 switch ( shape.type ) {  
873     case Shape_Circle:  
874         shape.DrawCircle();  
875         break;  
876     case Shape_Square:  
877         shape.DrawSquare();  
878         break;  
879     ...  
880 }
```

881 In this example, the calls to *shape.DrawCircle()* and *shape.DrawSquare()* should  
882 be replaced by a single routine named *shape.Draw()*, which can be called  
883 regardless of whether the shape is a circle or a square.

884 On the other hand, sometimes *case* statements are used to separate truly different  
885 kinds of objects or behavior. Here is an example of a *case* statement that is  
886 appropriate in an object-oriented program:

---

### 887 **C++ Example of a Case Statement That Probably Should not be 888 Replaced by Inheritance**

```
889 switch ( ui.Command() ) {  
890     case Command_OpenFile:  
891         OpenFile();  
892         break;
```

```

893     case Command_Print:
894         Print();
895         break;
896     case Command_Save:
897         Save();
898         break;
899     case Command_Exit:
900         ShutDown();
901         break;
902     ...
903 }

```

In this case, it would be possible to create a base class with derived classes and a polymorphic *DoCommand()* routine for each command. But the meaning of *DoCommand()* would be so diluted as to be meaningless, and the *case* statement is the more understandable solution.

***Avoid using a base class's protected data in a derived class (or make that data private instead of protected in the first place)***

As Joshua Bloch says, "Inheritance breaks encapsulation" (2001). When you inherit from an object, you obtain privileged access to that object's protected routines and data. If the derived class really needs access to the base class's attributes, provide protected accessor functions instead.

## Multiple Inheritance

Inheritance is a power tool. It's like using a chainsaw to cut down a tree instead of a manual cross-cut saw. It can be incredibly useful when used with care, but it's dangerous in the hands of someone who doesn't observe proper precautions.

If inheritance is a chain saw, multiple inheritance is a 1950s-era chain saw with no blade guard, not automatic shut off, and a finicky engine. There are times when such a tool is indispensable, mostly, you're better off leaving the tool in the garage where it can't do any damage.

Although some experts recommend broad use of multiple inheritance (Meyer 1997), in my experience multiple inheritance is useful primarily for defining "mixins," simple classes that are used to add a set of properties to an object. Mixins are called mixins because they allow properties to be "mixed in" to derived classes. Mixins might be classes like *Displayable*, *Persistent*, *Serializable*, or *Sortable*. Mixins are nearly always abstract and aren't meant to be instantiated independently of other objects.

Mixins require the use of multiple inheritance, but they aren't subject to the classic diamond-inheritance problem associated with multiple inheritance as long as all mixins are truly independent of each other. They also make the design more comprehensible by "chunking" attributes together. A programmer will

915 *The one indisputable fact*  
 916 *about multiple*  
 917 *inheritance in C++ is that*  
 918 *it opens up a Pandora's*  
 919 *box of complexities that*  
 920 *simply do not exist under*  
 921 *single inheritance.*  
 —Scott Meyers

933 have an easier time understanding that an object uses the mixins *Displayable* and  
934 *Persistent* than understanding that an object uses the 11 more specific routines  
935 that would otherwise be needed to implement those two properties.

936 Java and Visual Basic recognize the value of mixins by allowing multiple  
937 inheritance of interfaces but only single class inheritance. C++ supports multiple  
938 inheritance of both interface and implementation. Programmers should use  
939 multiple inheritance only after carefully considering the alternatives and  
940 weighing the impact on system complexity and comprehensibility.

## 941 Why Are There So Many Rules for Inheritance?

942 **CROSS-REFERENCE** For  
943 more on complexity, see  
944 “Software’s Primary  
945 Technical Imperative:  
946 Managing Complexity” in  
947 Section 5.2

This section has presented numerous rules for staying out of trouble with inheritance. The underlying message of all these rules is that, *inheritance tends to work against the primary technical imperative you have as a programmer, which is to manage complexity*. For the sake of controlling complexity you should maintain a heavy bias against inheritance. Here’s a summary of when to use inheritance and when to use containment:

- 948 • If multiple classes share common data but not behavior, then create a  
949 common object that those classes can contain.
- 950 • If multiple classes share common behavior but not data, then derive them  
951 from a common base class that defines the common routines.
- 952 • If multiple classes share common data and behavior, then inherit from a  
953 common base class that defines the common data and routines.
- 954 • Inherit when you want the base class to control your interface; contain when  
955 you want to control your interface.

## 956 Member Functions and Data

957 **CROSS-REFERENCE** For  
958 more discussion of routines  
in general, see Chapter 7,  
959 “High-Quality Routines.”

Here are a few guidelines for implementing member functions and member data effectively.

### 960 *Keep the number of routines in a class as small as possible*

961 A study of C++ programs found that higher numbers of routines per class were  
962 associated with higher fault rates (Basili, Briand, and Melo 1996). However,  
963 other competing factors were found to be more significant, including deep  
964 inheritance trees, large number of routines called by a routine, and strong  
965 coupling between classes. Evaluate the tradeoff between minimizing the number  
of routines and these other factors.

***Disallow implicitly generated member functions and operators you don't want***

Sometimes you'll find that you want to disallow certain functions—perhaps you want to disallow assignment, or you don't want to allow an object to be constructed. You might think that, since the compiler generates operators automatically, you're stuck allowing access. But in such cases you can disallow those uses by declaring the constructor, assignment operator, or other function or operator *private*, which will prevent clients from accessing it. (Making the constructor private is a standard technique for defining a singleton class, which is discussed later in this chapter.)

***Minimize direct routine calls to other classes***

One study found that the number of faults in a class was statistically correlated with the total number of routines that were called from within a class (Basili, Briand, and Melo 1996). The same study found that the more classes a class used, the higher its fault rate tended to be.

***Minimize indirect routine calls to other classes***

Direct connections are hazardous enough. Indirect connections—such as *account.ContactPerson().DaytimeContactInfo().PhoneNumber()*—tend to be even more hazardous. Researchers have formulated a rule called the “Law of Demeter” (Lieberherr and Holland 1989) which essentially states that Object A can call any of its own routines. If Object A instantiates an Object B, it can call any of Object B's routines. But it should avoid calling routines on objects provided by Object B. In the *account* example above, that means *account.ContactPerson()* is OK, but *account.ContactPerson().DaytimeContactInfo()* is not.

This is a simplified explanation, and, depending on how classes are arranged, it might be acceptable to see an expression like *account.ContactPerson().DaytimeContactInfo()*. See the additional resources at the end of this chapter for more details.

***In general, minimize the extent to which a class collaborates with other classes***

Try to minimize all of the following:

- Number of kinds of objects instantiated
- Number of different direct routine calls on instantiated objects
- Number of routine calls on objects returned by other instantiated objects

982 **FURTHER READING** Good  
983 accounts of the Law of  
984 Demeter can be found in  
985 *Pragmatic Programmer*  
986 (Hunt and Thomas 2000),  
987 *Applying UML and Patterns*  
988 (Larman 2001), and  
989 *Fundamentals of Object-*  
990 *Oriented Design in UML*  
(Page-Jones 2000).

## Constructors

Here are some guidelines that apply specifically to constructors. Guidelines for constructors are pretty similar across languages (C++, Java, and Visual Basic, anyway). Destructors vary more, and so you should check out the materials listed in the “Additional Resources” section at the end of the chapter for more information on destructors.

### *Initialize all member data in all constructors, if possible*

Initializing all data members in all constructors is an inexpensive defensive programming practice.

### *Initialize data members in the order in which they’re declared*

Depending on your compiler, you can experience some squirrely errors by trying to initialize data members in a different order than the order in which they’re declared. Using the same order in both places also provides consistency that makes the code easier to read.

### *Enforce the singleton property by using a private constructor*

If you want to define a class that allows only one object to be instantiated, you can enforce this by hiding all the constructors of the class, then providing a *static getInstance()* routine to access the class’s single instance. Here’s an example of how that would work:

---

#### Java Example of Enforcing a Singleton With a Private Constructor

```
public class MaxId {
    // constructors and destructors
    private MaxId() {
        ...
    }
    ...

    // public routines
    public static MaxId GetInstance() {
        return m_instance;
    }
    ...

    // private members
    private static final MaxId m_instance = new MaxId();
    ...
}
```

The private constructor is called only when the static object *m\_instance* is initialized. In this approach, if you want to reference the *MaxId* singleton, you would simply refer to *MaxId.GetInstance()*.

***Enforce the singleton property by using all static member data and reference counting***

An alternative means of enforcing the singleton property is to declare all the class's data static. You can determine whether the class is being used by incrementing a reference counter in the object's constructor and decrementing it in the destructor (C++) or *Terminate* routine (Java and Visual Basic).

The reference-counting approach comes with some systemic pitfalls. If the reference is copied, then the class data member won't necessarily be incremented, which can lead to an error in the reference count. If this approach is used, the project team should standardize on conventions to use reference-counted objects consistently.

***Prefer deep copies to shallow copies until proven otherwise***

One of the major decisions you'll make about complex objects is whether to implement deep copies or shallow copies of the object. A deep copy of an object is a member-wise copy of the object's member data. A shallow copy typically just points to or refers to a single reference copy.

Deep copies are simpler to code and maintain than shallow copies. In addition to the code either kind of object would contain, shallow copies add code to count references, ensure safe object copies, safe comparisons, safe deletes, and so on. This code tends to be error prone, and it should be avoided unless there's a compelling reason to create it.

The motivation for creating shallow copies is typically to improve performance. Although creating multiple copies of large objects might be aesthetically offensive, it rarely causes any measurable performance impact. A small number of objects might cause performance issues, but programmers are notoriously poor at guessing which code really causes problems. (For details, see Chapter 25.) Because it's a poor tradeoff to add complexity for dubious performance gains, a good approach to deep vs. shallow copies is to prefer deep copies until proven otherwise.

If you find that you do need to use a shallow-copy approach, Scott Meyers' *More Effective C++*, Item 29 (1996) contains an excellent discussion of the issues in C++. Martin Fowler's *Refactoring* (1999) describes the specific steps needed to convert from shallow copies to deep copies and from deep copies to shallow copies. (Fowler calls them reference objects and value objects.)

## 6.4 Reasons to Create a Class

If you believe everything you read, you might get the idea that the only reason to create a class is to model real-world objects. In practice, classes get created for many more reasons than that. Here's a list of good reasons to create a class.

### *Model real-world objects*

Modeling real-world objects might not be the only reason to create a class, but it's still a good reason! Create a class for each real-world object that your program models. Put the data needed for the object into the class, and then build service routines that model the behavior of the object. See the discussion of ADTs in Section 6.1 for examples.

### *Model abstract objects*

Another good reason to create a class is to model an *abstract object*—an object that isn't a concrete, real-world object, but that provides an abstraction of other concrete objects. A good example is the classic *Shape* object. *Circle* and *Square* really exist, but *Shape* is an abstraction of other specific shapes.

On programming projects, the abstractions are not ready made the way *Shape* is, so we have to work harder to come up with clean abstractions. The process of distilling abstract concepts from real-world entities is non-deterministic, and different designers will abstract out different generalities. If we didn't know about geometric shapes like circles, squares and triangles, for example, we might come up with more unusual shapes like squash shape, rutabaga shape, and Pontiac Aztek shape. Coming up with appropriate abstract objects is one of the major challenges in object-oriented design.

### *Reduce complexity*

The single most important reason to create a class is to reduce a program's complexity. Create a class to hide information so that you won't need to think about it. Sure, you'll need to think about it when you write the class. But after it's written, you should be able to forget the details and use the class without any knowledge of its internal workings. Other reasons to create classes—minimizing code size, improving maintainability, and improving correctness—are also good reasons, but without the abstractive power of classes, complex programs would be impossible to manage intellectually.

### *Isolate complexity*

Complexity in all forms—complicated algorithms, large data sets, intricate communications protocols, and so on—is prone to errors. If an error does occur, it will be easier to find if it isn't spread through the code but is localized within a class. Changes arising from fixing the error won't affect other code because only one class will have to be fixed—other code won't be touched. If you find a

**CROSS-REFERENCE** The reasons to create a class overlap with the reasons to create routines. For details, see Section 7.1, "Valid Reasons to Create a Routine."

**CROSS-REFERENCE** For more on identifying real-world objects, see "Find Real-World Objects" in Section 5.3.

### **KEY POINT**

better, simpler, or more reliable algorithm, it will be easier to replace the old algorithm if it has been isolated into a class. During development, it will be easier to try several designs and keep the one that works best.

### *Hide implementation details*

The desire to hide implementation details is a wonderful reason to create a class whether the details are as complicated as a convoluted database access or as mundane as whether a specific data member is stored as a number or a string.

### *Limit effects of changes*

Isolate areas that are likely to change so that the effects of changes are limited to the scope of a single class or, at most, a few classes. Design so that areas that are most likely to change are the easiest to change. Areas likely to change include hardware dependencies, input/output, complex data types, and business rules. The subsection titled “Hide Secrets (Information Hiding)” in Section 5.3 described several common sources of change. Several of the most common are summarized in this section.

### *Hide global data*

If you need to use global data, you can hide its implementation details behind a class interface. Working with global data through access routines provides several benefits compared to working with global data directly. You can change the structure of the data without changing your program. You can monitor accesses to the data. The discipline of using access routines also encourages you to think about whether the data is really global; it often becomes apparent that the “global data” is really just class data.

### *Streamline parameter passing*

If you’re passing a parameter among several routines, that might indicate a need to factor those routines into a class that share the parameter as class data. Streamlining parameter passing isn’t a goal, per se, but passing lots of data around suggests that a different class organization might work better.

### *Make central points of control*

It’s a good idea to control each task in one place. Control assumes many forms. Knowledge of the number of entries in a table is one form. Control of devices—files, database connections, printers, and so on—is another. Using one class to read from and write to a database is a form of centralized control. If the database needs to be converted to a flat file or to in-memory data, the changes will affect only the one class.

The idea of centralized control is similar to information hiding, but it has unique heuristic power that makes it worth adding to your programming toolbox.

**CROSS-REFERENCE** For a discussion of problems associated with using global data, see Section 13.3, “Global Data.”

**CROSS-REFERENCE** For details on information hiding, see “Hide Secrets (Information Hiding)” in Section 5.3.



***Facilitate reusable code***

Code put into well-factored classes can be reused in other programs more easily than the same code embedded in one larger class. Even if a section of code is called from only one place in the program and is understandable as part of a larger class, it makes sense to put it into its own class if that piece of code might be used in another program.

**HARD DATA**

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

**CROSS-REFERENCE** For more on implementing the minimum amount of functionality required, see "A program contains code that seems like it might be needed someday" in Section 24.3.

Notably, the core of NASA's approach to creating reusable classes does not involve "designing for reuse." NASA identifies reuse candidates at the ends of their projects. They then perform the work needed to make the classes reusable as a special project at the end of the main project or as the first step in a new project. This approach helps prevent "gold-plating"—creation of functionality that isn't required and that adds complexity unnecessarily.

***Plan for a family of programs***

If you expect a program to be modified, it's a good idea to isolate the parts that you expect to change by putting them into their own classes. You can then modify the classes without affecting the rest of the program, or you can put in completely new classes instead. Thinking through not just what one program will look like, but what the whole family of programs might look like is a powerful heuristic for anticipating entire categories of changes (Parnas 1976).

Several years ago I managed a team that wrote a series of programs used by our clients to sell insurance. We had to tailor each program to the specific client's insurance rates, quote-report format, and so on. But many parts of the programs were similar: the classes that input information about potential customers, that stored information in a customer database, that looked up rates, that computed total rates for a group, and so on. The team factored the program so that each part that varied from client to client was in its own class. The initial programming might have taken three months or so, but when we got a new client, we merely wrote a handful of new classes for the new client and dropped them into the rest of the code. A few days' work, and voila! Custom software!

### *Package related operations*

In cases in which you can't hide information, share data, or plan for flexibility, you can still package sets of operations into sensible groups such as trig functions, statistical functions, string-manipulation routines, bit-manipulation routines, graphics routines, and so on.

### *To accomplish a specific refactoring*

Many of the specific refactorings described in Chapter 24 result in new classes—including converting one class to two, hiding a delegate, removing a middle man, and introducing an extension class. These new classes could be motivated by a desire to better accomplish any of the objectives described throughout this section.

## Classes to Avoid

While classes in general are good, you can run into a few gotchas. Here are some classes to avoid.

### *Avoid creating god classes*

Avoid creating omniscient classes that are all-knowing and all-powerful. If a class spends its time retrieving data from other classes using *Get()* and *Set()* routines (that is, digging into their business and telling them what to do), ask whether that functionality might better be organized into those other classes rather than into the god class (Riel 1996).

### *Eliminate irrelevant classes*

If a class consists only of data but no behavior, ask yourself whether it's really a class and consider demoting it to become an attribute of another class.

### *Avoid classes named after verbs*

A class that has only behavior but no data is generally not really a class. Consider turning a class like *DatabaseInitialization()* or *StringBuilder()* into a routine on some other class.

## Summary of Reasons to Create a Class

Here's a summary list of the valid reasons to create a class:

- Model real-world objects
- Model abstract objects
- Reduce complexity
- Isolate complexity
- Hide implementation details

**CROSS-REFERENCE** This kind of class is usually called a structure. For more on structures, see Section 13.1, "Structures."

- 1222 • Limit effects of changes
- 1223 • Hide global data
- 1224 • Streamline parameter passing
- 1225 • Make central points of control
- 1226 • Facilitate reusable code
- 1227 • Plan for a family of programs
- 1228 • Package related operations
- 1229 • To accomplish a specific refactoring

## 1230 6.5 Language-Specific Issues

1231 Approaches to classes in different programming languages vary in interesting  
1232 ways. Consider how you override a member routine to achieve polymorphism in  
1233 a derived class. In Java, all routines are overridable by default, and a routine  
1234 must be declared *final* to prevent a derived class from overriding it. In C++,  
1235 routines are not overridable by default. A routine must be declared *virtual* in the  
1236 base class to be overridable. In Visual Basic, a routine must be declared  
1237 *overridable* in the base class, and the derived class should use the *overrides*  
1238 keyword.

1239 Here are some of the class-related areas that vary significantly depending on the  
1240 language:

- 1241 • Behavior of overridden constructors and destructors in an inheritance tree
- 1242 • Behavior of constructors and destructors under exception-handling  
1243 conditions
- 1244 • Importance of default constructors (constructors with no arguments)
- 1245 • Time at which a destructor or finalizer is called
- 1246 • Wisdom of overriding the language's built-in operators, including  
1247 assignment and equality
- 1248 • How memory is handled as objects are created and destroyed, or as they are  
1249 declared and go out of scope

1250 Detailed discussions of these issues are beyond the scope of this book, but the  
1251 "Additional Resources" section at the end of this chapter points to good  
1252 language-specific resources.

## 6.6 Beyond Classes: Packages

**CROSS-REFERENCE** For more on the distinction between classes and packages, see “Levels of Design” in Section 5.2.

Classes are currently the best way for programmers to achieve modularity. But modularity is a big topic, and it extends beyond classes. Over the past several decades, software development has advanced in large part by increasing the granularity of the aggregations that we have to work with. The first aggregation we had was the statement, which at the time seemed like a big step up from machine instructions. Then came subroutines, and later came classes.

It’s evident that we could better support the goals of abstraction and encapsulation if we had good tools for aggregating groups of objects. Ada supported the notion of packages more than a decade ago, and Java supports packages today.

C++’s and C#’s namespaces are a good step in the right direction, though creating packages with them is a little bit like writing web pages directly in html.

If you’re programming in a language that doesn’t support packages directly, you can create your own poor-programmer’s version of a package and enforce it through programming standards that include

- naming conventions that differentiate which classes are public and which are for the package’s private use
- naming conventions, code-organization conventions (project structure), or both that identify which package each class belongs to
- Rules that define which packages are allowed to use which other packages, including whether the usage can be inheritance, containment, or both

These workarounds are good examples of the distinction between programming *in* a language vs. programming *into* a language. For more on this distinction, see Section 34.4, “Program Into Your Language, Not In It.”

CC2E.COM/0672

**CROSS-REFERENCE** This is a checklist of considerations about the quality of the class. For a list of the steps used to build a class, see the checklist “The Pseudocode Programming Process” in **Chapter 9**, page 000.

---

### CHECKLIST: Class Quality

---

#### Abstract Data Types

- ☐ Have you thought of the classes in your program as Abstract Data Types and evaluated their interfaces from that point of view?

**Abstraction**

- ☐ Does the class have a central purpose?
- ☐ Is the class well named, and does its name describe its central purpose?
- ☐ Does the class's interface present a consistent abstraction?
- ☐ Does the class's interface make obvious how you should use the class?
- ☐ Is the class's interface abstract enough that you don't have to think about how its services are implemented? Can you treat the class as a black box?
- ☐ Are the class's services complete enough that other classes don't have to meddle with its internal data?
- ☐ Has unrelated information been moved out of the class?
- ☐ Have you thought about subdividing the class into component classes, and have you subdivided it as much as you can?
- ☐ Are you preserving the integrity of the class's interface as you modify the class?

**Encapsulation**

- ☐ Does the class minimize accessibility to its members?
- ☐ Does the class avoid exposing member data?
- ☐ Does the class hide its implementation details from other classes as much as the programming language permits?
- ☐ Does the class avoid making assumptions about its users, including its derived classes?
- ☐ Is the class independent of other classes? Is it loosely coupled?

**Inheritance**

- ☐ Is inheritance used only to model "is a" relationships?
- ☐ Does the class documentation describe the inheritance strategy?
- ☐ Do derived classes adhere to the Liskov Substitution Principle?
- ☐ Do derived classes avoid "overriding" non overridable routines?
- ☐ Are common interfaces, data, and behavior as high as possible in the inheritance tree?
- ☐ Are inheritance trees fairly shallow?
- ☐ Are all data members in the base class private rather than protected?

**Other Implementation Issues**

- ☐ Does the class contain about seven data members or fewer?
- ☐ Does the class minimize direct and indirect routine calls to other classes?

- 1319 ☐ Does the class collaborate with other classes only to the extent absolutely  
1320 necessary?
- 1321 ☐ Is all member data initialized in the constructor?
- 1322 ☐ Is the class designed to be used as deep copies rather than shallow copies  
1323 unless there's a measured reason to create shallow copies?

1324 **Language-Specific Issues**

- 1325 ☐ Have you investigated the language-specific issues for classes in your  
1326 specific programming language?
- 

CC2E.COM/0679

1328 **Additional Resources**

1329 **Classes in General**

1330 Meyer, Bertrand. *Object-Oriented Software Construction, 2d Ed.* New York:  
1331 Prentice Hall PTR, 1997. This book contains an in-depth discussion of Abstract  
1332 Data Types and explains how they form the basis for classes. Chapters 14-16  
1333 discuss inheritance in depth. Meyer provides a strong argument in favor of  
1334 multiple inheritance in Chapter 15.

1335 Riel, Arthur J. *Object-Oriented Design Heuristics*, Reading, Mass.: Addison  
1336 Wesley, 1996. This book contains numerous suggestions for improving program  
1337 design, mostly at the class level. I avoided the book for several years because it  
1338 appeared to be too big (talk about people in glass houses!). However, the body of  
1339 the book is only about 200 pages long. Riel's writing is accessible and enjoyable.  
1340 The content is focused and practical.

1341 **C++**

1342 CC2E.COM/0686 Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and  
1343 Designs, 2d Ed.*, Reading, Mass.: Addison Wesley, 1998.

1344 Meyers, Scott, 1996, *More Effective C++: 35 New Ways to Improve Your  
1345 Programs and Designs*, Reading, Mass.: Addison Wesley, 1996. Both of  
1346 Meyers' books are canonical references for C++ programmers. The books are  
1347 entertaining and help to instill a language-lawyer's appreciation for the nuances  
1348 of C++.

1349 **Java**

1350 CC2E.COM/0693 Bloch, Joshua. *Effective Java Programming Language Guide*, Boston, Mass.:  
1351 Addison Wesley, 2001. Bloch's book provides much good Java-specific advice  
1352 as well as introducing more general, good object-oriented practices.

## Visual Basic

The following books are good references on classes in Visual Basic:

Foxall, James. *Practical Standards for Microsoft Visual Basic .NET*, Redmond, WA: Microsoft Press, 2003.

Cornell, Gary and Jonathan Morrison. *Programming VB .NET: A Guide for Experienced Programmers*, Berkeley, Calif.: Apress, 2002.

Barwell, Fred, et al. *Professional VB.NET, 2d Ed.*, Wrox, 2002.

## Key Points

- Class interfaces should provide a consistent abstraction. Many problems arise from violating this single principle.
- A class interface should hide something—a system interface, a design decision, or an implementation detail.
- Containment is usually preferable to inheritance unless you're modeling an "is a" relationship.
- Inheritance is a useful tool, but it adds complexity, which is counter to the Primary Technical Imperative of minimizing complexity.
- Classes are your primary tool for managing complexity. Give their design as much attention as needed to accomplish that objective.