PREFER A TREE TO A FOREST

The last and most common design decision you'll have to make when creating composite objects is sometimes called the *forest and the trees* problem. Simply stated, should your composite objects be shallow or deep?

In a *shallow containment hierarchy* (forest), most composite objects have many fields, and the fields are composed of relatively basic types. (These basic types aren't necessarily the Java's primitive types, but could be AWT components, strings, or other "built-in" objects.) As the previous section mentioned, building a system in this way—from only a few parts—tends to promote stability.

In a *deep containment hierarchy* (tree), your system is composed of more vertical layers. This is similar to what we observe in biological systems. Your body, for instance, has a circulatory and a digestive system. Rather than be a "shallow" system, the circulatory system is, itself, broken down into subsystems: the blood, the vascular system, the heart, and the lungs. Each of these systems can be further broken down. This is a "deep" hierarchy.

The main argument in favor of deep hierarchies is that, as humans, our short-term memory can handle only a limited amount of information. (This number is generally agreed to be between three and eight discrete pieces of information.) To handle greater amounts of information, we combine related pieces together into chunks. When you create a deep hierarchy, you limit the amount of information you need to absorb at any one level, effectively increasing your processing capability.

On the other hand, as you'll see when you study inheritance in Chapters 9 and 10, deep hierarchies have their own problems, because understanding the hierarchy itself may require you to search the contained classes to learn important information about how they work. Because the interface of a class limits what you can actually do with the class, this is less often a problem with containment than it is with inheritance.

Given these tradeoffs, deep hierarchies generally have an advantage over wide, shallow hierarchies. When writing procedural programs, you might create a series of functions that help clarify a higher-level piece of code. Deep hierarchies bring the same abstraction benefit to composite objects.

Object Notation Revisited

As you saw in Chapter 5, "Designing Classes and Objects," object-oriented design methodologies grew out of the adoption of object-oriented programming languages in the 1980s and 1990s, much like structured design techniques grew out of the use of structured programming languages in the 1960s and 1970s. And, as was true with the earlier structured methodologies, several different processes, graphical notations, and heuristics were adopted and promoted by several different groups.

In late 1994, three of the most successful object-oriented methodologists—Grady Booch, Jim Rumbaugh, and Ivar Jacobson—decided to join forces and create a standard set of graphical notations for object-oriented design. The result was the *Unified Modeling Language* (UML), which is currently going through a standardization process with the Object Management Group. Although not all of the major methodologists have jumped on board, UML seems poised to become a commonly understood notation for describing object-oriented systems.

Although this book isn't a handbook on UML—we recommend Martin Fowler's excellent *UML Distilled* as an accessible introduction to the notation—knowing how to read the major UML diagrams will certainly help you communicate with other object-oriented designers, because it provides a common language.

CLASS DIAGRAMS: DESCRIBING RELATIONSHIPS