5

Elegance and Classes

SECTION 5.0 INTRODUCTION

In previous chapters, we have discussed some foundational topics necessary for developing a good object-oriented design of a software system. In particular, we discussed the features of object-oriented languages and some of the advantages over other types of languages that they provide to a designer and programmer. We also discussed low-level implementation issues regarding these features.

In this chapter, we will discuss how and when to use these features and we will introduce other guidelines to help design and build elegant systems "from scratch." We will compare various possible competing designs of systems in terms of the classes involved, their responsibilities, their relationships with their collaborators, and their implementations.

SECTION 5.1 STARTING OUT FINDING CLASSES AND THEIR RELATIONSHIPS

Gang of Four

In [1], the authors state, "The hard part about object-oriented design is decomposing a system into objects." In this section, we will discuss some techniques to help you get started with such a decomposition.

Let us suppose that we are asked by a customer to create a new application. As we briefly discussed in Chapter 1, we should not start by writing code. Instead, we need start at a higher level. Here is a simplified outline of the steps that should be taken:

1. We need to develop a precise specification of the software system.
2. We need to determine what classes to create and what will be the responsibilities or behaviors of objects of those classes and of the other objects with which they collaborate.
3. We need to specify these behaviors more precisely by designing the *protocol,* or public interfaces, of the classes in the design.
4. Finally, we can code the classes and the methods that implement their protocols.

We will discuss these concepts and some of the issues they raise by considering a very simple project. Let us suppose that our goal is to develop a program for a customer that allows us to gather statistics on the frequency of occurrence of words

in text files. More precisely, let us suppose that the project was initially presented to us by the customer in the form of the following statements:

```
When the Java program starts, it analyzes the specified
text file. It constructs a summary report regarding each
word that occurs in the file and the number of times that
word occurs, sorted from most frequently occurring word to
least frequently occurring.
```

Students with one or two semesters of Java programmming should quite easily be able develop a Java program with the specified behavior, but there is a good possibility that the program could be rejected by the customer as inadequate. Do you see the problem? The description of the desired program is far too vague. Before moving to steps 2–4 of the outline above, the customer needs to be queried for further clarification to avoid wasting time writing code that is not what the customer really wants. In other words, before the design can begin, a precise specification needs to be created.

Let us see what is wrong with the specification for our word frequency program. You should be able to easily come up with a half dozen unanswered questions about the program. Here are some such questions:

1. What is the user interface for this program? In particular, how does the program know what text file to analyze? For example, does the program display a dialog box, does it ask the user to type a file name on a command line, or does it expect the user to supply the name as a command-line argument when starting up the program?
2. What should happen if the user specifies the name of a file that is nonexistent?
3. What should the format of the summary be? Can it be specified by the user?
4. When counting occurrences of words, does capitalization matter? Can the user specify how to treat capitalization?
5. Can the same execution run of the program be used to analyze several text files, or do you need to rerun the program whenever you want to analyze another file?
6. What is meant by a "word"? What distinguishes a word from an arbitrary sequence of characters? Are we counting only words that appear in a dictionary? If so, which dictionary?
7. When counting occurrences of words, do we distinguish between singular and plural forms of the same word, such as "shell" versus "shells"? How about "mouse" versus "mice"?
8. When reporting the results, what order should be used for two words that have the same frequency of occurrence?
9. Is the efficiency of the program regarding space or time an important concern?
10. Is the program going to be enhanced later to deal with input other than text files and with other forms of summary reports?

One of the techniques that are useful for clarifying the vague parts of a specification is to create, with the help of the customer, "use cases" or "user stories." A *use case* is a sequence of steps indicating how the program is to behave in a

certain situation to achieve a particular goal. To develop a complete specification, there should be a use case for each of the various ways the system is expected to behave for each goal. Because our project is so simple, there won't be many use cases. In fact, for our project, only one use case is needed.

Let us suppose that the customer decided that the use case should have the following steps:

1. The user starts up the program by typing the following command on the command line of a console window: "java," followed by the name of the program, followed by the full path name of the text file. For example, the command might be

   ```
   java WordCounter /users/smith/Hamlet.txt
   ```

2. The program checks that the file exists. If not, it reports an error message and exits.
3. The program traverses the file, treating each byte as a character and treating each sequence of characters not containing any delimiters as a word. The delimiter characters are Java's default whitespace characters.
4. The program keeps track of the number of occurrences of each word. Two words are considered equal if and only if they contain exactly the same sequence of characters.
5. The program prints to the console window a list of all the words and their frequencies, with words with the highest frequency first and with one word and frequency (separated by a tab) per line.
6. The program quits.

Notice that the use case actually has two variations, one of them successful and one unsuccessful. These variations define different *scenarios* of the use case. The successful scenario is the one where the word frequencies get computed and displayed. The unsuccessful scenario is where the program detects that there is no file with the given name and reports an error message.

The different use cases for a software system can be displayed in a UML use case diagram. Our application is so simple that such a diagram is not all that helpful, but for more complex software systems, it can provide a very helpful overview of the goals and uses of the system. See Figure 5.1 for a use case diagram for our word frequency counter application. In such a diagram, each circle corresponds to another use case. The stick figure corresponds to an *actor*, which is a role that a user

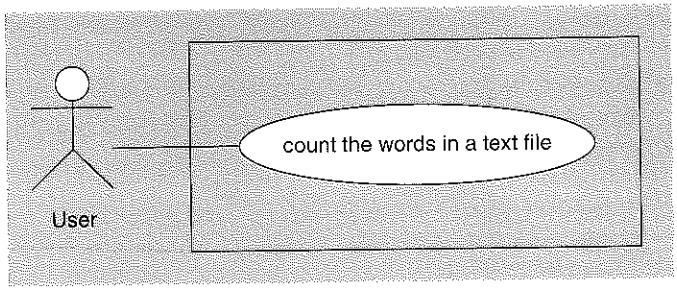

FIGURE 5.1 A use case diagram for our word frequency counter application.

of the system can play. In our case, there is only one such role, namely the role of the user typing in the command on the command line to run the program. See Appendix A for more information regarding UML use case diagrams.

Note that the steps in the use case provide answers to the first seven of the unanswered questions above, but it doesn't give us all the information we need. To finish our specification so that we can proceed to the next steps in the software design process, let us answer the remaining questions somewhat arbitrarily:

- If two words have the same frequency, they should be reported in alphabetical order using the ASCII ordering.
- The efficiency of the program is important enough that only one read-through of the text file is permitted by the program. Furthermore, the program is not allowed to copy the whole text file into memory, since it might use up too much memory.
- The program is going to be enhanced later to deal with other inputs and outputs of an as-yet-unknown form.

Now that we have a more precise specification of the project, we can begin to think about the design. In particular, we need to determine what objects and classes will be used in our design.

There are many possible candidates for classes, there are many ways to divide the work among them, and there are many ways to have the classes interact. For example, the number of classes in a design could be one or it could be in the hundreds. The responsibilities could all be in one class with the other classes being just behaviorless data holders, or a design could divide the responsibilities evenly among the participating classes of objects. The collaborations between the objects could be extensive, with every object communicating directly with every other object, or the collaborations could be minimal, with each object communicating with as few other objects as possible. With virtually unlimited choices, how should you proceed?

Before reading any further, it is valuable to attempt to come up with your own design to compare to the forthcoming design. You may even come up with a better design than we do.

Extract Nouns and Verbs

An initial but overly simplistic way to obtain candidate classes for a design is to study the project specification and extract all the nouns. Each noun can correspond to a potential object or class. Similarly, by extracting the verbs, we can get potential actions or behaviors of the objects.

Here are some of the nouns used in our specification:

program, user, file, word, number, times, report, pathname, capitalization, characters, line, window, name, character

Some of those nouns seem to correspond to useful classes, such as a Program class that could be in charge of the whole process (although a better name could also be chosen for such a class) and File and Report classes. But it is not clear whether the rest of the nouns would correspond to meaningful classes in our design. For

example, do we need a User class? Do we need a Word class, or will the String class suffice for representing words?

Let's also look at some of the verbs in the specification:

starts, analyzes, constructs, returns, occurred

Again, it is clear that some of those words do correspond to useful actions that we will want our objects to perform, such as "construct" and "analyze," but most of the verbs are not at all helpful in determining what behaviors our objects should have.

Use Concepts from the Application Domain

One problem with the noun and verb technique is that it relies too much on the particular words used in the specification. In addition to the candidates that that technique provides, it is valuable to look at the problem domain for additional candidates. For example, if you are designing a Monopoly game, the problem domain is board games. Natural candidates for classes in this domain include Player, Piece, and Board, and candidates for actions include moving a piece. If you are designing an application for drawing pictures, natural candidates for classes include Picture, Tool, Paint, and Color, and candidates for actions include drawing and erasing.

Use CRC Cards

One excellent approach for deciding on the classes, their behaviors, and their responsibilities is to use CRC cards. A *CRC card* is just a note card (4″ × 6″ or maybe 5″ × 7″) that corresponds to one class in the design. CRC stands for "class, responsibilities, and collaborators" in that the card is divided into three sections, one for the class name, one for the class' responsibilities, and one for the class' collaborators. One of the advantages of using actual paper cards is that their small size forces you to summarize the responsibilities of each class in a few words and so helps the design team avoid getting bogged down in low-level details too soon. Another advantage of using paper cards (in contrast to cards on a computer screen) is that they can be easily moved around, regrouped, and carried by members of the design team during role playing or during design activities.

A good way to use such cards is through role playing. When doing role playing, blank cards are handed out to members of the design team, one card per person. Then the people act out a use case for the project. As the team members play their roles, they may realize that a new class may need to be created, in which case one of the members with a blank card writes the name of that new class on his/her card. From that point on, that person plays the role of an object of that class. Each time some new action needs to be performed, the team decides which class has the responsibility for performing that task, and the team member holding the card for that class writes the responsibility down on the card. The team member also writes down the other classes that the class needs to collaborate with in order to accomplish the job. Such role playing should be performed for the main use cases of the application.

Role playing can also be augmented in many ways. For example, it is sometimes helpful to use an actual physical object to pass around to indicate whose method is currently being executed in the thread under discussion.

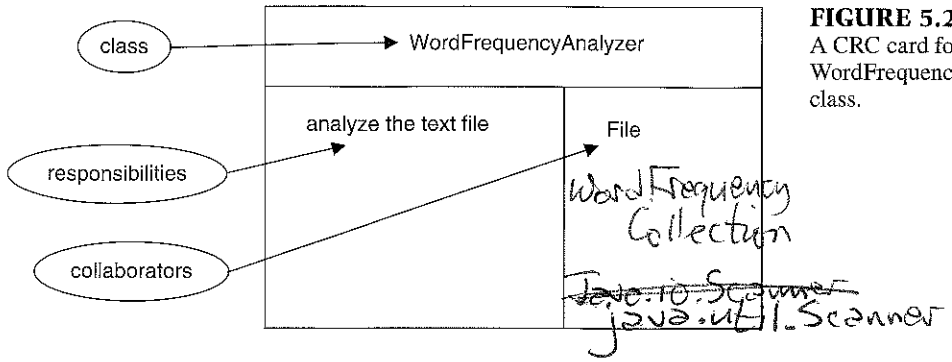

FIGURE 5.2 A CRC card for the WordFrequencyAnalyzer class.

Let's use CRC cards to help us with the design of the word frequency counter project. Along the way, you'll notice that we go back and forth between the cards as we refine our design.

The core of the program is the analysis of the text file. A class has to be responsible for that analysis, and so let us start by creating a "WordFrequencyAnalyzer" class and give it that responsibility. See Figure 5.2 for our first CRC card. Let us suppose that a WordFrequencyAnalyzer object will be given a File from which it will get the words to analyze, and so the WordFrequencyAnalyzer object will collaborate with a File object. We have added its basic responsibility and its collaborator to the card in Figure 5.2.

Do we now need to create a CRC card for the File class? Technically, we should do so and we should list the responsibilities and collaborators of that class. However, to simplify our presentation and because the standard File class in the java.io library meets our needs, we won't create a CRC card for it at this time.

Once the WordFrequencyAnalyzer has finished its analysis, the results need to be printed. The WordFrequencyAnalyzer could be responsible for this action, but it seems more elegant to give a different object that responsibility. This decision then leads to the question as to how the WordFrequencyAnalyzer is going to store and make available the results of its analysis. Because storing or maintaining information is a responsibility that seems just as important as manipulating information, let us create a new WordFrequencyCollection class that stores the results of the analysis. We now have a second CRC card to create. See Figure 5.3. Once we add

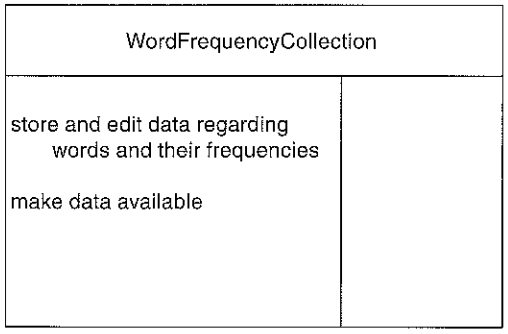

FIGURE 5.3 The WordFrequencyCollection CRC card.

this new class, we need to go back to the CRC card for WordFrequencyAnalyzer and add new responsibilities and a new collaborator. The new collaborator is the WordFrequencyCollection class, and the new responsibilities include creating, adding data to, and then making available the WordFrequencyCollection that contains the results of the analysis of the text file.

We will make the WordFrequencyCollection class responsible for making all its data available for clients, but we will leave formatting and print that data to other classes.

We now have a rough, very high-level design of our system. The program will create a WordFrequencyAnalyzer object, pass it the file to analyze, tell it to analyze the file, and finally tell it to return the resulting WordFrequencyCollection object. Then the program will get the data from this object and print it to the console.

Let us now refine our design a little. First of all, where does the WordFrequency-Analyzer get the File to analyze? Our top level design says it comes from the program. Therefore, let us add a new CRC card for a class containing the main program. Let us call that class the WordCounter class. See Figure 5.4 for its CRC card. It has responsibilities for checking for a nonexistent file, creating the File and WordFrequencyAnalyzer, and asking the analyzer to analyze the file. It is also responsible for getting and printing the result. That result is stored in a WordFrequencyCollection object.

Next, let us refine the method by which the WordFrequencyAnalyzer analyzes the text file. This object needs first to create an empty WordFrequencyCollection object. It then needs to repeatedly get the next word from the file and process it. But how can it do that job? The File object corresponding to the text file has no methods for extracting words from the file. A FileReader object from the java.io library might be useful here, except that it reads only a character at a time and we want it to read a word at a time. It turns out that the java.util.Scanner class has just the abilities we need. Therefore, we will add it as a collaborator in the WordFrequencyAnalyzer's CRC card. The final copies of our three CRC cards are as they appear in Figures 5.2–5.4 except that the WordFrequencyAnalyzer's card has two more collaborators: WordFrequencyCollection and Scanner.

We now have a sufficiently refined design to be able to proceed to the next step. However, we would first like to point out that there are many possible variations on

FIGURE 5.4 The WordCounter CRC card.

WordCounter	
report an error if there is no file with a given name create a File and a WordFrequencyAnalyzer initiate the analysis get and print the result	File WordFrequency-Analyzer WordFrequency-Collection

this design. For example, the WordCounter could create the Scanner and pass that object to the WordFrequencyAnalyzer instead of a File object. Which is better? This question is an exercise at the end of the chapter.

We would also like to point out that in a larger project, further low-level details of the design and issues related to the other use cases of the system would need to be considered in order to develop a more complete design before moving on to the next step of the process. Since our design is so simple and since we are just introducing these topics rather than attempting to cover them exhaustively, we will move on now.

Class Protocols

Now that we have potential classes and their responsibilities and collaborators, we will proceed to construct the protocols, or public interfaces, of all the classes in our design. As a first step, we will convert each CRC card to a class and each responsibility to a method. However, keep in mind that, in general, not all responsibilities of a class will ultimately correspond to methods of that class. Similarly, not every CRC card will ultimately end up being implemented as a class nor will all classes in the final implementation necessarily come from the initial set of CRC cards. For example, we never created a card for the String class, although we will certainly use that class. Also, a card could correspond to a Java interface, and we might need to create one or more concrete classes that implement that interface.

Here is our first attempt at the classes and protocols, obtained by turning each responsibility into a method and adding constructors:

```
public class WordCounter
{
   public WordCounter(); //constructor
   public void checkFileExistence(String filename);
   public File createFile();
   public WordFrequencyAnalyzer createAnalyzer();
   public void initiateAnalysis();
   public getAndPrintResult();
}
public class WordFrequencyAnalyzer
{
   public WordFrequencyAnalyzer(); //constructor
   public void analyzeText(File file);
   public WordFrequencyCollection getResults();
}
public class WordFrequencyCollection
{
   public WordFrequencyCollection(); //constructor
   public void editCollection();
   public String toString();
}
```

Note that, in the process of constructing these protocols for the classes on our CRC cards, we kept in mind the issues discussed in the last chapter regarding methods and

method signatures. That is, we attempted to use intention-revealing method names, and we used verb phrases for names of modifier methods.

At this point, it is valuable to step back and decide whether these protocols are appropriate. The methods in the second class seems appropriate, but not the methods in the WordCounter class or the WordFrequencyCollection class. Let us inspect these classes in more detail.

Almost all the responsibilities of the WordCounter class are just steps of the main method that the user would call to do the analysis. They could be implemented as private methods in the WordCounter class, but they do not belong in the public interface. Therefore, a more appropriate protocol for the WordCounter class is the following:

```
public class WordCounter
{
  public static void main(String[] args);
}
```

Regarding the WordFrequencyCollection class, what does the `editCollection` method do? It is quite meaningless. The responsibility of "managing the collection" is too vague and should be refined down to several methods. The only one we really need for our application is a method for adding new words to the collection. So we will add that method to our protocol:

```
public class WordFrequencyCollection
{
  public WordFrequencyCollection(); //constructor
  public void add(String word);
  public String toString();
}
```

The WordFrequencyCollection class is also responsible for making the results available. The protocol above does so through the `toString` method. However, such an approach is not very helpful to clients. According to Java conventions, the `toString` method is supposed to create a humanly readable form of the state of the object, not to make available data from the class to users of the class. Consider, for example, a user who only wants to know how many different words appear in the file, regardless of their frequency. Under the current design, that user would need to search and parse the output of the `toString` method to find that information. How can we better make the data in the collection available?

There are a variety of approaches that could be taken. For example, the collection could have an `iterator` method that returns an Iterator over the words in the collection. Also, there could be a method that, for each word, returns the frequency of occurrence of that word. Following this approach, we get a much better protocol for the WordFrequencyCollection class:

```
public class WordFrequencyCollection implements Iterable<String>
{
  public WordFrequencyCollection(); //constructor
  public void add(String word);
```

```
  public Iterator<String> iterator();
  public int getFrequency(String word);
}
```

With these protocols in place, we have reached the point where we can begin implementing these classes. For example, here is one possible implementation of the `main` method of the WordCounter class:

```
public class WordCounter
{
  public static void main(String[] args)
  {
    //create the file
    File file = new File(args[0]);
    //test to see whether the file exists and can be read
    try {
      new FileReader(file);
    } catch (FileNotFoundException e) {
      System.out.println("error: file cannot be read");
      System.exit(0);
    }
    //analyze the file
    WordFrequencyAnalyzer analyzer =
                                    new WordFrequencyAnalyzer();
    analyzer.analyzeText(file);
    //get and display the results
    WordFrequencyCollection collection = analyzer.getResults();
    for(String word : collection) {
      System.out.println(word + ":" + collection.getFrequency(word) );
    }
  }
}
```

There is still some low-level design left to work out before implementing the remaining classes, such as determining how the WordFrequencyCollection stores its data. We leave the remaining design and implementation as an exercise.

The class diagram containing these three classes is shown in Figure 5.5.

The Big Picture

Let us review the process we followed with our program. The first step was to get a precise specification of the problem. Use cases were helpful in this step. The next step was to determine what classes and their responsibilities and collaborators are needed to accomplish our task. CRC cards were helpful in this step. The third step consisted of determining the precise protocol or public interface of the classes in our design. The final step was implementing the classes.

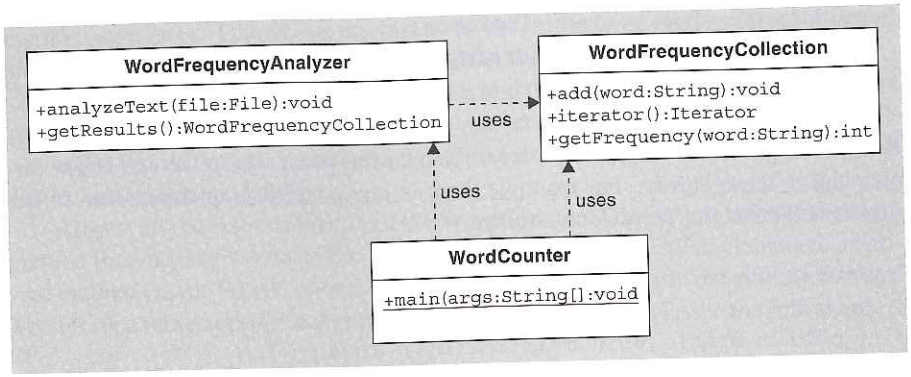

FIGURE 5.5 The word frequency analyzer classes.

The word frequency counter example should give you the basic idea of how to execute each of these steps of the design process and the tools that you might want to use along the way. You may choose to use a different approach for designing and implementing a program, but the general lessons learned here and the tools introduced, such as CRC cards and use cases, should be valuable regardless of your approach.

One of the lessons to be learned from this discussion can be summarized very generally:

Guideline

When working in the early high-level design phase, keep the discussion at a high level. In particular, avoid wasting time on implementation details and low-level data representations at the initial stages.

How well have we stayed at a high level in the initial stages of the design of the word frequency counter program? We could have done better at avoiding references to low-level value classes. For example, we could have referred to the words of the text file as objects of a "Word" class instead of as Strings. One advantage of storing our words in such an object instead of in a String is that other behavior can be added to it and other information than the sequence of characters can be stored in it, such as the line and column number of the start of the word in the original text file. This extra behavior and information might make the Word class more generally useful. For example, knowing the line and column number of a word would be useful in an enhanced version of our program that highlights the occurrence of words in a window displaying the contents of the text file.

Similarly, we could have represented the text file by a TextFile object instead of automatically using the java.io.File class, and we could have created a FileScanner class for extracting the words from the file instead of automatically using the java .util.Scanner class.

Another way of stating the guideline is, in the initial design phase, you should avoid trying to map your classes to existing Java library classes and instead design the "ideal" classes for your program. Then, as you refine the design, you can use the Java library classes as necessary to implement your ideal classes.

SECTION 5.2 MAXIMIZING COHESION

As you have hopefully seen in the preceding discussion, an extremely important part of the design process is determining which class should be responsible for which behavior. When making such determinations, it is useful to recall one of the guidelines from Chapter 4. In that chapter, we said that methods should do one thing only and do it well. As you shall see, one of the fundamental design principles presented in this chapter is that *classes* should behave the same way.

Guideline

Every class should be responsible for doing one thing only and doing it well.

This rule does not mean that each class should have exactly one method. Instead, it says that a class should model one concept, and all the methods in the class should be related to and appropriate for that concept. That is, all the responsibilities of the class should focus on the concept being modeled by the class. One advantage of following this guideline is that a class' behavior and its role in a software system is much clearer to everyone. Furthermore, the class is more reusable if it isn't encumbered with responsibilities and data irrelevant to its main purpose for being.

For example, a class that does one thing quite well is the String class. All the methods and data of that class refer to one and only one concept, namely a sequence of characters. The methods of that class provide useful tools for manipulating that sequence.

A simple example of an ill-formed class would be one that is responsible for storing all the information relating to a person (e.g., name, address, age) and all the information relating to that person's current car (e.g., the brand, model, color, age). Instead, the person data and car data should be stored in separate objects, which, if necessary, have references to each other. These two classes are much more reusable than the original class.

Another ill-formed class is a "god" class that controls all other objects in a large program (see Figure 5.6). Such a class acts like a master class with all the responsibilities in the program, and all other classes are treated like slave classes or just data holders. As discussed in Chapter 2, this approach is counter to the OO paradigm. In OO programming, there should be decentralized control, with the responsibilities for various actions spread out among the cooperating classes.

Another term that can be used for describing the focused nature of a class is "cohesion." When we say we want each class to "do one thing only," we mean that

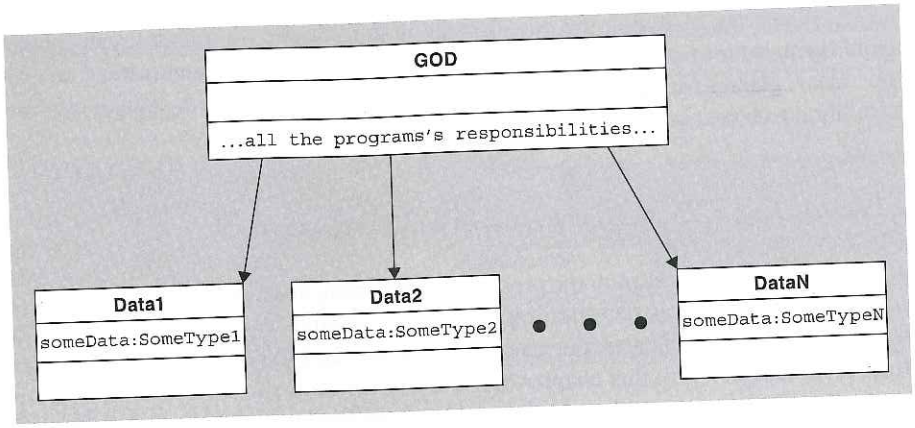

FIGURE 5.6 A god class and its data classes.

we want each class to have high cohesion in that all the class' behaviors and responsibilities are tightly related. The design principle of maximizing cohesion is one of the patterns of GRASP (general responsibility assignment software patterns) documented in [2].

One simple measure of the cohesiveness of a class is how easy it is to describe the purpose of the class in a simple sentence without using the word "and." If you cannot do so, then the class is probably less cohesive than it should be.

SECTION 5.3 SEPARATION OF RESPONSIBILITY

Another way of saying that a class should do one thing only is that different kinds of responsibilities should be delegated to different classes. Unfortunately, it is not always easy to decide how the responsibilities should be spread out among classes that do one thing only and do it well. Consider the following examples.

1. When an array of objects needs to be sorted, the objects need to be compared to each other. Should the objects know how to compare themselves to other objects with a method similar to the `compareTo` method in the Comparable interface, or should a separate object, such as the Comparator discussed in Section 3.10, be responsible for doing the comparing? There is no one right answer to this question, which is why the Java API includes both options. For example, the Arrays class in the java.util package includes two methods that sort arrays of objects. One method uses the `compareTo` method of each object, and the other uses a Comparator to do the comparing.
2. A *graph* is a set of nodes each of which is connected by edges to an arbitrary number of other nodes. See Figure 5.7 for a picture of a graph. A graph could be modeled by a Graph class that has a collection of Node objects and each Node object has a collection of references to the other Nodes to which it is connected. Users of graphs often visit the nodes by traversing the edges—for example, to

find a path from one node to another. In such cases, the users need to keep track of which nodes in the graph have already been visited so that they don't visit them again. To keep track of the already-visited nodes, there are several approaches that could be taken:

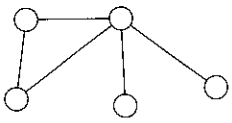

FIGURE 5.7
A graph with five nodes and five edges.

a. There could be a public `setVisited` method in each Node that the user can call to set the Node as visited and a `getVisited` method that the user can call to see whether the Node has been visited.

b. The Graph object could have a public `setVisited(Node n)` method that sets the Node as visited and a `getVisited(Node n)` method that returns true or false indicating whether the Node has been visited.

c. An object other than the Nodes and Graph objects could keep track of the visited Nodes.

One major problem with the first two approaches is that they do not allow two concurrent traversals of a graph. That is, if one traverser visits certain nodes first, then the other traverser may likely miss those nodes since they have already been marked as visited. Even if traversals are done sequentially, things can go awry if the first traverser, when finished, forgets to reset all the Nodes to unvisited before the second traverser begins. The third option is best. The object keeping track of the visited nodes could be the traverser or some other object employed by the traverser.

3. Consider the drawing program mentioned in Section 3.9. Suppose that the user of the program can select (i.e., highlight) any number of the shapes by clicking on them. Who should be responsible for keeping track of which shapes are currently selected? Again, there are several options, including:

a. The shapes themselves keep track of whether they are selected.

b. The view displaying the shapes keep track of the selected shapes.

c. Some other object keep track of the selected shapes.

The first approach seems most appropriate from an OO perspective; after all, shouldn't the shapes be responsible for their own behavior and state? However, consider what happens when the program needs to delete the currently selected shapes. In that case, the only way to find the selected ones is for some object (e.g., the view) to traverse the collection of all the shapes and ask them individually whether they are selected, and, if so, delete them. This can be inefficient if there are a large number of shapes and only a few of them have been selected. In the case of the second and third options, should the shapes themselves also know whether they are selected? This problem has no obvious best solution—the programmer should choose the solution that is most appropriate based on the context.

From these examples, you may have realized the point we are driving at:

Guideline

Different kinds of responsibilities should be separated among different objects.

How can you tell whether you have given a class too many responsibilities? There is no easy answer, although it is helpful to think of the three categories of responsibilities [3] that a class can have: knowing, doing, and controlling or deciding. A class that is only a data holder can be thought of as having "knowing" responsibilities; it is a class that knows some information and provides some kind of access to it. A class that manipulates data that it knows or that is passed to it or gathered from other objects has "doing" responsibilities. A class that controls the action of various other objects, deciding which object is to do what and when, has "controlling" responsibilities. Very broadly, one should consider separating knowing, doing, and controlling behaviors into different classes.

There is another general rule that can help you decide which class gets which responsibilities. In the third example at the beginning of this section, we suggested that, if only the shapes know they are selected and if we want to delete the selected shapes, then some object needs to find the selected shapes and delete them. What object should do that job of finding and deleting? The natural choice is the object that maintains the collection of shapes. That object has the necessary data to perform the desired tasks, and so it should be the one to do so. This principle is called the *Expert* pattern, another instance of the use of GRASP [2].

Guideline (Expert pattern)

The object that contains the necessary data to perform a task should be the object that performs the task.

The String class follows this pattern. A String object contains a fixed finite sequence of characters and can do most of the tasks you might want done with that sequence. For example, it can give you the substring starting at a given index, and it can tell you where a given character occurs in the sequence and whether its sequence comes before or after another String's sequence alphabetically.

For another example, consider a CarDealer object that stores, among other things, a collection of Car objects corresponding to the cars currently in stock. Suppose you want to know if there are any blue minivans in stock. You (the user) can do the work yourself by asking the CarDealer for a list of all cars and then traversing the list looking for a blue minivan. Or you can ask the CarDealer object if it has any blue minivans in stock. The latter option is clearly preferable. The CarDealer is the object with the necessary data to perform the task, and so it should be the one that does so.

In summary, the Expert pattern is being violated whenever a class A has a method that gets several pieces of data from an object of class B, manipulates those pieces, and then possibly returns a result. The object of class B has all the necessary data, and so that object should be performing the manipulation rather than the A

object. That is, the method in class A should probably be a method in class B. The Move Method refactoring [4] would be very useful here.

Others have rephrased the Expert pattern in alternate forms. For example, Owen Astrachan presents it as a fundamental law of OO design [5]: "Ask not what you can do to an object, ask what an object can do to itself." The Expert pattern can also be rephrased in the form of a question, "What is it I'm doing with this data, and why doesn't the class do it for me?"[6].

One hint that your objects may not be doing their jobs properly is the existence of a lot of calls to accessor methods (`getX` methods). If there are many such calls, then you should ask yourself what the other objects are doing with the data they get when they call one of those methods? Shouldn't your object with the `getX` methods be doing that job for them?

Notice how this issue relates to our earlier concept of a well-designed class doing one thing only and doing it well. An object that does its job well not only retrieves data for you, but also it manipulates that data for you, if appropriate, instead of expecting you to do all the manipulations yourself.

The word "appropriate" was used in the preceding paragraph because there is usually a tension between the principle of doing one thing only and doing it well and the Expert pattern. For example, consider an application that is the front end of a database. The application gets some data from the database and then displays it in a window. The Expert pattern suggests that the database be responsible for displaying its own data. However, a database object should also have high cohesion, which suggests it should not be responsible for two completely different activities, namely, managing the data stored in it and displaying that data in a GUI.

Now that we have some design guidelines under our belts, let us go back to the design of our word frequency counter application in Section 5.1 and consider how well that design follows these guidelines. Do the classes in that design do one thing only and do it well? Do they follow the Expert pattern? Do they separate responsibilities appropriately? We cannot answer these questions fully without looking at the complete implementation of the classes, but we can make some remarks based on the class' protocols.

The WordFrequencyCollection class is very cohesive. It is responsible for maintaining a collection of data, and it does nothing else. The `iterator` and `getFrequency` methods are appropriate ways for it to return the information it has gathered. If it were to print the data, as well, then it would be doing something it shouldn't. Instead it appropriately lets others deal with the printing. Furthermore, it protects the data from damage by only exposing it via the `iterator` and `getFrequency` methods.

The WordFrequencyAnalyzer is also cohesive. It is responsible only for analyzing a file and returning the results. If its `getResults` method returns the original copy of the collection, then one could argue that it doesn't protect its data as strongly as possible, in that another object could manipulate that collection unbeknownst to the analyzer, but that flaw is implementation dependent. One could also argue that the class doesn't do its thing very well because of the division of its work into the two `analyzeText` and `getResults` methods. These two methods are inevitably called one right after the other, so one could argue that they should be combined

into one method that analyzes the text and returns the result. There is some merit to that argument.

The WordCounter program is cohesive in that it has one method and so does only one thing. Namely, it provides a simple interface to the user who wants to just pass in a filename and get the results back. The use of a simple interface for a common task is an example of the *Facade* pattern, which we will study in more detail in Chapter 9.

Finally, one could argue that all of these classes lack some desirable methods, but that argument is not related to the cohesiveness of the class or whether it properly separates responsibilities. We will address that argument in a later section of this chapter.

SECTION 5.4 DUPLICATION AVOIDANCE

In the preceding section, we talked about separating the responsibilities among classes and, in particular, making sure that no class has too many different kinds of responsibilities. But there is another question about responsibilities that we haven't yet addressed: Is there anything wrong with two different classes having the same responsibility? Must there be one and only one class given each responsibility? The answer is easy to guess.

Guideline

Avoid duplication.

This guideline is also known as the *DRY* principle: "Don't repeat yourself." It actually says much more than "don't duplicate the responsibilities among classes." Duplication can occur in many forms, some of which have been addressed in earlier chapters. For example, there might be duplicate copies of the same information, there might be duplicated code within a method or between two methods, there might be duplicate methods in two different classes. In addition, there might be duplication of processes, that is, unnecessary duplicate execution of a piece of code. All of these forms of duplication result in designs and code with bad smells.

What's actually wrong with duplication? In summary, code with duplication is less readable and less maintainable than code with no duplication.

To give you a better feeling for the problems that can occur from duplication, let us consider data duplication. Why should duplicate copies of the same data be avoided? If several classes need to know the same information—for example, if there are several school administration objects that need to access the same student records—doesn't it make sense to give them all copies of that information? In some cases it does make sense if the information never changes (i.e., it is immutable). In that case, duplicate copies are okay, although they can waste space, especially if

they are large. However, if the information that is duplicated is modifiable, then it is very easy for the copies to become unsynchronized; that is, one copy might be updated while, for some reason, maybe accidentally, one or more of the other copies are not.

Instead of duplicating the data, how about keeping only one copy of the data but allowing duplicate points of access to that data? Is that also bad? For example, suppose you have an object of class Company that is responsible for maintaining a collection of Employee objects. Suppose the Company class implements its collection with an ArrayList<Employee>. Other objects may need to use the Employee objects and so the Company might have a `getEmployees` method that returns the ArrayList of Employee objects. As a result, two objects may have access to the same ArrayList. Is that bad?

The problem is that the other objects may maliciously or unintentionally render the collection invalid by removing Employees who should be there or by adding new Employee objects to the ArrayList or even adding null. In this case, the Company is not properly maintaining the data against intentional or unintentional corruption by others.

What should the Company do instead of handing over its collection to anyone who asks? There are several options:

1. Replace the `getEmployees` method with an `iterator` method that returns an Iterator over the ArrayList. Iterators are discussed in more detail in Chapter 7.
2. Have the `getEmployees` method return an ArrayList of the Employees, but make it a new ArrayList that is a shallow clone of the Company's ArrayList.
3. Assuming that other objects rarely need to see all the Employee objects, have a getter method that finds and returns an employee specified by particular criteria.
4. Have the Company class follow the Expert pattern and do all the manipulation of Employee objects for the other objects so that the Employees are always hidden. That is, do not give other objects direct access to Employee objects, and instead force those other objects to ask the Company to access the Employee objects for them.

The issue here relates to how much you trust your collaborators. A class' default behavior should trust no one. In particular, if the Company doesn't trust other objects, then it should never let them see the actual ArrayList itself, but only the data in the ArrayList.

Guideline

Only one class should be responsible for knowing and maintaining a set of data, even if that data is used by many other classes.

A corollary to this guideline is that you should always keep data in only one place.

In a good design, one class is chosen as being the "gatekeeper" for the data, that is, the one responsible for maintaining the primary source of data. Other objects must ask the gatekeeper for a reference to that data when they need access to it. The other objects can then temporarily use or even modify the data passed to them, if appropriate. But they should not, in general, have a permanent reference to the primary source of the data nor should they create or maintain their own copy of it except on a temporary basis.

Also, just as methods should always keep objects in a well-formed state, so must the gatekeeper always keep its data is in a well-formed state, which requires it be the sole object with access to the information.

That said, consider the situation in which you have a gatekeeper of a massive collection of objects. If the collection changes infrequently but there are frequent requests for the size of the collection, then it makes little sense to traverse the collection counting the number of objects every time its size needs to be known. Instead, for efficiency, it makes sense to have a separate variable holding the size of the collection that is updated when the collection changes and can then be returned when a request is made for the collection's size. But doesn't this situation violate the guideline against duplication? That is, the collection itself contains the information concerning its size (although you have to traverse the collection to get that information), and the extra variable also contains information about the collection's size, thus duplicating that information. In fact, this is not a serious violation of the guideline against duplication. The key idea of the guideline is that there should be one *primary* source of the information. Other secondary sources, such as the size variable, can exist, as long as it is acknowledged that the primary source is where the actual data resides.

What about the other forms of duplication we mentioned above, such as code duplication within a method or method duplication? Like data duplication, the problem with code duplication is the copies can become "out of sync" when one copy is updated but not the others. Many of these forms of duplication of code can be handled by refactoring. For example, if you have a method with duplicate sections of code, refactor it by extracting the duplicated section into a new auxiliary method. If you have two classes with duplicate methods, then move the duplicated method up into the superclass. Removing code duplication more generally, however, is not so easy because sometimes it can be subtly hidden.

In addition, as mentioned in Section 4.6, there is another form of duplication that can cause problems, namely, the duplication between the actual software itself and the documentation (internal or external) of that software. External and internal documentation must precisely reflect the design and the implementation of the software because, if they become out of sync, then the documentation becomes misleading, which can be worse than no documentation at all. If you could generate the documentation from the source code or generate the source code from the documentation automatically, then duplication problems are avoided, but such generation is not easy to do. Javadoc tools will convert one form of documentation to another, but they do not help keep the code and documentation in sync.

Duplication of processes should be also avoided for efficiency reasons. An example of where such duplication might occur is the checking of preconditions, as discussed in Section 4.5. If a method has preconditions that must hold for it to work properly, who should check those preconditions? Should the caller of the method check, or should the method itself do the checking when it first starts executing? It would be wasteful of resources if they both checked. That is, the duplication of the process of checking the preconditions should be avoided. It is interesting, however, to note that the practice of defensive programming can lead to such duplication. A good defensive programmer, like a defensive driver, always assumes that others will do something wrong and so is prepared. Such preparation naturally includes checking preconditions. But if the calling program and the callee are both coded defensively, this precondition check may in fact be duplicated.

Finally, in addition to everything we have said so far about duplication of data, code, documentation, and processes, you should keep in mind that the appearance of any form of duplication in your software might be a symptom of a bigger problem in your design or implementation. If you see a lot of duplication, you should probably rethink the roles, responsibilities, and relationships among the classes and components involved to see whether a better design can be found that eliminates the duplication.

Is there any duplication in our word frequency counter classes? The WordFrequencyCollection class is the gatekeeper for the word frequency data. It only returns the data via `iterator` and `getFrequency` methods and so trusts no one, as is appropriate. The WordFrequencyAnalyzer class creates the WordFrequencyCollection, fills it with words, and then returns the collection to anyone who asks for it. As a result, others could modify the collection. Does that mean that the analyzer is failing in its task of gatekeeping the collection? In fact, the analyzer's responsibility is to create and fill the collection using words from a file, not to manage the collection after that point. So, in that sense, it is not failing at its task. There is, in fact, little duplication between the three classes, although, depending on the implementation, there may be some duplication within any one of the classes.

SECTION 5.5 COMPLETE AND CONSISTENT PROTOCOLS

Up to this point in this chapter, our discussions have centered on determining the purpose of each class. That is, what concept should each class represent or what role should a class play? The preceding sections argued that each class should play one and only one role in order to maximize cohesion and separate responsibilities. Furthermore, they argued that each responsibility should be assigned to exactly one class.

But once we have decided on the role of a class, then we still need to decide what behaviors and attributes the class should have with regard to that role. Do we include just the minimal necessary behavior? Or, while we are at it, should we add

more behavior to the class in the hope of making it more reusable? What methods should we include to implement this behavior?

The answers to these questions vary from class to class. Some classes are very application specific. In particular, a class with just the main method that executes when an application starts running is not a class that will be reusable for other applications, and so it makes little sense to spend a lot of energy designing that class for reusability. However, considerable attention and energy should be devoted to the design of other classes, especially classes that will become part of a library on which many other applications will depend. For the rest of this section, we will assume we are talking about such classes, and so we will consider their design very carefully.

Guideline

Give classes a complete interface.

By "complete," we mean that the class should have the full set of appropriate behavior so that it can perform any reasonable action related to the role that it plays. If, for example, we have a GUI component and it has a `setSelected` method that highlights the component, then it should have a `setUnselected` method to remove the highlighting. Even better would be to have one method `setSelected(boolean b)` that highlights the component if `b` is true and unhighlights it if `b` is false. For completeness, such a component should also have an `isSelected` boolean function that tells you whether the component is currently highlighted.

How full should the set of behaviors be? That is, how many public methods should be included? One extreme is to create the minimum number of essential methods. By "essential" methods, we mean methods that any class that does one thing well must have. Such methods' implementations typically are intimately tied to the representation of the data stored in the class. Informally, we can think of an essential method as one that cannot be implemented by calling other methods of the class. All nonessential methods we call *convenience* methods. At the other extreme from a class with only essential methods is a class that has a large number of convenience methods in addition to the essential methods.

For example, consider a rotatable Rectangle class with a `rotate(double degrees)` method. Suppose that a significant number of the rotations that are expected to be performed are actually 90-degree counterclockwise rotations. Then a `rotateLeft` method could be included in the Rectangle class for convenience. The `rotateLeft` method is not essential, in that it can be implemented by a call to `rotate(90)`. An alternative is to leave the `rotateLeft` method out of the Rectangle class, and let subclasses define that method if necessary. As a third alternative, you could create a RectangleUtilities class that is not a subclass of Rectangle

and that has a number of static utility methods that take a Rectangle as one parameter and do things to it, such as rotate it by 90 degrees.

A designer needs to be careful not to overwhelm the users by adding too many methods to a class in an attempt to ensure completeness. In other words, don't try to anticipate all possible uses to which your class will be put. It is better to include a core set of essential methods, and a few convenience methods, and let the users construct their own convenience methods or extend the class through subclassing if they need a larger range of methods. The key is to make sure that the core set is sufficiently large so that users can create any convenience methods they deem worthwhile.

If you do add convenience methods to your class, should you implement them by calls to other methods or by directly accessing the data stored in the class? The advantage of the direct access is that you might be able to make the convenience methods more efficient. However, there are also advantages to implementing convenience methods through calls to the essential methods. For example, consider the java.util.AbstractCollection class, which was created to make it easy for programmers to implement the Collection interface. The AbstractCollection class implements all the methods in that interface except the `iterator` and `size` methods. The implemented methods in AbstractCollection get information about the data in the collection through calls to `size` and `iterator`. Therefore, a programmer who wishes to create a class that implements the Collection interface need only create a concrete subclass of AbstractCollection and, in that subclass, implement the `iterator` and `size` methods. To create a modifiable Collection, the programmer will also need to override the `add` method and implement the `remove` method in the Iterator returned by the `iterator` method. In summary, by implementing all the convenience methods in terms of the essential methods (`iterator`, `size`, `add`, and the Iterator's `remove`), the AbstractCollection class saves programmers a significant amount of work.

There is one more issue that needs to be addressed regarding a well-formed, reusable class.

Guideline

A well-formed class has a consistent interface.

By "consistent," we mean that the methods that do similar things should be laid out similarly. For example, suppose a class maintains and manipulates an indexed collection of people, and the class has a method that allow the user to set the *i*th person's name and another that allows the user to set the *i*th person's age. In that case, the two methods should have similar names and they should have similar arguments in the same order. That is, the method signatures should be something like `setAge(int age, int index)` and `setName(String name, int index)`. It would be very confusing to the user and hard to remember if, for example, the first method used the index as the first parameter and the second used the index as the second parameter.

Overriding toString, hashCode, clone, and equals

When designing a complete set of methods for a new class, one decision the designer must make is whether to override the `toString`, `hashCode`, `clone`, and `equals` method implementations inherited from the Object class. In Sections 4.7 and 4.8, we addressed *how* to implement the `equals` and `clone` methods if you decide to override the inherited versions, but we had little discussion regarding *when* to override them. Here are some guidelines.

Consider a class that is mutable. In choosing the implementation of `equals`, you need to ask yourself whether you want to require that two objects that are equal remain equal for the duration of the program, even if the state of one of the objects changes. If so, then object identity (the inherited version of `equals`) is probably the one that makes the most sense. Or do you want them to be considered equal only when their transient states happen to match? In that case, you need to override the inherited `equals` method.

In contrast, consider a class in which its objects are immutable. In that case, you typically want to consider two objects to be equal if their (fixed) states are the same, in which case you would want to override `equals` to test equality of state rather than object identity.

However, when you are deciding whether to override `equals`, you need to consider the guideline discussed in the preceding chapter that states that, if you don't want to violate the Liskov Substitution Principle, then, in any inheritance chain, there should be at most one class that overrides the Object class' `equals` method. Therefore, if one of your class' superclasses already overrides `equals`, then your class should not do so.

The `hashCode` method should be overridden whenever the `equals` method is overridden, to avoid problems when the objects are used as the keys in hash tables. In particular, it is important that for all objects `a` and `b`, if `a.equals(b)` is true then `a.hashCode() == b.hashCode()`. A good discussion of this issue can be found in [7].

An overriding `toString` method of your class should almost always be added. The purpose of the `toString` method is to display the state of the object as text in a humanly readable form. The default implementation of `toString` in the Object class displays the name of the class of the object followed by "@" and the object's hash code value, which is not the kind of information you normally want.

When should you override the inherited `clone` method? If your class is immutable, then it makes no sense to use `clone` at all. (Why would you ever need a copy?) In contrast, if your class is mutable, it might make sense to override `clone` if you want to let others create copies of your object. Keep in mind, as discussed in Section 4.8, cloning has to be done very carefully (for example, do you want a deep or shallow clone?). Also there are other alternatives that you should seriously consider instead of cloning that provide the same service. For example, you could provide a copy constructor for your class. That is, for a Triangle class, provide a constructor of the form

```
public Triangle(Triangle t)
```

that constructs a new Triangle that is in the same state as the Triangle passed as a parameter. This constructor has effectively created a clone for you. Another alternative is to provide a factory method similar to the factory method discussed at the end of Section 2.6. Such a method for the Triangle class might look like

```
public static Triangle
    createInstance(Triangle t)
```

This method could even be in a separate factory class that also creates instances of other classes. But keep in mind that, regardless of whether you override `clone`, use a copy constructor or use a factory method , the issues discussed in Section 4.8 must be addressed, especially if you want a deep clone.

An example of inconsistent behavior-in this case not within a class but among several classes—is the way the size of a Java Collection is determined. To find the size of an array `A`, you use the public instance variable `A.length`. To find the size of an ArrayList `v`, you use `v.size()`. To find the size (number of characters) of a String `s`, you use `s.length()`. Avoid this kind of inconsistency as much as possible.

To finish this section, let us see how well our three word frequency counter classes conform to the guidelines in this section. The WordCounter class is an example of a class that is application-specific, and so giving it a complete and consistent interface is not very appropriate. The WordFrequencyAnalyzer makes one computation and returns the result, and so there is little else that is needed. We could extend its behavior to make it a little more complete by having it reuse the same WordFrequencyCollection in every analysis and adding a `clear` or `reset` method that clears the WordFrequencyCollection. Separating the resetting from the analyzing puts an extra burden on the users in that they need to remember to call `clear` if they want to start over with new text. But it also gives them the flexibility of being able to accumulate the word frequencies from more than one file. The WordFrequencyCollection is the worst offender of this section's guidelines. For example, it has an `add` method but not a `remove` method. To be reusable, it should, in fact, implement most or all of the methods in the java.util.Collection interface.

SECTION 5.6 MUTABILITY VS. IMMUTABILITY REVISITED

In the preceding section, we talked about the need for symmetry in our class' public interface. In particular, if there is a `getX` method, you should consider adding a matching `setX` method. This makes sense for mutable classes, but not for immutable ones. Therefore, as the design of a project is being refined, one of the issues that needs to be addressed is whether various classes should be made mutable or immutable. The Java libraries include both kinds of classes. For example, the java.util.Date and java.awt.Point classes are mutable, but the String is class immutable. How do you decide which path to take for each of your classes? In this section, we will address that question.

One of the reasons the Java API designers made the String class immutable is to allow the sharing of Strings. Every place the String `"hello"`, for example, is needed in a program, exactly the same String object can be used, which can save a considerable amount of memory. (To handle the case where the user wants to modify a string, the Java designers created the StringBuffer class, whose objects are mutable.)

One can think of immutable objects as ones with guaranteed behavior. You never need to worry about whether they follow particular invariants, nor do you ever need to recheck their state once they have been created. In particular, an immutable object will always give the same result from repeated calls to one of its functions, if those functions don't access external data.

One of the disadvantages of immutability is that there can be no modifier methods, which means that if you want change the state of an object, you need to create a whole new object, even if you will never use the original object again. For example,

if you have a very long string `s` and you want an extra letter `"a"` at the front, you have to create a StringBuffer object to use or else write code such as

```
s = "a" + s;
```

which will create a new string from `"a"` and `s` instead of modifying or reusing the old string referred to by `s`.

What classes in your design should be immutable? The general answer is "as many classes as possible" because the advantages to immutability far outweigh the disadvantages. In particular, objects at the lowest level, that is, representing data, that are just "value objects" should be immutable. The class wrappers for the primitives types (Integer, Float, etc.) are examples of such immutable classes.

Once you've decided that one of your classes A should be immutable, how do you go about making sure it is so? The three most important things to do are to:

1. Make all instance variables of A private (which should be done in virtually all classes anyway for reasons mentioned earlier).
2. Exclude any modifier methods from A.
3. Prevent the overriding of methods by subclasses of A by making A final or by making all of A's methods final.

To guarantee immutability, there are some more subtle problems to consider as well, some of which are discussed in the exercises, the rest of which are beyond the scope of this book. A more detailed analysis of the issues surrounding immutability can be found in [7].

What if you need the use of a class that is identical to a mutable class except you need your class to be immutable? For example, suppose you need an immutable version of the java.awt.Point class. One approach would be to ignore that Point class and write your own independent FixedPoint class from scratch, since such a class is so simple.

A second approach would be to make FixedPoint a subclass of Point that overrides the modifier methods (`move`, `translate`, and `setLocation`) so that they either do nothing or throw an exception. See Figure 5.8. Note that the Collections

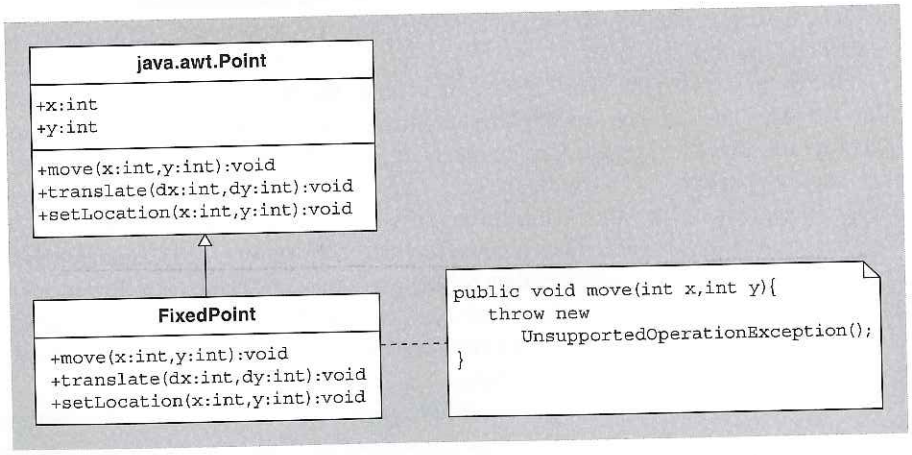

FIGURE 5.8 FixedPoint class implemented as a subclass of java.awt.Point.

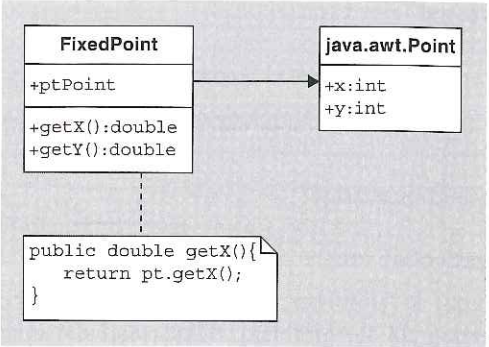

FIGURE 5.9 FixedPoint implemented using delegation to java.awt.Point.

class in the java.util package takes this approach in that it has six methods named `unmodifiableX`, where *X* is Collection, List, Map, Set, SortedMap, or SortedSet. These methods take an object of class *X* that is modifiable and return a new read-only (immutable) view of *X*. In the new collection, any call to a modifier method results in an `UnsupportedOperationException` being thrown. This approach unfortunately will not work for us. Our FixedPoint class will not be immutable even if `move`, `setLocation`, and `translate` are overridden because the Point class has public instance variables `x` and `y`, which are accessible and modifiable from anywhere. Subclasses of Point cannot hide these inherited public instance variables.

A third approach is to use delegation instead of inheritance. See Figure 5.9. That is, create a new "wrapper" class with a read-only interface that contains a Point object in which it stores its data. Consider the following example:

```
public class FixedPoint //attempting immutability using delegation
{
  private Point pt;
  public FixedPoint(Point p) //constructor
  { this.pt = p; }
  public FixedPoint(int x, int y) //constructor
  { this.pt = new Point(x, y); }
  public double getX() { return pt.getX(); }
  public double getY() { return pt.getY(); }
  public Point getLocation() { return pt; }
}
```

Notice that there are no modifier methods, and so the interface is much cleaner than if you were to subclass Point. Unfortunately, we are not out of the woods yet. This FixedPoint class, at first glance, might look like an immutable class, but there are two problems. The first one is that the `getLocation` method returns the Point being wrapped, which leaves it exposed to modification by the user. That is, the user could write code like the following, which modifies the FixedPoint:

```
FixedPoint fp = new FixedPoint(3, 4);
System.out.println(fp.getX()); //prints 3
```

```
Point loc = fp.getLocation();
loc.x = 5;
System.out.println(fp.getX()); //prints 5
```

One way to fix that problem is to reimplement the FixedPoint's `getLocation()` method as follows:

```
public Point getLocation()
{
  return pt.getLocation();
}
```

(Why does this version fix the problem? Hint: read the documentation for the `getLocation` method in the java.awt.Point class.)

But even with that fix, there is another subtler problem with our class that makes it mutable. Here is some user code that shows this problem:

```
Point p = new Point(3, 4);
FixedPoint fp = new FixedPoint(p);
System.out.println(fp.getX()); //prints 3
p.x = 5;
System.out.println(fp.getX()); //prints 5
```

The problem is that the internal point has external references to it and so can be modified. To avoid this problem, the FixedPoint class needs either to get rid of the constructor that takes a Point as an argument or to make a "defensive copy" of the Point passed in as the argument and then use this hidden copy. Here is code that will work:

```
public final class FixedPoint //corrected version
{
  private Point pt;
  public FixedPoint(Point p)
  { this.pt = new Point(p); } //copy constructor
  public FixedPoint(int x, int y)
  { this.pt = new Point(x, y); }
  public double getX() { return pt.getX(); }
  public double getY() { return pt.getY(); }
  public Point getLocation() { return pt.getLocation(); }
}
```

With this implementation, the `pt` instance variable refers to a Point object to which there is no outside reference and to which there is no way to get an outside reference, even with subclassing. So the FixedPoint class really is immutable.

SECTION 5.7 DESIGNING FOR CHANGE

Much of our discussion in this chapter and earlier chapters has had to do with reusability and readability of software. For example, we want intention-revealing names so that our code will be readable. We want our classes to have cohesion and

a complete and consistent interface so that they can be easily reused. However, there are other, just as important issues to consider when designing classes. These are the issues of maintainability, modifiability, and extendibility.

There are many reasons why modifications to existing classes may be necessary, including fixing bugs, making optimizations, and adding new behavior. The hard part of making such modifications is doing it in a way that does not introduce new bugs or make the code "brittle". In fact, the technique of refactoring, discussed in the preceding chapter, was developed for just the purpose of changing code without adding bugs.

Well-written code is designed with the possibility of future change in mind. This statement has also been phrased as [8]: "Change happens. Deal with it." Another way to say it is [9]: "To prosper, an interface must be well suited for its task—simple, general, regular, predictable, robust—and it must adapt gracefully as its user and its implementation change." In other words, to have a maintainable, modifiable, and extendible system, you need to accept the fact that changes will need to be made to it and prepare for them.

Guideline

Design your classes and interfaces so that they can handle change.

The rule above is easier stated than followed. How can you design for change when change can come in so many forms? In fact, you can't anticipate all the forms of change, but there are general guidelines that you can follow as you design your code that will greatly improve the ability of your classes to handle change when it does happen. One such guideline is known as the Open-Closed Principle [10].

One way to state the Open-Closed Principle is: "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"[11]. That is, you should design your classes and modules so that they are easy to extend if new behavior is needed, but design them in a way so that existing classes never need to be modified. That is, design them so that they are easy to extend by adding new classes, extending existing classes, and reusing existing classes, but without modifying existing classes. Note that "extension" here does not just mean Java inheritance. Rather we are referring more generally to extending a system by adding new features or new behavior.

The benefit of following this principle should be clear. When software needs to be changed, as it always does, one approach is to change the existing classes. For example, you can change your software by adding new methods or changing existing methods of your classes. Unfortunately, this approach will likely require you to change other classes as well that depend on the changed classes, which in turn may require you to change classes that depend on these classes, and so on. Not only are all these cascading changes time consuming, but they are likely to introduce new errors in the code. Clearly it is advantageous to minimize the modification of existing, working code and instead to extend that code by adding new classes that incorporate the changes, leaving the old classes alone.

Guideline

Follow the Open-Closed Principle when designing software. That is, design software to be open for extension but closed for modification.

One situation for which it is particularly important that you follow the Open-Closed Principle is when you are writing code for libraries. The users of such libraries often do not have access to the source code for the classes in the libraries, and so they cannot modify the code even if they want to. But even if they have access to the source code, changes to that code are likely to introduce new bugs and break other code. Think about the mess they would be in if, for example, programmers rewrote the Swing classes every time they needed new behavior from them instead of subclassing them or combining them in new ways! In contrast to libraries, code that is specific to one application is more likely not to be reused or extended, and so you have to decide whether it is worth the time and energy to follow the Open-Closed Principle.

How do you design classes so that the Open-Closed Principle holds? Let us start with some examples.

Suppose you were told to write a program to manage the purchase/maintenance/disposal of a company's fleet of vehicles, which happen to all be Ford minivans. Can you assume that in the future the vehicles will always be Ford minivans? Not if you want the software to be closed to future modification. Can you assume that the vehicles will always be purchased from Ford? Not if you want your software to be closed. In particular, if you have a Vehicle class with an instance variable `dealer`, the type of dealer should not be FordDealer that stores information about the Ford dealer from whom the company purchased the vehicle. Instead, the instance variable should be of type Dealer, which could represent all kinds of dealers.

For another example, consider an electrical outlet in your home that currently (pun intended) provides power to an electric clock you purchased from Acme 10 years ago. There are at least three ways to view the outlet, with analogies in OO design:

1. If we think of the outlet only as a Acme clock outlet (that is, if we reserve that outlet only for clocks of the brand of the current clock), then, if Acme stops making clocks, the outlet becomes useless when your clock breaks. In this view, the outlet is analogous to a AcmeClock variable that can refer to (plug in) only clocks of type AcmeClock.
2. If we think of the outlet as a "electric clock" outlet, then, when our current clock breaks or becomes obsolete, we can buy a new electric clock of any brand or style and plug it into the outlet. In this view, the outlet is analogous to a variable of the type of an abstract ElectricClock class that can refer to objects of any subclass (brand) of ElectricClock. This view of the outlet is much more useful that the first view, but it is still restrictive in that only electric clocks are allowed to use the outlet.
3. However, if we think of the outlet as an "electric appliance" outlet, then, we can plug any appliance, not just a clock, into the outlet. The appliances need have

no relationship with each other other than the fact that they have a plug that fits the outlet (the electrical interface). This view corresponds to having an ElectricAppliance interface and a variable of type ElectricAppliance that can contain any object of any class that implements the ElectricAppliance interface.

It is clear that the third view provides much more flexibility to the users of your classes by making it easier to change the value of the variable when necessary. The idea is to define your variables and values to have the widest possible type rather than the narrowest type. The widest possible type in Java is an interface that can be implemented by any number of classes.

For another simple Java example, consider a library with the following method that returns true if the given object is in the given linked list:

```
public static boolean contains(Object o, LinkedList L)
{
  for( Object l : L )
    if( l.equals(o) )
      return true;
  return false;
}
```

This method might be useful for someone with a LinkedList to search, but it is not useful if that person stores her data in some other structure. Can we improve this method to make it more generally useful? If you look carefully at the implementation, you will see that the only behavior of the LinkedList class that is being used is its `iterator` method. Therefore, the `contains` method will still work correctly and be much more generally useful if we replace "LinkedList" with "Collection" as the type of parameter L. Then any class that implements the Collection interface can be searched by this method. But we shouldn't stop there. We can make it even more generally useful by widening the parameter type further. A wider type than Collection is Iterable, the interface implemented by all classes with an `iterator` method. In summary, by converting the signature of the method to

```
contains(Object o, Iterable L)
```

we still have a correct method, but we have made it as generally useful as possible.

Guideline

Code to interfaces, not classes. That is, wherever possible write your code so that objects are referred to by the interfaces they implement instead of by the concrete class to which they belong.

One way to view coding to interfaces or, more generally, designing your classes to handle change is to think of it as a way of making it easy to back out of any design or implementation decisions that you previously made. For example, if you hard wire your code to refer to a TreeSet, then it is hard to back out of that design and instead use another container such as a HashSet. In contrast, if you had written

your code so that the TreeSet was stored in a variable of type Set (an interface) and only the interface's methods were used, then it would be easy to replace the TreeSet with a HashSet and little or no other code would need to be modified.

Note that much of our discussion in this section can be thought of in terms of minimizing the interconnections between parts of the program. For example, by coding to interfaces instead of to a concrete class, you reduce the interdependence among the concrete classes. The concept of reducing such connections goes by the name of "minimizing coupling." Other equivalent terminology for this concept is "keeping classes orthogonal" and "minimizing the interdependence among classes." The principle of minimizing coupling is another example of using GRASP [2].

We can also consider interdependence on a larger scale, such as among groups of classes, than just between two classes. For example, suppose we decide that we need to replace a group of classes that work together to accomplish a task with another group of classes that accomplish the same task. More specifically, let's assume our application uses 20 classes, and a group of 5 of them need to be replaced with a group of 4 new classes. Suppose further that each of the 20 classes uses every other class. In this case, the UML class diagram for those 20 classes will contain 380 associations and will look like a complete graph with 20 vertices. In this case, every class will have to be modified to accommodate the replacement of the five classes. However, if the classes had been designed so that each class is minimally coupled with the other classes, then the group of 5 cooperating classes may have lots of associations among themselves, but the 15 classes outside the group of 5 would most likely have only a few connections to the group of 5, making it much easier to replace those 5 classes. In other words, to accommodate change, it is advantageous for a UML class diagram to look more like a tree than a complete graph.

What else can you do to your software design so that the Open-Closed Principle holds? Two techniques that can help, if used properly, are encapsulation and information hiding.

Encapsulation [1] means grouping together related items and putting a wall around them or protecting them from access by others. Although encapsulation can be done in most any programming language, object-oriented languages like Java provide a natural mechanism, called a *class*, for encapsulating data and the methods that operate on that data. Java also provides a package mechanism for encapsulating groups of classes.

How can the encapsulated items be protected? One technique that is helpful is *information hiding*, which means keeping information hidden from others. In particular, it is helpful to keep the implementation of a class or group of classes hidden as much as possible from the users of the class or classes. If information in a class is exposed, then clients are likely to use it, resulting in tighter coupling between that class and the clients' classes. One way to hide information has been mentioned before: keep all its instance variables private. This privacy is necessary if in the future we want to be able to change the implementation of a class without requiring other classes to also change. But there is another reason to keep instance variables private even if the implementation of the class will never change. We will explore that reason through an example.

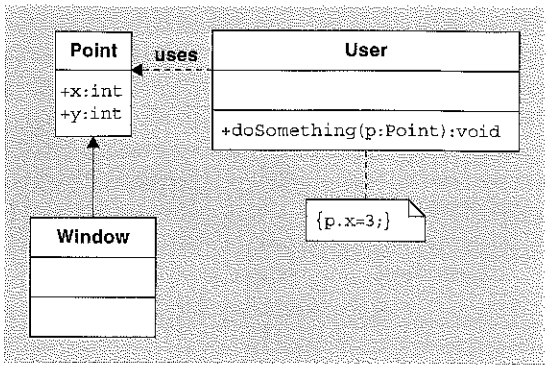

FIGURE 5.10 The Window may have no way of knowing the user changed the value of `p.x`.

Let us suppose you have a software system using a Point class with public integer instance variables `x` and `y`. Let us also suppose that in the next enhancement of the system, you need to display in a window the current values of `x` and `y` of a Point `p` and you need to keep the values in the display updated so that whenever `x` or `y` change, the value in the display changes accordingly. How can you ensure that the displayed value of `x` and `y` will always be updated when a change is made to `p`? Because of the public accessibility of `x` and `y`, any other object that is aware of `p` can arbitrarily change the value of `x` or `y` without notifying anyone. For example, in Figure 5.10 a User and a Window both have access to a Point `p`, but they are unaware of each other. When the User's `doSomething` method is executed, the value of `p.x` changes, but the Window may be unaware of that change.

One solution to this dilemma is to have the display window repeatedly poll `p` to see whether `x` or `y` have changed. Alternatively, we could restrict the knowledge of the existence of `p` to only a few other classes who know that they need to notify the window when they change the value of `p`'s `x` or `y` variables. However, both these solutions are suboptimal in terms of the inefficiencies they impose on the system and the brittleness of the code. This problem stems from the public accessibility of the Point's instance variables `x` and `y`.

In contrast, suppose that the Point class had instead been implemented with private instance variables `x` and `y` and public getter and setter methods for them. Although this class seems identical to the version of Point with public instance variables, it, in fact, allows a cleaner solution to the problem through the appropriate subclassing of Point. For example, we could define a new subclass of Point called NotifierPoint that overrides the `setX` and `setY` methods of Point. In this subclass, the overriding methods would, in addition to changing the value of `x` and `y`, notify the window (or anyone else that wishes to be notified) that a change has been made and give the window the new values. If `p` is of class NotifierPoint, then the window need not repeatedly poll to find the value of `p`'s `x` and `y` and, furthermore, we need not restrict access to `p` to objects that are aware of the need to notify the window of any change. If any object wishes to change `p`'s values, that object merely uses one of `p`'s setter methods and the window will automatically be notified of the change. Therefore, the version of point with getter and setter methods is more open for

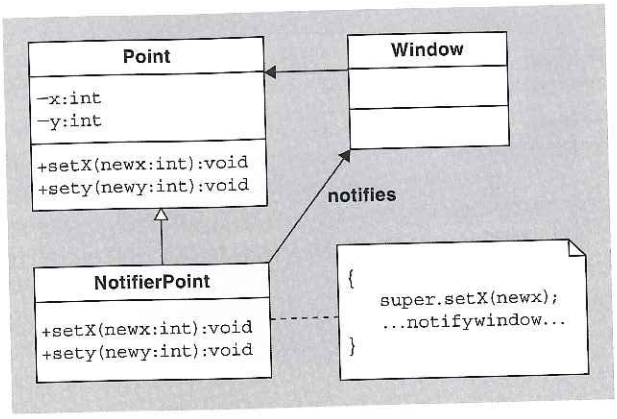

FIGURE 5.11 The NotifierPoint notifies the Window of the change in x.

extension. See Figure 5.11. (By the way, this idea of notifying interested parties when something happens is called the *Observer* pattern and is discussed in Chapter 8.)

In summary, private instance variables and, if necessary, public getter and setter methods are almost always preferable to public instance variables.

What else can you do in terms of information hiding besides making instance variables private? You can also make auxiliary methods private (in other words, keep the public protocol of the class small and clean). In addition, widening the return type of the public methods as much as possible will help hide implementation details. For example, consider a class that stores data in a HashSet and suppose there is a getter method for the HashSet. The getter method could be declared as `public HashSet getHashSet()`. This is the narrow version. It gives away the fact that the data is stored in a HashSet (or subclass of HashSet). If the getter method is declared as `public Set getSet()`, then, even if the same HashSet is returned, the actual class of object being returned is hidden.

How much of a class' implementation do you encapsulate and hide? The general rule is to encapsulate and hide as much as possible. It is far easier to expose something later if it no longer needs to be hidden than it is to hide something that was previously exposed. Hiding an item that has been exposed may require global changes to the system.

In summary, one of the most important things you can do to make your code elegant is to design them to handle change. The Open-Closed Principle states more specifically what you should strive for. Coding to interfaces, minimizing coupling, and using encapsulation and information hiding are guidelines to help you reach this goal.

How well are our three word frequency counter classes from Figure 5.5 designed for change? One could argue that there are only three classes and that this is a very small program, so it is not worth worrying about it. But, for the sake of discussion, let us evaluate it anyway. If all methods except the ones shown in Figure 5.5 and all instance variables of the classes are kept private, then the three classes properly hide their implementation details. Is there a way we can further reduce the coupling to facilitate future changes in the design or code? If you review the design and code, you will notice that there are no interfaces used. We could make the WordFrequencyAnalyzer more generally useful by changing `analyzeText` so that it takes a more

general kind of input than a File. For example, it could take an Iterator<String> or a Readable object as its parameter. In addition, note that the analyzer returns a WordFrequencyCollection in the `getResults` method. But does the return type need to be WordFrequencyCollection? That is, does any user of the analyzer need to know that the particular class of the object returned is WordFrequencyCollection? If the only methods of that collection that the user invokes are its `iterator` and `getFrequency` methods, then the return type of `getResults` could be an interface whose protocol includes those methods.

SECTION 5.8 LAW OF DEMETER

There is one final topic we want to address in this chapter. It relates to the expert pattern and the coupling between classes. It can also influence greatly how well your software can handle change. This topic is the Law of Demeter.

To understand the Law of Demeter, consider a general in an army that is setting up a base of operations in the field. One of the many jobs that need to be performed is the digging of foxholes. Here is one way that task could be accomplished. The general could get one of his colonels and tell him to get a major. The general would then tell the major to get a captain. The general would then tell the captain to get a sergeant. The general would then tell the sergeant to get a private. The general would then tell the private to dig some foxholes.

This approach is quite ridiculous, isn't it? And yet we might easily see the equivalent of that approach in code:

```
general.getColonel(c).getMajor(m).getCaptain(c).getSergeant(s)
.getPrivate(p).digFoxhole();
```

Just as it is not appropriate for the general to go through all those steps for getting a foxhole dug, it is inappropriate for code to go down such a message chain to get a task done.

Consider a more realistic but similar chain that might appear in an ATM application, where the program reads from the ATM card the bank name `b`, the branch number `r`, the customer name `c`, the account number `a`, and then attempts, starting with some associated CentralControl object, to get the customer's balance in that account:

```
Balance balance = centralControl.getBank(b).getBranch(r)
                  .getCustomer(c).getAccount(a).getBalance();
```

where `b` is the bank name, `r` is the branch name, `c` is the customer name, and `a` is the account number. Do you see the problems with such code? One problem is that you have now strongly coupled the ATM application, CentralControl, Bank, Branch, Customer, Account, and Balance classes. Any changes to the structure of any of the latter classes could affect the code in all the former classes. Just as important, such `getter` methods give the user access to the Bank, Branch, Customer, Account, and Balance objects and so the user could manipulate them (that is, invoke methods on them) in ways that are not intended for anyone other than a privileged set of objects. This design has a bad smell called *inappropriate intimacy*.

What are better ways to handle these situations? Let us go back to the general in the army and consider what such a general would really do. Most likely he wouldn't

even concern himself with details such as digging foxholes. But if he did, he would certainly call in one of his officers and tell him, "I don't care how it is done, but get someone to dig foxholes."

Similarly, one possible way to handle the request for an account balance would be to do something like the following:

```
Balance balance = centralControl.getBalance(b, r, c, a);
```

In this version, it is up to the CentralControl class to delegate or forward the responsibilities to other classes as necessary to get the desired balance. For example, the CentralControl could find the Bank with the name `b` and pass it the values of `r`, `c`, and `a` and ask it for the balance. The Bank could continue similarly. As a result, the classes are less strongly coupled in that the main application doesn't need to worry about the existence of Customer or Account objects and can leave those details to CentralControl or other classes with which the CentralControl communicates.

The distinction between the two ATM approaches can be seen in the UML sequence diagrams in Figures 5.12 and 5.13.

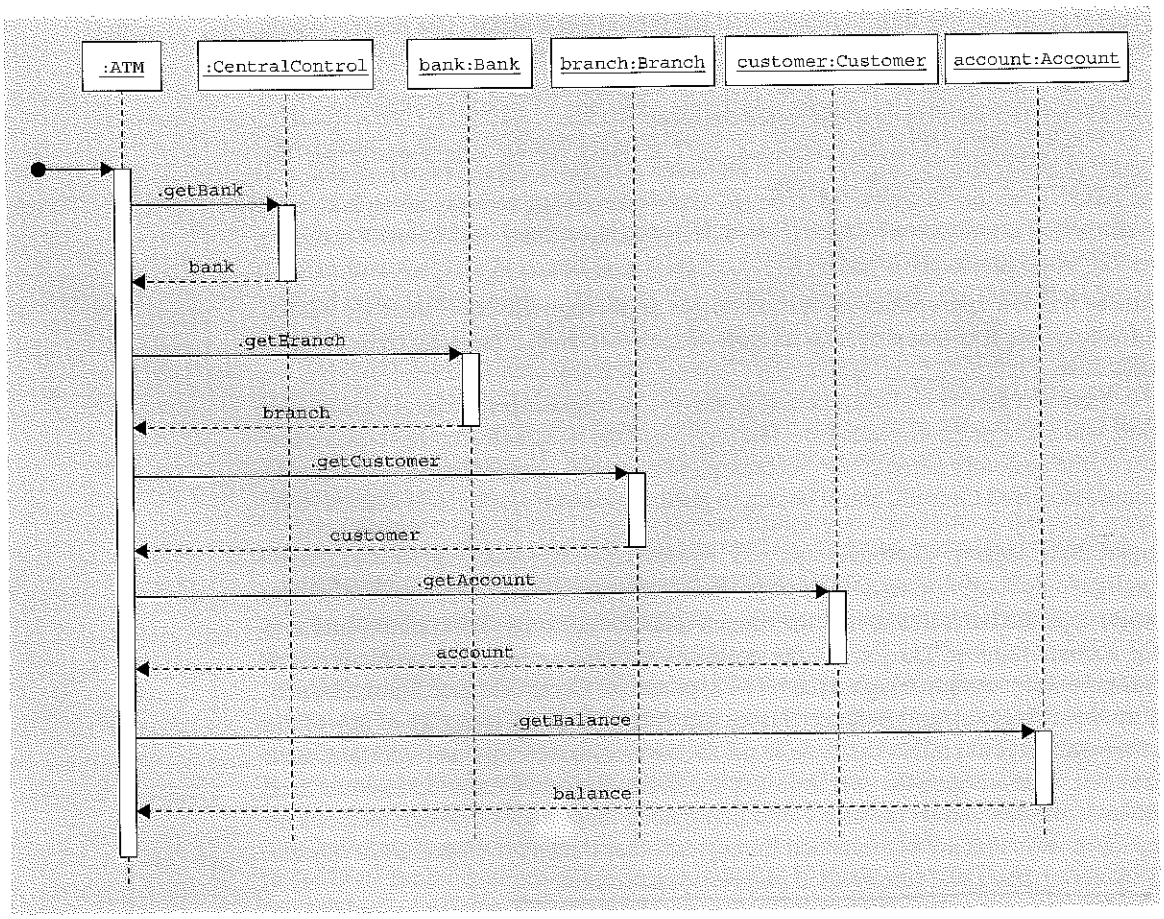

FIGURE 5.12 The UML sequence diagram for the first approach at getting the balance.

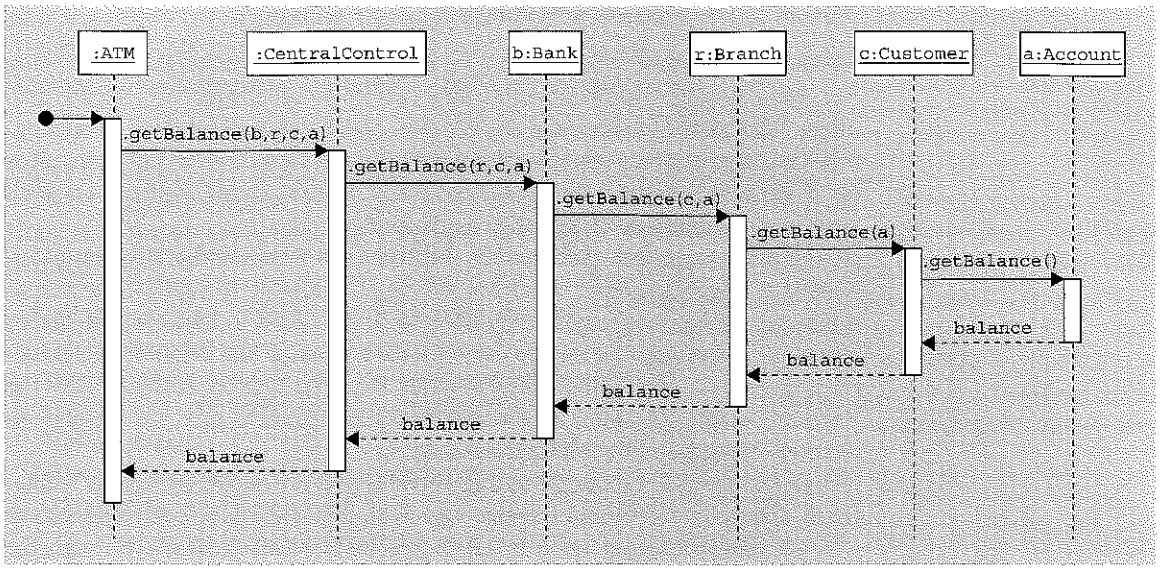

FIGURE 5.13 The UML sequence diagram for the second approach at getting the balance.

A UML *sequence diagram* is a diagram that shows message passing along a timeline so that you can see the order in which the messages are passed. Each rectangle at the top of the diagram corresponds to an object. Each object has a name and a class, separated by a colon. If the name or the class is not important, it can be omitted. The vertical axis corresponds to time, with time increasing as you go down. The thin white strip below each object indicates when the object is actively executing a method (i.e., when it has an activation record on the stack). The solid arrows between the timelines indicate the sending of a message and the dashed arrows indicate the return from the method call, with an optional label indicating the return value. The black dot with an arrow from it indicates a method call whose source is irrelevant. For more information about sequence diagrams, see Appendix A.

The method chaining seen in these examples are violations of the *Law of Demeter* [12], which states that, inside a method body, an object should send messages only to the following categories of objects:

1. This object itself
2. This object's instance variables
3. The method's parameters
4. Any object the method creates
5. Any object returned by a call to one of this object's methods
6. The objects in any collection that falls into these categories

Conspicuously absent are objects that are returned by messages sent to other objects. This law can also be stated as "Talk only to your immediate neighbors" or "Don't talk to strangers" [13].

Here are some more examples where this issue arises:

1. General contractors, when building a house or larger structure, often employ subcontractors. The subcontractors may in turn get other subcontractors to help them. The general contractor doesn't care about that second level of subcontractors. That is, the general contractor expects the first level of subcontractors to do the jobs for which they are contracted, and the general contractor doesn't care who they employ or how they get it done.
2. In the third example in Section 5.5, we discussed a canvas that displays a collection of shapes, some of which are selected (highlighted). Suppose the shapes know whether or not they are highlighted. If you need to know which shapes are highlighted, you can ask the panel for the collection and then iterate through it, gathering the highlighted ones yourself. However, this approach violates the Law of Demeter. A better approach is to ask the panel to give you the subcollection consisting of the highlighted shapes.
3. Similar to the ATM example above, suppose you are accessing a bank's database, and you need to find the customer with a given name. You could (a) ask the database for a collection of all the customers and then step through the collection looking for the one with the given name, (b) ask the database for a collection of all the customers and then ask the collection to find the desired customer for you, or (c) give the database the name and ask it to do the searching to find the customer for you. Clearly, version (c) is the preferable one from the perspective of the Law of Demeter.

What should you do if you encounter code that breaks Demeter's Law through long message chaining, and you want to refactor it to conform to the letter of the law? There are a variety of refactorings that might apply, including Hide Delegate, Extract Method, and Move Method[4]. The important thing is that you need to rethink your design to see why there is such a chain and how it might be avoided.

Guideline

Obey the Law of Demeter.

It is important also to mention here that the Law of Demeter should really have been called a "guideline" instead of a "law." It is not an absolute that must always be followed. Otherwise, it would place a tremendous burden on your neighbors in that all requests of their neighbors must be handled through them. Given that you don't always know what kinds of requests will be made (remember that "change happens"), it is hard to anticipate all future requests.

Do our word frequency counter classes obey the Law of Demeter? In fact, the implementation of the `main` method of the WordCounter class breaks the law. Can you spot where? The method gets the collection from the analyzer and then it invokes the collection's `iterator` and `getFrequency` methods (can you see where `iterator` was called?). Luckily for us, this is a misdemeanor rather than a felony. It would be possible to eliminate the crime, for example, by adding a

`displayResults` method to the analyzer class that calls the collection's `iterator` and `getFrequency` methods and prints the results, but it is debatable whether the code would be made that much better by this change.

SECTION 5.9 SUMMARY AND FURTHER READING

Here is a list of the guidelines discussed in this chapter:

- When working in the early high-level design phase, keep the discussion at a high level. In particular, avoid wasting time on implementation details and low-level data representations at the initial stages.
- Every class should be responsible for doing one thing only and doing it well.
- Different kinds of responsibilities should be separated among different objects.
- (Expert pattern) The object that contains the necessary data to perform a task should be the object that performs the task.
- Only one class should be responsible for knowing and maintaining a set of data, even if that data is used by many other classes.
- Give classes a complete interface.
- Give classes a consistent interface.
- Design your classes and interfaces so that they can handle change.
- Follow the Open-Closed Principle when designing software. That is, design software to be open for extension but closed for modification.
- Code to interfaces, not classes. That is, wherever possible write your code so that objects are referred to by the interfaces they implement instead of by the concrete class to which they belong.
- Obey the Law of Demeter.

For more detailed discussions of these topics, consider the following references. Many of the topics in Section 5.1 can be found in [3]. The Bertrand Meyer text [10] is a classic that contains huge amounts of useful material on creating an elegant design. Another good explanation and example of the Law of Demeter can be found in [14]. The text [6] contains many good heuristics, similar to many of the guidelines in this text, to help you come up with an elegant design. The text [7] has similar guidelines but for lower-level implementations in Java rather than high-level design.

EXERCISES

1. Finish the design and implementation of the word frequency counting program in Section 5.1. It should gracefully handle all errors, including such errors as a missing command-line argument and nonexistent files.
2. In the design of the word frequency counting program in Section 5.1, the main method created a File, which it passed to the WordFrequencyAnalyzer. An alternative would be to pass the path name to the WordFrequencyAnalyzer constructor and have the analyzer create and deal internally with the File. What are any advantages or disadvantages to this alternate design?
3. In the design of the word frequency counting program in Section 5.1, the WordFrequencyAnalyzer has a void method that analyzes the files and a separate function that returns the results of the analysis. An alternative would be to combine these two behaviors into one method that would analyze the text file and return the resulting WordFrequencyCollection. What are any advantages or disadvantages to this alternate design?
4. In the design of the word frequency counting program in Section 5.1, the WordFrequencyAnalyzer is passed a File that it then analyzes by way of a Scanner that it creates. An alternative would be for the main

program in the WordCounter class to create the Scanner and then pass the Scanner to the WordFrequencyAnalyzer. What are any advantages or disadvantages of this alternate design?

5. In the design of the word frequency counting program in Section 5.1, a WordFrequencyCollection class was included. However, you may have noticed that a HashMap<String, Integer> that stores the words from the file as the keys and the numbers of occurrences as the corresponding values has all the functionality desired of the WordFrequencyCollection class. Therefore, one could eliminate the WordFrequency Collection class and just use this HashMap class in the design. What are any advantages or disadvantages of this alternate design?
6. In the design of the word frequency counting program in Section 5.1, the WordCounter program has one static `main` method in which everything happens. An alternative would be to move all the code from the current `main` method into a nonstatic `displayInfo` method (with the file name as a parameter) and have a new `main` method that just creates a WordCounter object and asks it to execute its `displayInfo` method. What are any advantages or disadvantages of this alternate design?
7. In Section 5.2, we mentioned that you should rethink your design if you have a class that maintains no data and just manipulates data passed to it from other classes. The Math class in the java.lang package is such a class. Have the Math class creators designed a poor class?
8. In Section 5.3, we gave the example of a drawing program and asked who should be responsible for keeping track of which shapes are currently selected. One possibility not mentioned there is the case where the shapes keep track of whether they are selected and, in addition, the view also keeps track of which shapes are selected. This design has the advantage that shapes know whether they are selected, which seems appropriate, but the view does as well, which simplifies manipulation (such as dragging or deletion) of all selected shapes. Give a disadvantage of this approach.
9. In Section 5.3, we used the example of a CarDealer object to get across the point that the object with the data should be the one to manipulate the data for you. In this example, we wanted the CarDealer object to filter the data for us. That is, a CarDealer object with only a `getAllCars()` method is less useful to customers than a CarDealer object with methods that filter the cars according to various criteria specified by the customer, such as `getAllCarsOfType(CarType)` or `getAllCarsOfColor(Color)`. However, there are many filtering criteria that might be applied besides type and color, such as the year, make, model, mileage. Also, the user might want to use the criteria in various combinations, such as "all those blue minivans manufactured by Ford or Chevrolet in the years 2001–2004 with at most 10,000 miles on them". How can a CarDealer object handle all possible combinations and types of filtering without have a separate method to deal with each one?
10. In Section 5.3, we said that a class that stores data for you should also manipulate it for you, as appropriate. One might conclude from this statement that a HashSet, for example, should not just store data, but should manipulate the data for you somehow. For example, consider a HashSet containing a collection of Person objects. The client of such a class might find it necessary to extract just the names of the people and put them in a new HashSet. Ideally, the HashSet class would have a method to do this extraction for you. However, there are an unlimited number of ways that the Person objects might be manipulated, and so you cannot fill the HashSet class with methods covering all possible cases.
    a. Suppose you were asked to enhance the HashSet class. Suggest one new method for that class that could be used to manipulate the data in a variety of ways.
    b. Why wasn't such a method ever added to the HashSet class?
11. Suppose you were told to implement a class whose objects need to store 10–20 pieces of data and the data is of several kinds. Some people might argue that you should store each piece of data in a separate instance variable in your object. Others might argue that it is ludicrous to have one instance variable for each piece of data, and so it would be better to store the data all in one instance variable, such as an array `data` of type Object[]. For example, a Person object could store the person's first name in location `data[0]`, the last name in `data[1]`, the address in `data[2]`, the age in `data[3]`, etc. Give your opinion on these two approaches. Discuss the approach you would use.
12. In Section 5.4, we mention that a vehicle management program should store information on the car dealers in a variable of type Dealer, rather than FordDealer, even if all cars are Fords. How should Dealer and FordDealer be related? For example, should Dealer be

an interface and FordDealer a class that implements the interface? Should Dealer be an abstract class and FordDealer a subclass? Is there a third alternative that might be better? Briefly explain.

13. A class designer is designing a Person class that he wants to be as popular and as reusable as the String class, and so he is attempting to maximize the reusability through the addition of as many methods as he can think of. Some of the methods he has added are listed below.

- `Person(int age, String name, //String address) //constructor`
- `String getName()`
- `void setName(String name)`
- `String toString() //returns the //name of the person`
- `int getSSN() //returns the //person's Social Security Number`
- `String getSSNAsString() //returns the SSN as a String`
- `Person[] siblings() //returns //an array of the siblings of the //person`
- `Race getRace()`
- `Person getBoss() //returns the //person who is employing this //person`
- `boolean isDogOwner()`
- `boolean isUSCitizen()`
- `String getAddress()`
- `int numberOfCarsOwned()`
- `int getAge()`
- `int setAge()`

a. For each method, indicate whether (a) it is a core (essential) method that should be included, (b) it is an inessential but convenient method and so should be included, or (c) it should not be included. Briefly explain your reasoning.

b. Rewrite the signatures and return types of any of the methods if you think they can be improved. Explain each of your changes.

c. Give the signature and return type of any missing methods that should be included in the Person class and explain why.

14. Suppose you are writing a method `getChildren()` in a Person class that returns an array of Persons. Then, among other things, you need to deal with the case where the person on whom the method is invoked has no children. What should your method return in that case? An obvious thing to return is null. Comment on this approach .

15. Suppose you have a class one of whose instance methods does not access (either read or write directly or indirectly) any of the class' instance variables. Comment on the design of this class.

16. We like to preach that all applications should have their `main` method in a class by itself, named for the kind of application it is. For example, if your application simulates rockets, you might call the class with the `main` method "RocketSimulationApp." Give a reason for putting the `main` method in its own class.

17. Suppose you have an IntegerList class that is an ordered mutable collection of (not-necessarily-unique) integers. Suppose it has only the following methods:

```
void cons(int x) //adds x to the front
    //of this list
int first() //returns the first integer
    //in this list
int second() //returns the second
    //integer in this list
int third() //returns the third integer
    //in this list
int nth(int n) //returns the n-th
    //integer in this list
IntegerList rest() //returns a new list
    //identical to this list
  //except that the first integer of
  //this list is omitted
int size() //returns the length of the
    //list (number of elements)
```

Assume that `first`, `second`, `third`, and `nth`, respectively, throw an Exception if there is no first, second, third, or *n*-th integer, respectively, in the list.

a. Show that `second`, `third`, `nth`, and `size` are convenience methods by implementing them using the other methods of the class.

b. Is it appropriate to have the `second`, `third`, and `nth` methods? That is, do they really add enough convenience to be included in the IntegerList class? Briefly explain.

18. Suppose you have an IntegerSet class that represents an unordered mutable collection of integers without duplication (i.e., the same integer does not appear twice in the set). Suppose it has only the following methods:

```
void add(int x)    //adds x to this set
  //if it is not already there
void remove(int x)   //deletes x from
  //this set if it is in the set
```

```
int size()  //returns the number of
  //integers in the set
boolean contains(int x)  //returns true
  //if x is in this set
```

Assume that add does nothing if x is already in the set. Similarly, assume remove does nothing if x is not in the set.

a. Show that contains is not necessary (although convenient) by showing that it can be implemented using the other methods in the class.

b. Using just these methods of IntegerSet, is it possible to write a loop that will print all the elements of this set? If so, do it. If not, say why not.

19. Suppose you are designing an IntegerSet class that represents an unordered collection of integers without duplication (i.e., the same integer does not appear twice in the set). Some methods you might wish to include are the following:

- `int[] toArray() //returns an array //containing the integers in this //set`
- `boolean contains(int x) //returns //true if x is in this set`
- `boolean containsAll(IntegerSet otherSet) //returns true if this //set contains all the integers in //the other set`
- `int size() //returns the number of //integers in this set`
- `boolean isEmpty() //returns true //if this set has no integers in it`
- `boolean equals(Object obj) // //returns true if obj is a Set //containing exactly the same //integers as this Set`

a. For each of these methods, determine whether it is essential. That is, determine whether it must be included in the IntegerSet class or whether it is a method that can be implemented by the user through the use of the other methods in the list above.

b. Find a minimal set of essential methods.

c. Should all six of these methods be included in the IntegerSet class even if they are not all essential methods? Why?

d. Suggest other methods that should be added to the IntegerSet class to make it a do-one-thing-only-and-do-it-well class. Explain.

20. As mentioned in Section 5.5, a well-designed class A will have a toString() method that provides a readable representation of objects of class A. It is nice when class A can also provide a kind of "inverse" constructor that takes such a String as a parameter and creates and returns an object of class A initialized with the data as it was displayed in the String. For example, the java.util.Date class' toString() method returns a string of the form "Fri Jul 01 12:00:00 EST 2002." It also provides a constructor (now deprecated) that takes such a string as a parameter and returns a new Date object initialized with the values given in that string. For this exercise, take the FixedPoint class given at the end of Section 5.6 and add a readable toString() method and a constructor that takes a string of the form returned by the toString() method and constructs a new FixedPoint object initialized with the values given in the string.

21. Let us suppose you are unhappy with the limitations of the java.awt.Rectangle class, and so you subclass it to get an EnhancedRectangle class. Suppose that one of the things that the new class has is a method showCoordinates() that creates and brings up a dialog box displaying the top, left, bottom, right coordinates of the rectangle. What do you think of this enhanced class? Is it a good design?

22. In Section 5.7, we mentioned that String objects are immutable. But what if you have a String s = "hello" and you execute the statement

```
s.replace('e', 'a');
```

Doesn't that change "hello" to "hallo" and so mutate the string? Explain this supposed contradiction.

23. In Section 5.6, we mentioned that subclasses of the Java.awt.Point class cannot hide the inherited public instance variables x and y. But are we really correct? Consider the following classes:

```
public class A { public int x = 3; }
public class B extends A {
private int x = 4; }
```

Also consider what happens when you try to access the public x inherited by B from A:

```
B b = new B();
System.out.println(b.x);
```

The Java compiler will give an error message when you try to compile those two lines, claiming that x is private in B. So in this case, it appears that the private instance variable x of B does in fact hide the public instance variable x that B inherits from A. Doesn't this example show that you *can* hide the inherited public instance variables of a superclass? Explain.

24. If a class is immutable, why is it important that no subclasses of it be allowed to override any of its methods?

25. The java.math.BigInteger class documentation says that it represents "Immutable arbitrary-precision integers." But the class is not final nor are its methods final, contradicting the discussion in Section 5.6 regarding how to make a class immutable. What's going on here?

26. The java.util.Date class represents low-level data corresponding to a specific instance in time.

a. By the discussion in Section 5.6, it ought to be immutable. However, the class was designed as a mutable class. Come up with some guesses as to why the Java designers made Date mutable. Was it a good design choice? Discuss.

b. Create an elegant FixedDate class that has similar behavior to the Date class except that it is immutable. Ignore the deprecated methods in the Date class.

27. As discussed in Section 5.6, the main disadvantage of immutable classes is that a whole new copy needs to be made whenever you want to create an object that is slightly different from the original, even if you don't need the original any more. This can lead to a significant amount of copying if you are incrementally constructing a large immutable object. For example, if you are constructing a large string in a loop by repeatedly appending another character using the statement "`s += c`" where `s` is the string and `c` is the character, you end up creating a large number of intermediate strings, all of which eventually need to be garbage collected. Suggest a better way of constructing such a large string.

28. Suppose you have Person objects and Location objects, and you need to keep track of the place of birth of each Person. There are at least three ways to organize this information:

a. Each Person object has an instance variable of type Location that stores the place of birth.

b. Each Location keeps a collection of references to Person objects corresponding to the people born at that location.

c. A third object keeps a table of Person objects and their place-of-birth Location objects.

Discuss the advantages and disadvantages of each of these three designs.

29. Suppose the Department of Motor Vehicles (DMV) hired a software firm to design and build a vehicle registration system, and suppose that the designers initially developed a MotorVehicle class and a VehicleOwner class. Among other methods in the design, the MotorVehicle objects have a method `getOwner()` that returns a VehicleOwner, and the VehicleOwner objects have a `getVehiclesOwned()` method that returns a collection of MotorVehicles. Furthermore, since both corporations and individuals can be vehicle owners and since the designers decided they want to distinguish between them, they created a Corporation subclass of VehicleOwner. Furthermore, they added a `getEmployer()` method to the VehicleOwner class that returns a Corporation object (the employer of the vehicle owner). See Figure 5.14 for a diagram of these relationships.

The designers felt this design was sufficient to handle all situations. For example,

- The employer of a corporation can be itself (i.e., in the design, a corporation that owns a car can be thought of as being an employee of itself).
- A self-employed person will be his or her own employer, like corporations, assuming that person has incorporated himself or herself.

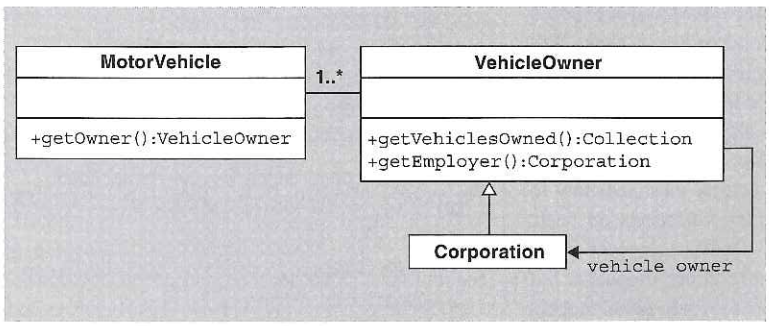

FIGURE 5.14 The relationship between vehicles and their owners.

- An unemployed owner could return null when asked for its employer.
- An unowned (e.g., abandoned) car could return null when asked for its owner.
- A person who doesn't own any cars could return an empty collection when asked for the vehicles he or she owns.
- You can easily distinguish between the two types of owners either by looking at their employers as follows:

```
if( owner.getEmployer() == owner )
   ...owner is a corporation...
else
   ...owner is a person...
```

or by testing the class of the owner as follows:

```
if( owner instanceof Corporation )
   ...owner is a corporation...
else
   ...owner is a person...
```

Analyze this design, listing all its faults, and then come up with an elegant alternate design.

30. To implement a Rectangle class which stores its position and size, you could use four integer instance variables `x`, `y`, `width`, `height` or you could use four Point instance variables that store the coordinates of the four corners of the rectangle.

a. Which of these two implementations is better and why?

b. Suppose the Rectangle has a method that allows it to be rotated by an arbitrary angle. In that case, the first implementation is insufficient, but what about the second implementation? Is it a reasonable implementation in this case?

c. Come up with a third better implementation in the case where the Rectangle can be rotated.

31. Consider the problem of designing grade book software for instructors to use to keep track of student grades on class assignments. Assume that the design includes four classes: a GradeBook class, a Student class, an Assignment class, and a Grade class. Tell how these classes should be related and what attributes and methods they should have.

32. Design a simple program for playing a poker game.

a. If you have never played poker, there are many Web sites devoted to poker that describe how to play the game and the many variations on basic poker.

b. Design your classes for one game of poker, but make it open to extension to other poker games and, more generally, other kinds of card games.

c. Include an explanation of why you designed your classes the way you did.

d. Include in your design your approach to comparing poker hands to see who is the winner.

e. You don't need to construct a GUI. In fact, you should design your code to make it easy for someone else to create various GUIs for your game, including console I/O and windows with buttons for clicking. In other words, you are to design the model for the poker game, not the view or controller. Your classes should do all the work relating to the game so that a view and controller would only need to call methods in your model.

f. Give sufficient details that one of your fellow students could easily implement the classes from your description.

g. Implement your poker game.

33. Often you have a Collection class and you want to remove or "filter" out some of the elements. For example, you might have a set of Integer objects, and you want to filter out all the odd integers, leaving only the even ones.

a. Design a way of filtering Collection classes. Your design should not require modification of the source code of existing Collection classes or interfaces. Your design should be as flexible and generally useful as possible. You should include an explanation of why you designed it the way you did.

b. Implement your design.

34. Suppose that you are using Java 1.4 or earlier and so there are no generics. In that case, if you want a list of strings, you might want to implement your own StringList class, as shown below. This class is elegant in that it allows you to work with lists of Strings, but it avoids all the downcasting that would be necessary if you were to use a general List class. What is inelegant about this implementation of the StringList class? There are several bad smells of varying importance.

```
public class StringList {
  private LinkedList v = new
    LinkedList();
  private Iterator iterator = new
    StringListIterator();

  //add a new string at the end
  public void add(String s) { v.add(s); }

  //get the i-th string
  public String get(int i){ return
    (String) v.get(i); }

  //remove and return the i-th string
```

```
public String remove(int i){ return
   (String)v.remove(i); }
//return the number of strings in
   //the list
public int size(){ return v.size(); }
//return an iterator over the list
public Iterator iterator() {
   return iterator; }
//inner class
class StringListIterator implements
   Iterator {
      int count = size();
      public boolean hasNext() { return
         count > 0; }
      public Object next() {
         if (count == 0)
            throw new
               NoSuchElementException
               ("StringList");
         return v.get(--count);
      }
      public void remove() {
         throw new
      UnsupportedOperationException();
      }
   }
}
```

35. Suppose you have created your own LinkedList class (not the java.util.LinkedList class) implemented with Nodes, each with data and next-node pointers. Furthermore, suppose you are stepping through such a list extracting the data from each node. As you step through the list, who should be responsible for keeping track of where you are in the list? There are several possibilities:
    a. You (the user of the list) keep a reference to the current node, and you repeatedly follow the current node's next pointer to move on to the next node.
    b The LinkedList object itself keeps a reference to the current node. You can ask the list for the data in the current node, and you can ask the list to move on to the next node, which causes the list to update its reference to point to the next node.
    c. A third object keeps track of where you are in the list. You ask that third object to find and return to you the data in the current node. You can also ask the third object to move on to the next node.

    Discuss these choices from the perspective of design elegance.

REFERENCES

1. Gamma, E., R. Hehn, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing. 1995. Reading, MA: Addison-Wesley.
2. Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. 1998. Upper Saddle River, NJ: Prentice Hall.
3. Wirfs-Brock, R. and A. McKean, *Object Design: Roles, Responsibilities, and Collaborations*. 2003. Reading, MA: Addison-Wesley.
4. Fowler, M., *Refactoring, Improving the Design of Existing Code*. Object Technology Series. 1999. Reading, MA: Addison-Wesley.
5. Astrachan, O., G. Berry, L. Cox, and G. Mitchener *Design Patterns: An Essential Component of CS Curricula*. in *SIGCSE Technical Symposium on Computer Science Education*. 1998. New York: ACM Press.
6. Riel, A.J., *Object-Oriented Design Heuristics*. 1996. Reading, MA: Addison-Wesley.
7. Bloch, J., *Effective Java Programming Language Guide*. The Java Series. 2001. Reading, MA: Addison-Wesley.
8. Shalloway, A. and J.R. Trott, *Design Patterns Explained*. Software Patterns Series. 2002. Reading, MA: Addison-Wesley.
9. Kernigan, B.W. and R. Pike, *The Practice of Programming*. Professional Computing Series. 1999. Reading, MA: Addison-Wesley.
10. Meyer, B., *Object-Oriented Software Construction*. 2nd ed. 1997. Upper Saddle River, NJ: Prentice-Hall.
11. Martin, R.C., *The Open Closed Principle*. C++ Report, 1996.

12. Lieberherr, K. and I. Holland, Assuring good style for object-oriented programs. *IEEE Software*, 1989. pp. 38–48.

13. Lieberherr, K. *Law of Demeter.* 1998. [Cited April 8, 2007; available from *http://www.ccs.neu.edu/home/lieber/LoD.html.*]

14. Bock, D. *The Paperboy, the Wallet, and the Law of Demeter.* Web page. [Cited April 8, 2007; available from *http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf.*]

McGraw-Hill Higher Education

OBJECT-ORIENTED DESIGN USING JAVA

Published by McGraw-Hill, a business unit of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020. Copyright © 2009 by The McGraw-Hill Companies, Inc. All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of The McGraw-Hill Companies, Inc., including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 0 DOC/DOC 0 9 8

ISBN 978–0–07–297416–4
MHID 0–07–297416–8

Global Publisher: *Raghothaman Srinivasan*
Executive Editor: *Michael Hackett*
Director of Development: *Kristine Tibbetts*
Developmental Editor: *Lora Kalb*
Executive Marketing Manager: *Michael Weitz*
Senior Project Manager: *Kay J. Brimeyer*
Senior Production Supervisor: *Laura Fuller*
Associate Design Coordinator: *Brenda A. Rolwes*
Cover Design: *Studio Montage, St. Louis, Missouri*
(USE) Cover Image: © *Comstock/PunchStock*
Compositor: *Newgen*
Typeface: *10/12 Times Roman*
Printer: *R. R. Donnelley Crawfordsville, IN*

Library of Congress Cataloging-in-Publication Data

Skrien, Dale John.
Object-oriented design using Java / Dale Skrien. — 1st ed.
p. cm.
Includes index.
ISBN 978–0–07–297416–4 — ISBN 0–07–297416–8 (hard copy : alk. paper) 1. Object-oriented programming (Computer science) 2. Java (Computer program language) I. Title.
QA76.64.S57123 2009
005.1'17–dc22

2007024087

www.mhhe.com