7

Mastering Inheritance and Composition

Inheritance and composition play major roles in the design of object-oriented (OO) systems. In fact, many of the most difficult and interesting design decisions come down to deciding between inheritance and composition.

These decisions have become much more interesting over the years as object-oriented design practices have evolved. Perhaps one of the most interesting debates revolves around inheritance. Although inheritance is one of the fundamental constructs of object-oriented development (a language must support inheritance to be considered object-oriented), some developers are even turning away from inheritance by implementing designs solely with composition.

It is common to use interface inheritance rather than direct inheritance for behaviors (implementing versus inheriting). Inheritance tends to be used often for data/models whereas implementation tends to be used for behaviors.

Regardless, both inheritance and composition are mechanisms for reuse. *Inheritance*, as its name implies, involves inheriting attributes and behaviors from other classes, where there is a true parent/child relationship. The child (or subclass) inherits directly from the parent (or superclass).

Composition, also as its name implies, involves building objects by using other objects. In this chapter we explore the obvious and subtle differences between inheritance and composition. Primarily, we will consider the appropriate times to use one or the other.

Reusing Objects

Perhaps the primary reason why inheritance and composition exist is object reuse. In short, you can build classes (which ultimately become objects) by utilizing other classes via inheritance and composition, which in effect are the only ways to reuse previously built classes.

Inheritance represents the is-a relationship that was introduced in Chapter 1, "Introduction to Object-Oriented Concepts." For example, a dog *is a* mammal.

Composition involves using other classes to build more complex classes—a sort of assembly. No parent/child relationship exists in this case. Basically, complex objects are composed of other objects. Composition represents a has-a relationship. For example, a car *has an* engine. Both the engine and the car are separate, potentially standalone objects. However, the car is a complex object that contains (has an) engine object. In fact, a child object might itself be composed of other objects; for example, the engine might include cylinders. In this case an engine *has a* cylinder, actually several.

When OO technologies first entered the mainstream, inheritance was often the first example used in how to design an OO system. That you could design a class once and then inherit functionality from it was considered one of the foremost advantages to using OO technologies. Reuse was the name of the game, and inheritance was the ultimate expression of reuse.

However, over time the luster of inheritance has dulled a bit. In fact, in some discussions, the use of inheritance itself is questioned. In their book *Java Design*, Peter Coad and Mark Mayfield have a complete chapter titled "Design with Composition Rather Than Inheritance." Many early object-based platforms did not even support true inheritance. As Visual Basic evolved into Visual Basic .NET, early object-based implementations did not include strict inheritance capabilities. Platforms such as the MS COM model were based on interface inheritance. Interface inheritance is covered in great detail in Chapter 8, "Frameworks and Reuse: Designing with Interfaces and Abstract Classes."

Today, the use of inheritance is still a major topic of debate. Abstract classes, which are a form of inheritance, are not directly supported in some languages, such as Objective-C and Swift. Interfaces are used even though they don't provide all the functionality that abstract classes do.

The good news is that the discussions about whether to use inheritance or composition are a natural progression toward some seasoned middle ground. As in all philosophical debates, there are passionate arguments on both sides. Fortunately, as is normally the case, these heated discussions have led to a more sensible understanding of how to utilize the technologies.

We will see later in this chapter why some people believe that inheritance should be avoided, and composition should be the design method of choice. The argument is fairly complex and subtle. In actuality, both inheritance and composition are valid class design techniques, and they each have a proper place in the OO developer's toolkit. And, at least, you need to understand both to make the proper design choice—not to mention maintenance of legacy code.

The fact that inheritance is often misused and overused is more a result of a lack of understanding of what inheritance is all about than a fundamental flaw in using inheritance as a design strategy.

The bottom line is that inheritance and composition are both important techniques in building OO systems. Designers and developers need to take the time to understand the strengths and weaknesses of both and to use each in the proper contexts.

Inheritance

Inheritance was defined in Chapter 1 as a system in which child classes inherit attributes and behaviors from a parent class. However, there is more to inheritance, and in this chapter we explore inheritance in greater detail.

Chapter 1 states that you can determine an inheritance relationship by following a simple rule: If you can say that Class B *is a* Class A, then this relationship is a good candidate for inheritance.

> **Is-a**
>
> One of the primary rules of OO design is that public inheritance is represented by an *is-a* relationship. In the case of interfaces you might add "behaves like" (implements). The data (attributes) that are inherited are the "is," the interfaces describing encapsulated behaviors are "acts like," and composition is "has a." The lines get pretty blurry, however.

Let's revisit the mammal example used in Chapter 1. Let's consider a `Dog` class. A dog has several behaviors that make it distinctly a dog, as opposed to a cat. For this example, let's specify two: A dog barks and a dog pants. So we can create a `Dog` class that has these two behaviors, along with two attributes (see Figure 7.1).

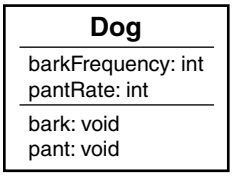

Figure 7.1 A class diagram for the `Dog` class.

Now let's say that you want to create a `GoldenRetriever` class. You could create a brand-new class that contains the same behaviors that the `Dog` class has. However, we could make the following, and quite reasonable, conclusion: A Golden Retriever is-a dog. Because of this relationship, we can inherit the attributes and behaviors from `Dog` and use it in our new `GoldenRetriever` class (see Figure 7.2).

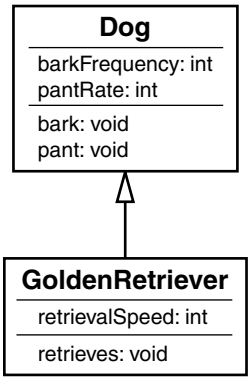

Figure 7.2 The `GoldenRetriever` class inherits from the `Dog` class.

The GoldenRetriever class now contains its own behaviors as well as all the more general behaviors of a dog. This provides us with some significant benefits. First, when we wrote the GoldenRetriever class, we did not have to reinvent part of the wheel by rewriting the bark and pant methods. Not only does this save some design and coding time, but it saves testing and maintenance time as well. The bark and pant methods are written only once and, assuming that they were properly tested when the Dog class was written, they do not need to be heavily tested again; but it does need to be retested because there are new interfaces, and so on.

Now let's take full advantage of our inheritance structure and create a second class under the Dog class: a class called LhasaApso. Whereas retrievers were bred for retrieving, Lhasa Apsos were bred for use as guard dogs. These dogs are not attack dogs; they have acute senses, and when they sense something unusual, they start barking. So we can create our LhasaApso class and inherit from the Dog class just as we did with the GoldenRetriever class (see Figure 7.3).

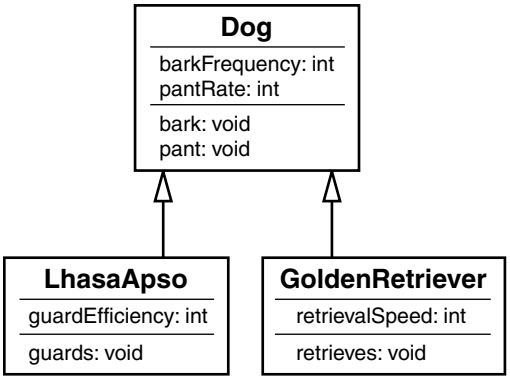

Figure 7.3 The LhasaApso class inherits from the Dog class.

Testing New Code

In our example with the GoldenRetriever class, the bark and pant methods should be written, tested, and debugged when the Dog class is written. Theoretically, this code is now robust and ready to reuse in other situations. However, the fact that you do not need to rewrite the code does not mean it should not be tested. However unlikely, there might be some specific characteristic of a retriever that somehow breaks the code. The bottom line is that you should always test new code. Each new inheritance relationship creates a new context for using inherited methods. A complete testing strategy should take into account each of these contexts.

Another primary advantage of inheritance is that the code for bark() and pant() is in a single place. Let's say there is a need to change the code in the bark() method. When you change it in the Dog class, you do not need to change it in the LhasaApso class and the GoldenRetriever class.

Do you see a problem here? At this level the inheritance model appears to work very well. However, can you be certain that all dogs have the behavior contained in the Dog class?

In his book *Effective C++*, Scott Meyers gives a great example of a dilemma with design using inheritance. Consider a class for a bird. One of the most recognizable characteristics of a bird is, of course, that it can fly. So we create a class called `Bird` with a `fly` method. You should immediately understand the problem. What do we do with a penguin, or an ostrich? They are birds, yet they can't fly. You could override the behavior locally, but the method would still be called `fly`. And it would not make sense to have a method called `fly` for a bird that does not fly but only waddles, runs, or swims. This is an example of the Liskov Substitution Principle of SOLID, which we discuss in Chapter 12, "The SOLID Principles of Object-Oriented Design."

This leads to some potentially significant problems. For example, if a penguin has a `fly` method, the penguin might understandably decide to test it out. However, if the `fly` method was in fact overridden and the behavior to fly did not exist, the penguin would be in for a major surprise when the `fly` method is invoked after jumping over a cliff. Imagine the penguin's chagrin when the call to the `fly` method results in waddling instead of flight (or even a no-op, which means no operation, where nothing happens at all). In this situation, waddling doesn't cut it. Just imagine if code such as this ever found its way into a spacecraft's guidance system.

In our dog example, we have designed the class so that all dogs have the ability to bark. However, some dogs do not bark. The Basenji breed is a barkless dog. Although these dogs do not bark, they do yodel. So should we reevaluate our design? What would this design look like? Figure 7.4 is an example that shows a more correct way to model the hierarchy of the `Dog` class.

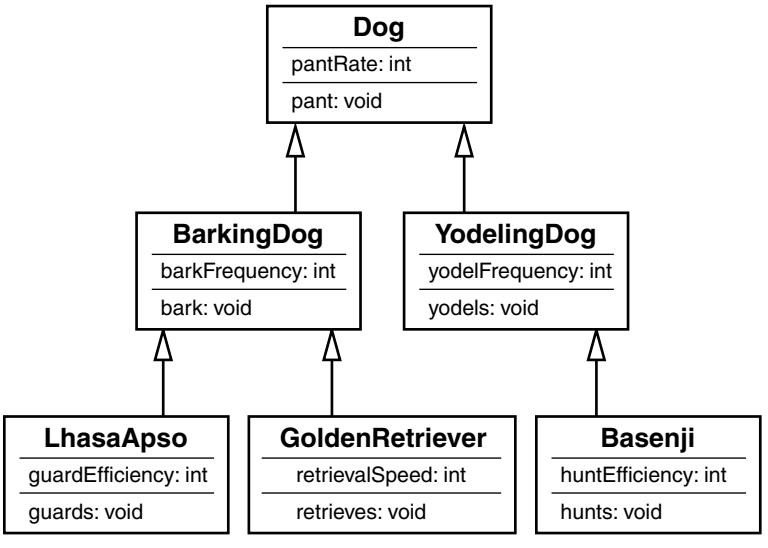

Figure 7.4 The `Dog` class hierarchy.

Generalization and Specialization

Consider the object model of the `Dog` class hierarchy. We started with a single class, called `Dog`, and we factored out some of the commonality between various breeds of dogs. This concept, sometimes called *generalization-specialization*, is yet another important consideration when using

inheritance. The idea is that as you make your way down the inheritance tree, things get more specific. The most general case is at the top of the tree. In our `Dog` inheritance tree, the class `Dog` is at the top and is the most general category. The various breeds—the `GoldenRetriever`, `LhasaApso`, and `Basenji` classes—are the most specific. The idea of inheritance is to go from the general to the specific by factoring out commonality.

In the `Dog` inheritance model, we started factoring out common behavior by understanding that although a retriever has some different behavior from that of a `LhasaApso`, the breeds do share some common behaviors—for example, they both pant and bark. Then we realized that all dogs do not bark—some yodel. Thus, we had to factor out the barking behavior into a separate `BarkingDog` class. The yodeling behavior went into a `YodelingDog` class. However, we realized that both barking dogs and barkless dogs still shared some common behavior—all dogs pant. Thus, we kept the `Dog` class and had the `BarkingDog` and the `YodelingDog` classes inherit from `Dog`. Now `Basenji` can inherit from `YodelingDog`, and `LhasaApso` and `GoldenRetriever` can inherit from `BarkingDog`.

We could have decided not to create two distinct classes for `BarkingDog` and `YodelingDog`. In this case we could implement all barking and yodeling as part of each individual breed's class—since each dog would sound differently. This is just one example of some of the design decisions that have to be made. Perhaps the best solution is to implement the barking and yodeling as interfaces, which we discuss in Chapter 8.

A design pattern, which is covered in Chapter 10, "Design Patterns," might be a good option in this case. A developer might not typically create these variations of `Dog`; they would either use a `Dog` (which implements `IDog`) or use a decorator to add behaviors to a `Dog` object.

Design Decisions

In theory, factoring out as much commonality as possible is great. However, as in all design issues, sometimes it really is too much of a good thing. Although factoring out as much commonality as possible might represent real life as closely as possible, it might not represent your model as closely as possible. The more you factor out, the more complex your system gets. So you have a conundrum: Do you want to live with a more accurate model or a system with less complexity? You must make this choice based on your situation, for there are no hard guidelines to make the decision.

> **What Computers Are Not Good At**
>
> Obviously, a computer model can only approximate real-world situations. Computers are good at number crunching but are not as good at more abstract operations.

For example, breaking up the `Dog` class into `BarkingDog` and the `YodelingDog` models real life better than assuming that all dogs bark, but it does add a bit of complexity.

Model Complexity

At this level of our example, adding two more classes does not make things so complex that it makes the model untenable. However, in larger systems, when these kinds of decisions are made over and over, the complexity quickly adds up. In larger systems, keeping things as simple as possible is usually the best practice.

There will be instances in your design when the advantage of a more accurate model does not warrant the additional complexity. Let's assume that you are a dog breeder and that you contract out for a system that tracks all your dogs. The system model that includes barking dogs and yodeling dogs works fine. However, suppose that you do not breed any yodeling dogs—never have and never will. Perhaps you do not need to include the complexity of differentiating between yodeling dogs and barking dogs. This will make your system less complex, and it will provide the functionality that you need.

Deciding whether to design for less complexity or more functionality is a balancing act. The primary goal is always to build a system that is flexible without adding so much complexity that the system collapses under its own weight. What happens if you need to add yodeling at a later point in the project?

Current and future costs are also a major factor in these decisions. Although it might seem appropriate to make a system more complete and flexible, this added functionality might barely add any benefit—the return on investment might not be there. For example, would you extend the design of your `Dog` system to include other canines, such as hyenas and foxes (see Figure 7.5)?

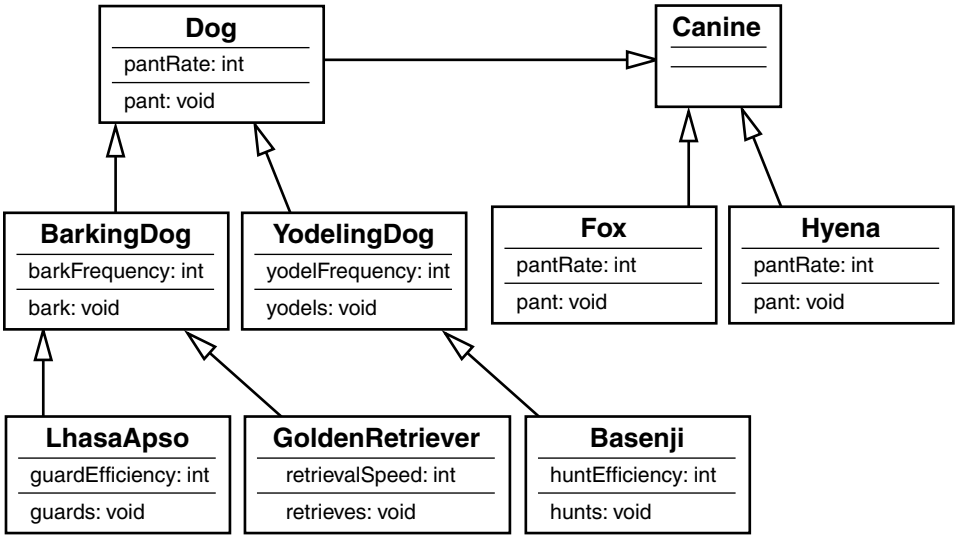

Figure 7.5 An expanded canine model.

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 classic example 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 with 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.

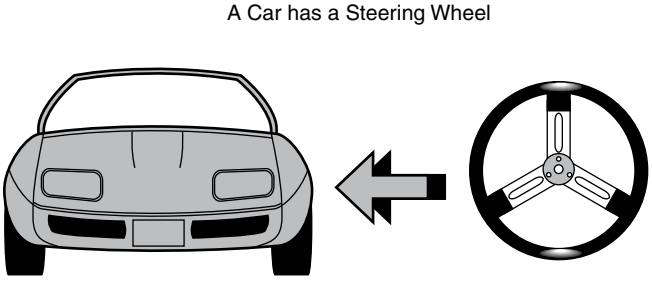

Figure 7.6 An example of composition.

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.

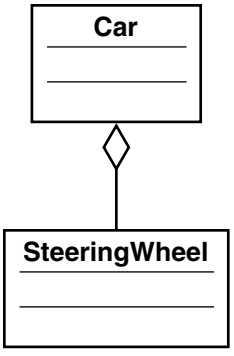

Figure 7.7 Representing composition in UML.

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.

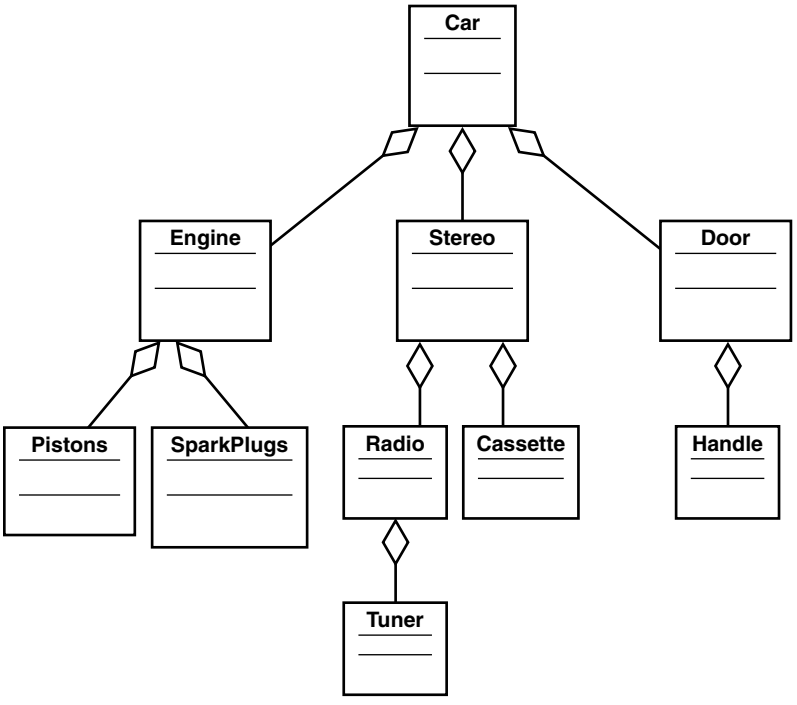

Figure 7.8 The `Car` class hierarchy.

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.

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.

How Inheritance Weakens Encapsulation

As already stated, encapsulation is the process of packaging classes into the public interface and the private implementation. In essence, a class hides everything that is not necessary for other classes to know about.

Peter Coad and Mark Mayfield make a case that when using inheritance, encapsulation is inherently weakened within a class hierarchy. They talk about a specific risk: Inheritance connotes strong encapsulation with other classes but weak encapsulation between a superclass and its subclasses.

The problem is that if you inherit an implementation from a superclass and then change that implementation, the change from the superclass *ripples through* the class hierarchy. This rippling effect potentially affects all the subclasses. At first, this might not seem like a major problem; however, as we have seen, a rippling effect such as this can cause unanticipated problems. For example, testing can become a nightmare. In Chapter 6, "Designing with Objects," we talked about how encapsulation makes testing systems easier. In theory, if you create a class called `Cabbie` (see Figure 7.9) with the appropriate public interfaces, any change to the implementation of `Cabbie` should be transparent to all other classes. However, in any design a change to a superclass is certainly not transparent to a subclass. Do you see the conundrum?

If the other classes were directly dependent on the implementation of the `Cabbie` class, testing would become more difficult, if not untenable. By using a different design approach, by abstracting out the behaviors and inheriting only attributes, these issues noted above go away.

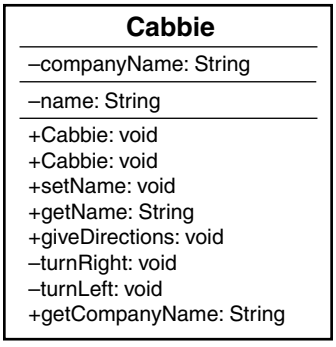

Figure 7.9 A UML diagram of the `Cabbie` class.

> **Keep Testing**
>
> Even with encapsulation, you would still want to retest the classes that use `Cabbie` to verify that no problem has been introduced by the change.

If you then create a subclass of `Cabbie` called `PartTimeCabbie`, and `PartTimeCabbie` inherits the implementation from `Cabbie`, changing the implementation of `Cabbie` directly affects the `PartTimeCabbie` class.

For example, consider the UML diagram in Figure 7.10. `PartTimeCabbie` is a subclass of `Cabbie`. Thus, `PartTimeCabbie` inherits the public implementation of `Cabbie`, including the method `giveDirections()`. If the method `giveDirections()` is changed in `Cabbie`, it will have a direct impact on `PartTimeCabbie` and any other classes that might later be subclasses of `Cabbie`. In this subtle way, changes to the implementation of `Cabbie` are not necessarily encapsulated within the `Cabbie` class.

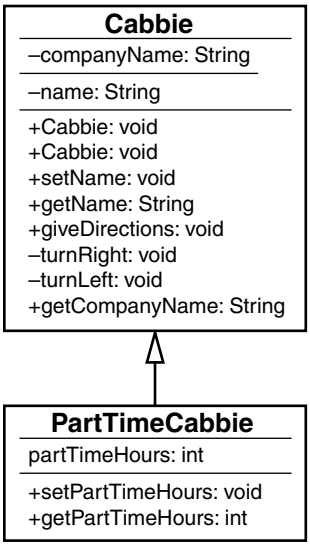

Figure 7.10 A UML diagram of the `Cabbie`/`PartTimeCabbie` classes.

To reduce the risk posed by this dilemma, it is important that you stick to the strict is-a condition when using inheritance. If the subclass were truly a specialization of the superclass, changes to the parent would likely affect the child in ways that are natural and expected. To illustrate, if a `Circle` class inherits implementation from a `Shape` class, and a change to the implementation of `Shape` breaks `Circle`, then `Circle` was not truly a `Shape` to begin with.

How can inheritance be used improperly? Consider a situation in which you want to create a window for the purposes of a graphical user interface (GUI). One impulse might be to create a window by making it a subclass of a rectangle class:

```
public class Rectangle {

}

public class Window extends Rectangle {

}
```

In reality a GUI window is much, much more than a rectangle. It is not a specialized version of a rectangle, as is a square. A true window might contain a rectangle (in fact, many rectangles); however, it is not a true rectangle. In this approach, a `Window` class should not inherit from `Rectangle`, but it should contain `Rectangle` classes.

```
public class Window {

    Rectangle menubar;
    Rectangle statusbar;
    Rectangle mainview;

}
```

A Detailed Example of Polymorphism

Many people consider polymorphism a cornerstone of OO design. Designing a class for the purpose of creating totally independent objects is what OO is all about. In a well-designed system, an object should be able to answer all the important questions about it. As a rule, an object should be responsible for itself. This independence is one of the primary mechanisms of code reuse.

As stated in Chapter 1, polymorphism literally means *many shapes*. When a message is sent to an object, the object must have a method defined to respond to that message. In an inheritance hierarchy, all subclasses inherit the interfaces from their superclass. However, because each subclass is a separate entity, each might require a separate response to the same message.

To review the example in Chapter 1, consider a class called `Shape`. This class has a behavior called `Draw`. However, when you tell somebody to draw a shape, the first question is likely to be, "What shape?" Simply telling a person to draw a shape is too abstract (in fact, the `Draw` method in `Shape` contains no implementation). You must specify which shape you mean. To do this, you provide the actual implementation in `Circle` and other subclasses. Even though `Shape` has a `Draw` method, `Circle` overrides this method and provides its own `Draw` method. Overriding basically means replacing an implementation of a parent with your own.

Object Responsibility

Let's revisit the Shape example from Chapter 1 (see Figure 7.11).

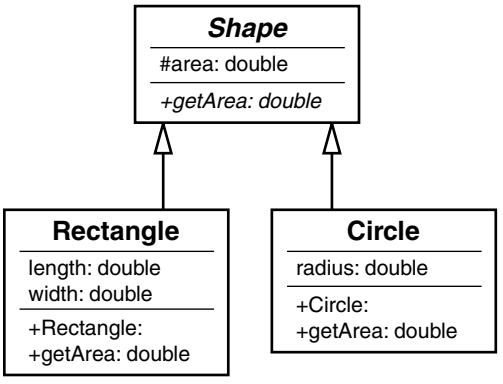

Figure 7.11 The Shape class hierarchy.

Polymorphism is one of the most elegant uses of inheritance. Remember that a Shape cannot be instantiated. It is an abstract class because it has an abstract method, `getArea()`. Chapter 8 explains abstract classes in great detail.

However, `Rectangle` and `Circle` can be instantiated because they are concrete classes. Although `Rectangle` and `Circle` are both shapes, they have some differences. As shapes, their area can be calculated. Yet the formula to calculate the area is different for each. Thus, the area formulas cannot be placed in the Shape class.

This is where polymorphism comes in. The premise of polymorphism is that you can send messages to various objects, and they will respond according to their object's type. For example, if you send the message `getArea()` to a `Circle` class, you will invoke a different calculation than if you send the same `getArea()` message to a `Rectangle` class. This is because both `Circle` and `Rectangle` are responsible for themselves. If you ask `Circle` to return its area, it knows how to do this. If you want a circle to draw itself, it can do this as well. A Shape object could not do this even if it could be instantiated because it does not have enough information about itself. Notice that in the UML diagram (Figure 7.11), the `getArea()` method in the Shape class is italicized. This designates that the method is abstract.

As a very simple example, imagine that there are four classes: the abstract class Shape, and concrete classes `Circle`, `Rectangle`, and `Star`. Here is the code:

```
public abstract class Shape{

    public abstract void draw();

}
```

```
public class Circle extends Shape{

    public void draw() {

        System.out.println("I am drawing a Circle");

    }
}

public class Rectangle extends Shape{

    public void draw() {

        System.out.println("I am drawing a Rectangle");

    }
}

public class Star extends Shape{

    public void draw() {

        System.out.println("I am drawing a Star");

    }
}
```

Notice that only one method exists for each class: `draw()`. Here is the important point regarding polymorphism and an object being responsible for itself: The concrete classes themselves have responsibility for the drawing function. The `Shape` class does not provide the code for drawing; the `Circle`, `Rectangle`, and `Star` classes do this for themselves. Here is some code to prove it:

```
public class TestShape {

    public static void main(String args[]) {

        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();
        Star star = new Star();

        circle.draw();
        rectangle.draw();
        star.draw();

   }

}
```

The test application TestShape creates three classes: Circle, Rectangle, and Star. To draw these classes, TestShape asks the individual classes to draw themselves:

```
circle.draw();
rectangle.draw();
star.draw();
```

When you execute TestShape, you get the following results:

```
C:\>java TestShape
I am drawing a Circle
I am drawing a Rectangle
I am drawing a Star
```

This is polymorphism at work. What would happen if you wanted to create a new shape, such as Triangle? Simply write the class, compile it, test it, and use it. The base class Shape does not have to change—nor does any other code:

```
public class Triangle extends Shape{

    public void draw() {

        System.out.println("I am drawing a Triangle");

    }
}
```

A message can now be sent to Triangle. And even though Shape does not know how to draw a triangle, the Triangle class does:

```
public class TestShape {

    public static void main(String args[]) {

        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();
        Star star = new Star();
        Triangle triangle = new Triangle ();

        circle.draw();
        rectangle.draw();
        star.draw();
        triangle.draw();

   }

}
```

```
C:\>java TestShape
I am drawing a Circle
I am drawing a Rectangle
I am drawing a Star
I am drawing a Triangle
```

To see the real power of polymorphism, you can pass the shape to a method that has absolutely no idea what shape is coming. Take a look at the following code, which includes the specific shapes as parameters:

```
public class TestShape {

   public static void main(String args[]) {

     Circle circle = new Circle();
     Rectangle rectangle = new Rectangle();
     Star star = new Star();

     drawMe(circle);
     drawMe(rectangle);
     drawMe(star);

   }

   static void drawMe(Shape s) {
     s.draw();
   }

}
```

In this case, the `Shape` object can be passed to the method `drawMe()`, and the `drawMe()` method can handle any valid `Shape`—even one you add later. You can run this version of `TestShape` just like the previous one.

Abstract Classes, Virtual Methods, and Protocols

Abstract classes, as they are defined in Java, can be directly implemented in .NET and C++ as well. Not surprisingly, the C# .NET code looks similar to the Java code, as shown in the following:

```
public abstract class Shape{

   public abstract void draw();

}
```

The Visual Basic .NET code is written like this:

```
Public MustInherit Class Shape

    Public MustOverride Function draw()

End Class
```

The same functionality can be provided in C++ using virtual methods with the following code:

```
class Shape
{
    public:
        virtual void draw() = 0;
}
```

As mentioned in previous chapters, Objective-C and Swift do not fully implement the functionality of abstract classes.

For example, consider the following Java interface code for the `Shape` class we have seen many times:

```
public abstract class Shape{

    public abstract void draw();

}
```

The corresponding Objective-C (Swift) protocol is shown in the following code. Note that in both the Java code and the Objective-C code, there is no implementation for the `draw`() method.

```
@protocol Shape

@required
- (void) draw;

@end // Shape
```

At this point, the functionality for the abstract class and the protocol are pretty much equivalent; however, here is where the Java-type interface and protocols diverge. Consider the following Java code:

```
public abstract class Shape{

    public abstract void draw();

    public void print() {
        System.out.println("I am printing");
    };

}
```

In the preceding Java code, the `print` () method provides code that can be inherited by a subclass. Although this is also the case with C# .NET, VB .NET, and C++,the same cannot be said for an Objective-C protocol, which would look like this:

```
@protocol Shape

@required
- (void) draw;
- (void) print;

@end // Shape
```

In this protocol, the `print()` method signature is provided, and thus must be implemented by a subclass; however, no code can be included. In short, subclasses cannot directly inherit any code from a protocol. Thus, the protocol cannot be used in the same way as an abstract class, and this has implications when designing an object model.

Conclusion

This chapter gives a basic overview of what inheritance and composition are and how they are different. Many well-respected OO designers have stated that composition should be used whenever possible, and inheritance should be used only when necessary.

However, this is a bit simplistic. I believe that the idea that composition should be used whenever possible hides the real issue, which might be that composition is more appropriate in more cases than inheritance—not that it should be used whenever possible. The fact that composition might be more appropriate in most cases does not mean that inheritance is evil. Use both composition and inheritance, but only in their proper contexts.

In earlier chapters, the concepts of abstract classes and Java interfaces arose several times. In Chapter 8, we explore the concept of development contracts and how abstract classes and Java interfaces are used to satisfy these contracts.

References

Booch, Grady and Robert A. Maksimchuk and Michael W. Engel and Bobbi J. Young, Jim Conallen, and Kelli A. Houston. 2007. *Object-Oriented Analysis and Design with Applications*, Third Edition. Boston, MA: Addison-Wesley.

Coad, Peter, and Mark Mayfield. 1997. *Java Design*. Upper Saddle River, NJ: Prentice Hall.

Gilbert, Stephen, and Bill McCarty. 1998. *Object-Oriented Design in Java*. Berkeley CA: The Waite Group Press.

Meyers, Scott. 2005. *Effective C++*, Third Edition. Boston, MA: Addison-Wesley Professional.