Although this design might be prudent if you were a zookeeper, the extension of the `Canine` class is probably not necessary if you are breeding and selling domesticated dogs.

So as you can see, there are always trade-offs when creating a design.

> **Making Design Decisions with the Future in Mind**
>
> You might at this point say, "Never say never." Although you might not breed yodeling dogs now, sometime in the future you might want to do so. If you do not design for the possibility of yodeling dogs now, it will be much more expensive to change the system later to include them. This is yet another of the many design decisions that you have to make. You could possibly override the `bark()` method to make it yodel; however, this is not intuitive, and some people will expect a method called `bark()` to actually bark.

Composition

It is natural to think of objects as containing other objects. A television set contains a tuner and video display. A computer contains video cards, keyboards, and drives. The computer can be considered an object unto itself, and a flash drive is also considered a valid object. You could open up the computer and remove the hard drive and hold it in your hand. In fact, you could take the hard drive to another computer and install it. The fact that it is a standalone object is reinforced because it works in multiple computers.

The classic example of object composition is the automobile. Many books, training classes, and articles seem to use the automobile as the epitome of object composition. Besides the original interchangeable manufacture of the rifle, most people think of the automobile assembly line created by Henry Ford as the quintessential example of interchangeable parts. Thus, it seems natural that the automobile has become a primary reference point for designing OO software systems.

Most people would think it natural for a car to contain an engine. However, a car contains many objects besides an engine, including wheels, a steering wheel, and a stereo. Whenever a particular object is composed of other objects, and those objects are included as object fields, the new object is known as a *compound*, an *aggregate*, or a *composite object* (see Figure 7.6).

> **Aggregation, Association, and Composition**
>
> From my perspective, there are only two ways to reuse classes—with inheritance or composition. In Chapter 9, "Building Objects and Object-Oriented Design," we discuss composition in more detail—specifically, aggregation and association. In this book, I consider aggregation and association to be types of composition, although there are varied opinions on this.

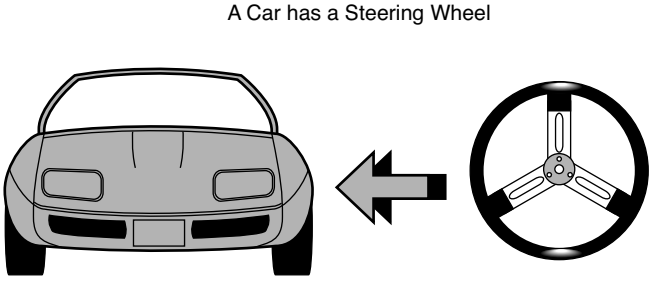

Figure 7.6 An example of composition.

Representing Composition with UML

To model the fact that the car object contains a steering wheel object, UML uses the notation shown in Figure 7.7.

> **Aggregation, Association, and UML**
>
> In this book, aggregations are represented in UML by lines with a diamond, such as an engine as part of a car. Associations are represented by just the line (no diamond), such as a stand-alone keyboard servicing a separate computer box.

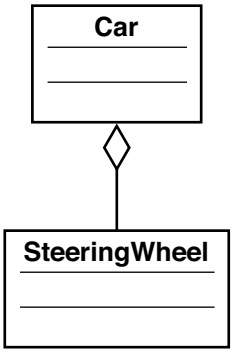

Figure 7.7 Representing composition in UML.

Note that the line connecting the `Car` class to the `SteeringWheel` class has a diamond shape on the `Car` side of the line. This signifies that a `Car` *contains* (has-a) `SteeringWheel`.

Let's expand this example. Suppose that none of the objects in this design use inheritance in any way. All the object relationships are strictly composition, and there are multiple levels of composition. Of course, this is a simplistic example, and there are many, many more object and object relationships in designing a car. However, this design is meant to be a simple illustration of what composition is all about.

Let's say that a car is composed of an engine, a stereo system, and a door.

> **How Many Doors and Stereos?**
>
> Note that a car normally has more than one door. Some have two, and some have four. You might even consider a hatchback a fifth door. In the same vein, it is not necessarily true that all cars have a stereo system. A car could have no stereo system, or it could have one. I have even seen a car with two separate stereo systems. These situations are discussed in detail in Chapter 9. For the sake of this example, just pretend that a car has only a single door (perhaps it's a special racing car) and a single stereo system.

That a car is made up of an engine, a stereo system, and a door is easy to understand because most people think of cars in this way. However, it is important to keep in mind when designing software systems, just like automobiles, that objects are made up of other objects. In fact, the number of nodes and branches that can be included in this tree structure of classes is virtually unlimited.

Figure 7.8 shows the object model for the car, with the engine, stereo system, and door included.

Note that all three objects that make up a car are themselves composed of other objects. The engine contains pistons and spark plugs. The stereo contains a radio and a CD player. The door contains a handle. Also note that there is yet another level. The radio contains a tuner. We could have also added the fact that a handle contains a lock; the CD player contains a fast forward button, and so on. Additionally, we could have gone one level beyond the tuner and created an object for a dial. The level and complexity of the object model is up to the designer.

> **Model Complexity**
>
> As with the inheritance problem of the barking and yodeling dogs, using too much composition can also lead to more complexity. A fine line exists between creating an object model that contains enough granularity to be sufficiently expressive and a model that is so granular that it is difficult to understand and maintain.

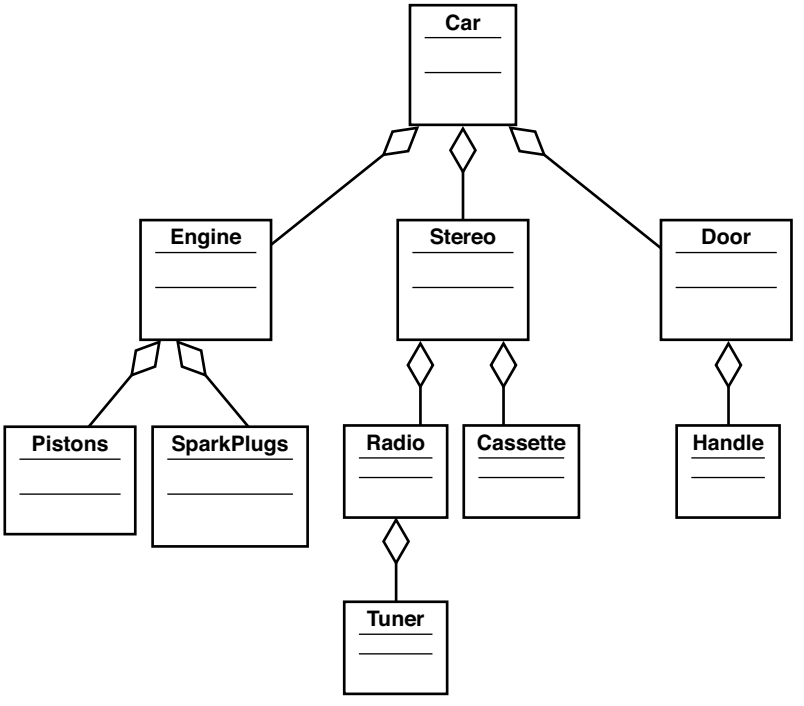

Figure 7.8 The `Car` class hierarchy.

Why Encapsulation Is Fundamental to OO

Encapsulation is the fundamental concept of OO. Whenever the interface/implementation paradigm is covered, we are talking about encapsulation. The basic question is what in a class should be exposed and what should not be exposed. This encapsulation pertains equally to data and behavior. When talking about a class, the primary design decision revolves around encapsulating both the data and the behavior into a well-written class.

Stephen Gilbert and Bill McCarty define encapsulation as "the process of packaging your program, dividing each of its classes into two distinct parts: the interface and the implementation." This is the message that has been presented over and over in this book.

But what does encapsulation have to do with inheritance, and how does it apply with regard to this chapter? This has to do with an OO paradox. Encapsulation is so fundamental to OO that it is one of OO design's cardinal rules. Inheritance is also considered one of the three primary OO concepts. However, in one way, inheritance actually breaks encapsulation! How can this be? Is it possible that two of the three primary concepts of OO are incompatible with each other? Let's explore this possibility.

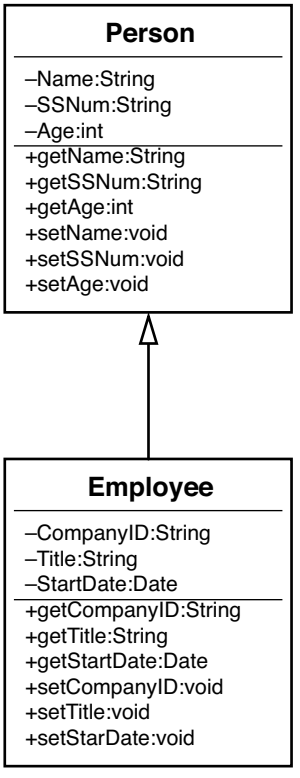

Figure 9.1 An inheritance relationship.

Composition Relationships

We have already seen that composition represents a part of a whole. Although the inheritance relationship is stated in terms of is-a, composition is stated in terms of has-a. We know intuitively that a car "has-a" steering wheel (see Figure 9.2).

> **Is-a and Has-a**
>
> Please forgive my grammar: For consistency, I will stick with "has a engine," even though "has an engine" is grammatically correct. I do this because I want to simply state the rules as "is-a" and "has-a."

The reason to use composition is that it builds systems by combining less complex parts. This is a common way for people to approach problems. Studies show that even the best of us can keep, at most, seven chunks of data in our short-term memory at one time. Thus, we like to use abstract concepts. Instead of saying that we have a large unit with a steering wheel, four tires, an engine, and so on, we say that we have a car. This makes it easier for us to communicate and keep things clear in our heads.

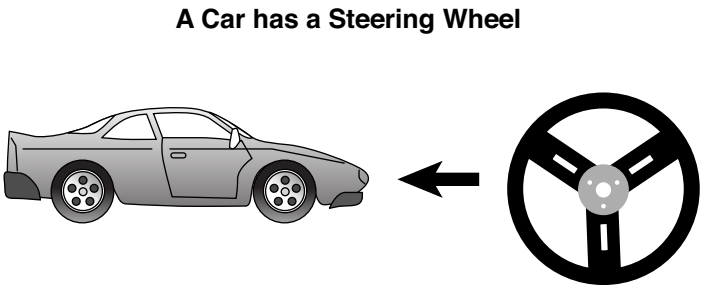

Figure 9.2 A composition relationship.

Composition also helps in other ways, such as making parts interchangeable. If all steering wheels are the same, it does not matter which specific steering wheel is installed in a specific car. In software development, interchangeable parts mean reuse.

In Chapters 7 and 8 of their book *Object-Oriented Design in Java*, Stephen Gilbert and Bill McCarty present many examples of associations and composition in much more detail. I highly recommend referencing this material for a more in-depth look into these subjects. Here we address some of the more fundamental points of these concepts and explore some variations of their examples.

Building in Phases

Another major advantage in using composition is that systems and subsystems can be built independently, and perhaps more importantly, tested and maintained independently.

There is no question that software systems are quite complex. To build quality software, you must follow one overriding rule to be successful: keep things as simple as possible. For large software systems to work properly and be easily maintained, they must be broken into smaller, more manageable parts. How do you accomplish this? In a 1962 article titled "The Architecture of Complexity," Nobel Prize winner Herbert Simon noted the following thoughts regarding stable systems:

- **"Stable complex systems usually take the form of a hierarchy, where each system is built from simpler subsystems, and each subsystem is built from simpler subsystems still."**—You might already be familiar with this principle because it forms the basis for functional decomposition, the method behind procedural software development. In object-oriented design, you apply the same principles to composition—building complex objects from simpler pieces.
- **"Stable, complex systems are nearly decomposable."**—This means you can identify the parts that make up the system and can tell the difference between interactions between

the parts and inside the parts. Stable systems have fewer links between their parts than they have inside their parts. Thus, a modular stereo system, with simple links between the speakers, turntable, and amplifier, is inherently more stable than an integrated system, which isn't easily decomposable.

- **"Stable complex systems are almost always composed of only a few different kinds of subsystems, arranged in different combinations."**—Those subsystems, in turn, are generally composed of only a few different kinds of parts.
- **"Stable systems that work have almost always evolved from simple systems that worked."**—Rather than build a new system from scratch—reinventing the wheel—the new system builds on the proven designs that went before it.

In our stereo example (see Figure 9.3), suppose the stereo system was totally integrated and was not built from components (that is, the stereo system was one big black-box system). In this case, what would happen if the CD player broke and became unusable? You would have to take in the entire system for repair. Not only would this be more complicated and expensive, but you would not have the use of any of the other components.

This concept becomes very important to languages such as Java and those included in the .NET framework. Because objects are dynamically loaded, decoupling the design is quite important. For example, if you distribute a Java application and one of the class files needs to be re-created (for bug fixes or maintenance), you would be required to redistribute only that particular class file. If all code was in a single file, the entire application would need to be redistributed.

Suppose the system is broken into components rather than a single unit. In this case, if the CD player broke, you could disconnect the CD player and take it in for repair. (Note that all the components are connected by patch cords.) This would be less complicated and less expensive, and it would take less time than having to deal with a single, integrated unit. As an added benefit, you could still use the rest of the system. You could even buy another CD player because it is a component. The repairperson could then plug your broken CD player into his repair systems to test and fix it. All in all, the component approach works quite well. Composition is one of the primary strategies that you, as a software designer, have in your arsenal to fight software complexity.

One major advantage of using components is that you can use components that were built by other developers, or even third-party vendors. However, using a software component from another source requires a certain amount of trust. Third-party components must come from a reliable source, and you must feel comfortable that the software is properly tested, not to mention that it must perform the advertised functions properly. There are still many who would rather build their own than trust components built by others.

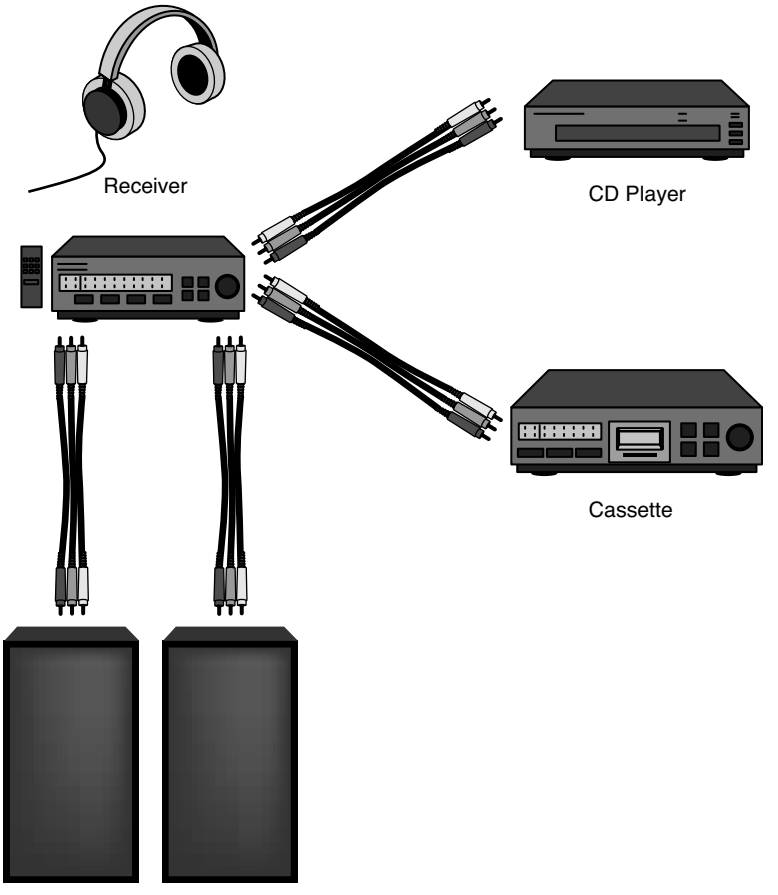

Figure 9.3 Building, testing, and verifying a complete system one step at a time.

Types of Composition

Generally, there are two types of composition: association and aggregation. In both cases, these relationships represent collaborations between the objects. The stereo example we just used to explain one of the primary advantages of composition represents an association.

Is Composition a Form of Association?

Composition is another area in OO technologies where there is a question of which came first, the chicken or the egg. Some texts say that composition is a form of association, and some say that an association is a form of composition. In any event, in this book, we consider inheritance and composition the two primary ways to build classes. Thus, in this book, association is considered a form of composition.

All forms of composition include a has-a relationship. However, subtle differences exist between associations and aggregations based on how you visualize the parts of the whole. In an aggregation, you normally see only the whole, and in associations, you normally see the parts that make up the whole.

Aggregations

Perhaps the most intuitive form of composition is aggregation. Aggregation means that a complex object is composed of other objects. A TV set is a clean, neat package that you use for entertainment. When you look at your TV, you see a single TV. Most of the time, you do not stop to think about the fact that the TV contains some microchips, a screen, a tuner, and so on. Sure, you see a switch to turn the set on and off, and you certainly see the picture screen. However, this is not the way people normally think of TVs. When you go into an appliance store, the salesperson does not say, "Let me show you this aggregation of microchips, a picture screen, a tuner, and so on." The salesperson says, "Let me show you this TV."

Similarly, when you go to buy a car, you do not pick and choose all the individual components of the car. You do not decide which spark plugs to buy or which door handles to buy. You go to buy a car. Of course, you do choose some options, but for the most part, you choose the car as a whole, a complex object made up of many other complex and simple objects (see Figure 9.4).

Associations

Whereas aggregations represent relationships where you normally see only the whole, associations present both the whole and the parts. As stated in the stereo example, the various components are presented separately and connect to the whole by use of patch cords (the cords that connect the various components).

Consider a computer system as an example (see Figure 9.5); the whole is the computer system. The components are the monitor, keyboard, mouse, and main box. Each is a separate object, but together they represent the whole of the computer system. The main computer is using the keyboard, the mouse, and the monitor to delegate some of the work. In other words, the computer box needs the service of a mouse, but does not have the capability to provide this service by itself. Thus, the computer box requests the service from a separate mouse via the specific port and cable connecting the mouse to the box.

> **Aggregation Versus Association**
>
> An aggregation is a complex object composed of other objects. An association is used when one object wants another object to perform a service for it.

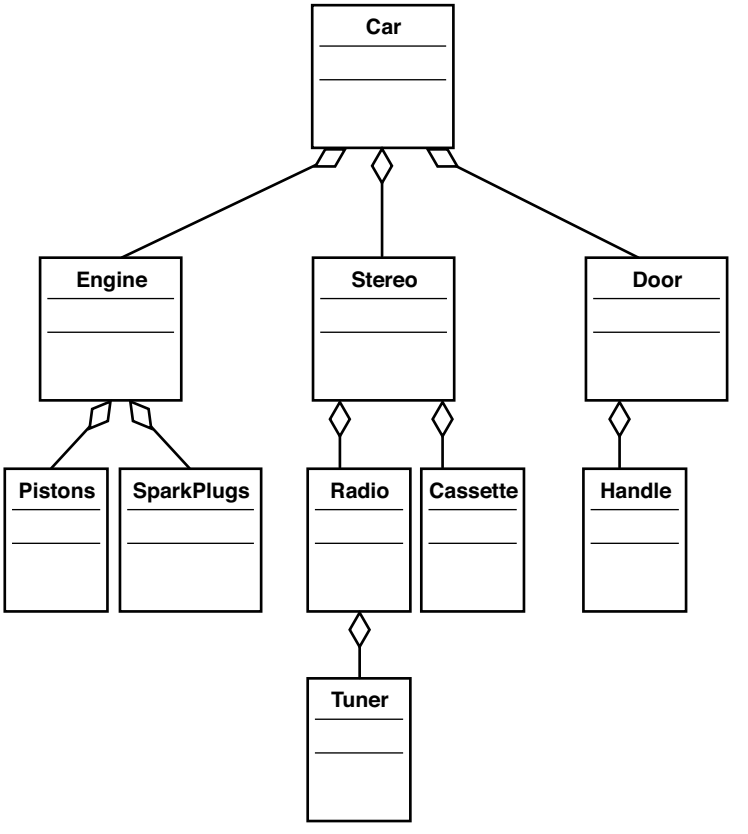

Figure 9.4 An aggregation hierarchy for a car.

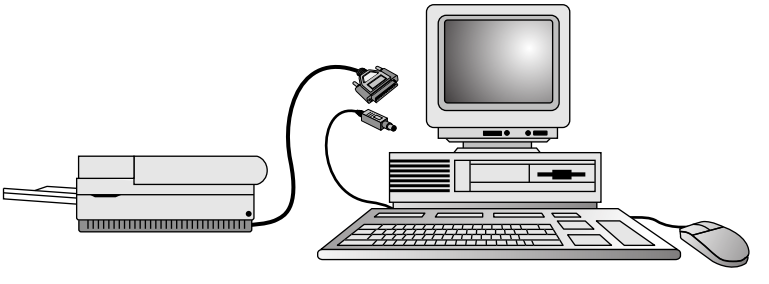

Figure 9.5 Associations as a separate service.

Using Associations and Aggregations Together

One thing you might have noticed in all the examples is that the dividing lines between what is an association and what is an aggregation are often blurred. Suffice it to say that many of your most interesting design decisions will come down to whether to use associations or aggregations.

For example, the computer system example used to describe associations also contains some aggregation. Although the interaction between the computer box, the monitor, the keyboard, and the mouse is association, the computer box itself represents aggregation. You see only the computer box, but it is actually a complex system made up of other objects, including chips, motherboards, video cards, and so on.

Consider that an `Employee` object might be composed of an `Address` object and a `Spouse` object. You might consider the `Address` object as an aggregation (basically a part of the `Employee` object), and the `Spouse` object as an association. To illustrate, suppose both the employee and the spouse are employees. If the employee is fired, the spouse is still in the system, but the association is broken.

Similarly, in the stereo example, the receiver has an association with the speakers as well as the CD. Yet, the speakers and the CD are themselves aggregations of other objects, such as power chords.

In the car example, although the engine, spark plugs, and doors represent composition, the stereo also represents an association relationship.

> **No One Right Answer**
>
> As usual, there isn't a single, absolutely correct answer when it comes to making a design decision. Design is not an exact science. Although we can make general rules to live by, these rules are not hard and fast.

Avoiding Dependencies

When using composition, it is desirable to avoid making objects highly dependent on one another. One way to make objects very dependent on each other is to mix domains. In the best of all worlds, an object in one domain should not be mixed with an object in another domain, except under certain circumstances. We can return again to the stereo example to explain this concept.

By keeping the receiver and the CD player in separate domains, the stereo system is easier to maintain. For example, if the CD component breaks, you can send the CD player off to be repaired individually. In this case, the CD player and the MP3 player have separate domains. This provides flexibility, such as buying the CD player and the MP3 player from separate manufacturers. So, if you decide you want to swap out the CD player with a brand from another manufacturer, you can.

Sometimes there is a certain convenience in mixing domains. A good example of this pertains to the existence of TV/VCR and TV/DVD combinations. Not only has object-oriented design

evolved since the first edition of this book was published, but consumer technology has evolved as well. VCRs are not as prevalent as they once were, and DVDs are following the same path. The movement of consumer preferences from VCRs to DVDs to streaming technologies in such a short period of time is a good example of why avoiding dependencies is a preferred design choice at many levels.

You need to determine what is more important in specific situations: whether you want convenience or stability. There is no right answer. It all depends on the application and the environment. In the case of the TV/VCR combination, we decided that the convenience of the integrated unit far outweighed the risk of lower unit stability (see Figure 9.6).

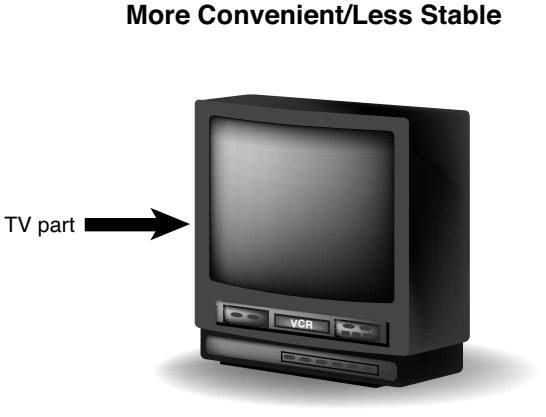

Figure 9.6 Convenience versus stability.

> **Mixing Domains**
>
> The convenience of mixing domains is a design decision. If the power of having a TV/VCR combination outweighs the risk and potential downtime of the individual components, the mixing of domains may well be the preferred design choice.

Cardinality

In their book *Object-Oriented Design in Java*, Gilbert and McCarty describe cardinality as the number of objects that participate in an association and whether the participation is optional or mandatory. To determine cardinality, Gilbert and McCarty ask the following questions:

- Which objects collaborate with which other objects?
- How many objects participate in each collaboration?
- Is the collaboration optional or mandatory?