Chapter 10

Mechanisms for Software Reuse

Object-oriented programming has been billed as the technology that will finally permit software to be constructed from general-purpose reusable components. Writers such as Brad Cox have even gone so far as to describe object orientation as heralding the "industrial revolution" in software development [Cox 1986]. While the reality may not quite match the expectations of OOP pioneers, it *is* true that object-oriented programming makes possible a level of software reuse that is orders of magnitude more powerful than that permitted by previous software construction techniques. In this chapter, we will investigate the two most common mechanisms for software reuse, which are known as *inheritance* and *composition*.

Inheritance in Java is made more flexible by the presence of two different mechanisms, interfaces and subclassing. In addition to contrasting inheritance and composition, we will contrast inheritance performed using subclassing and inheritance performed using interfaces, and relate all to the concept of substitutability.

10.1 Substitutability

The concept of substitutability fundamental to many of the most powerful software development techniques in object-oriented programming. You will recall that substitutability referred to the situations that occurs when a *variable* declared as one type is used to hold a *value* derived from another type.

In Java, substitutability can occur either through the use of classes and inheritance, or through the use of interfaces. We have seen examples of both in our earlier case studies. In the Pin Ball game program described in Chapter 7, the variable targets was declared as a PinBallTarget, but in fact held a variety of different types of values that were created using subclasses of PinBallTarget. In the Solitaire program from Chapter 9, the variable allPiles was declared as holding a CardPile, but in fact held values that were subclasses of CardPile, such as DeckPile and DiscardPile:

```
public class Solitare {
    static public CardPile allPiles [ ];

    public void init () {
            ...
        allPiles = new CardPile[13];
            // then fill them in
        allPiles[0] = new DeckPile(335, 30);
        allPiles[1] = new DiscardPile(268, 30);
    }
    ...
}
```

We have also seen examples of substitutability arising through interfaces. An example is the instance of the class FireButtonListener created in the Cannon-ball game (Chapter 6). The class from which this value was defined was declared as implementing the interface ActionListener. Because it implements the ActionListener interface, we can this value as a parameter to a function (in this case, addActionListener) that expects an ActionListener value.

```
class CannonWorld extends Frame {
    ...
    private class FireButtonListener implements ActionListener {
        public void actionPerformed (ActionEvent e) {
            ...
            }
    }

    public CannonWorld () {
            ...
        fire.addActionListener(new FireButtonListener());
    }
}
```

We will return to the distinction between inheritance of classes and inheritance of interfaces in Section 10.1.2.

10.1.1 The Is-a Rule and the Has-a Rule

A commonly employed rule-of-thumb that can be used to understand when inheritance is an appropriate software technique is known colloquially as *is-a* and *has-a* (or *part-of*) relationship.

The *is-a* relationship holds between two concepts when the first is a specialized instance of the second. That is, for all practical purposes the behavior and data associated with the more specific idea form a subset of the behavior and data associated with the more abstract idea. For example, all the examples of inheritance we described in the early chapters satisfy the *is-a* relationship (a Florist *is-a* Shopkeeper, a Dog *is-a* Mammal, a PinBall *is-a* Ball, and so on). The relationship derives its name from a simple rule of thumb that tests the relationship. To determine if concept X is a specialized instance of concept Y, simply form the English sentence "*An* X *is a* Y". If the assertion "sounds correct," that is, if it seems to match your everyday experience, you may judge that X and Y have the *is-a* relationship.

The *has-a* relationship, on the other hand, holds when the second concept is a component of the first but the two are not in any sense the same thing, no matter how abstract the generality. For example, a Car *has-a* Engine, although clearly it is not the case that a Car *is-a* Engine or that an Engine *is-a* Car. A Car, however, *is-a* Vehicle, which in turn *is-a* MeansOfTransportation. Once again, the test for the *has-a* relationship is to simply form the English sentence "*An* X *has a* Y", and let common sense tell you whether the result sounds reasonable.

Most of the time, the distinction is clear-cut. But, sometimes it may be subtle or may depend on circumstances. In Section 10.2 we will use one such indefinite case to illustrate the two software development techniques that are naturally tied to these two relationships.

10.1.2 Inheritance of Code and Inheritance of Behavior

There are at least two different ways in which a concept can satisfy the *is-a* relationship with another concept, and these are reflected in two different mechanisms in the Java language. Inheritance is the mechanism of choice when two concepts share *structure* or *code* relationship with each other, while an interface is the more appropriate technique when two concepts share the *specification of behavior*, but no actual code.

This can be illustrated with examples from the Java run-time library. The class Frame, from which most Java windows inherit, provides a great deal of code, in the form of methods that are inherited and used without being overridden. Thus, inheritance using the extends modifier in the class heading is the appropriate mechanism to use in this situation.

```
    // a cannon game is a type of Frame
public class CannonGame extends Frame {
    ...
}
```

On the other hand, the characteristics needed for an ActionListener (the object type that responds to button presses) can be described by a single method, and the implementation of that method cannot be predicted, since it differs from one application to another. Thus, an interface is used to describe only the necessary requirements, and no actual behavior is inherited by a subclass that implements the behavior.

```
class CannonWorld extends Frame {
    ...
        // a fire button listener implements the action listener interface
    private class FireButtonListener implements ActionListener {
        public void actionPerformed (ActionEvent e) {
        ...
        }
    }
}
```

In general, the class-subclass relationship should be used whenever a subclass can usefully inherit either code, data values, or behavior from the parent class. The interface mechanism should be used when the child class inherits only the specification of the expected behavior, but no actual code.

10.2 Composition and Inheritance Described

Two different techniques for software reuse are *composition* and *inheritance*. We have explicitly noted uses of inheritance in earlier chapters, and have several places used composition as well, although we have not pointed it out. One way to view these two different mechanisms is as manifestations of the *has-a* rule and the *is-a* rule, respectively.

Although in most situations the distinction between *is-a* and *has-a* is clear-cut, it does happen occasionally that it can be difficult to determine which mechanism is most appropriate to use in a particular situation. By examining one such indefinite case, we can more easily point out the differences between the use of inheritance and the use of composition. The example we will use is taken from the Java library, and concerns the development of a Stack abstraction from an existing Vector data type.

The Vector data type is described in detail in Chapter 19. While abstractly a vector is most commonly thought of as an indexed collection, the Java implementation also permits values to be added or removed from the end of the collection, growing and shrinking the container as necessary. The particular methods of interest for this discussion can be described as follows:

```
class Vector {
        // see if collection is empty
    public boolean isEmpty () { ... }

        // return size of collection
```

```
    public int size () { ... }

        // add element to end of collection
    public void addElement (Object value) { ... }

        // return last element in collection
    public Object lastElement () { ... }

        // remove element at given index
    public Object removeElementAt (int index) { ... }

    ...
}
```

A *stack* is an abstract data type that allows elements to be added or removed from one end only. If you think about a stack of dishes sitting on a counter you can get a good intuition. It is easy to access the topmost dish, or to place a new dish on the top. It is much more difficult to access any dish other than the topmost dish. In fact, it might be that the only way to do this is to remove dishes one by one until you reach the dish you want.

The Stack abstractions defined here will be slightly simpler than the version provided by the Java library. In particular, the library abstraction will throw an exception if an attempt is made to access or remove an element from an empty stack, a condition we will ignore. The Java library stack routine names the empty test method *empty*, instead of the method isEmpty from class Vector, and finally the Stack abstraction provides a method to search the stack to determine if it includes a given element. We will not describe this method here.

10.2.1 Using Composition

We will first investigate how the stack abstraction can be formed with composition. Recall from our earlier discussion that an object is simply an encapsulation of data values and behavior. When composition is employed to reuse an existing data abstraction in the development of a new data type, a portion of the state of the new data structure is simply an instance of the existing structure. This is illustrated in Figure 10.1, where the data type Stack contains a private instance field named theData, which is declared to be of type Vector.

Because the Vector abstraction is stored as part of the data area for our stack, it must be initialized in the constructor. The constructor for class Stack allocates space for the vector, giving a value to the variable theData.

Operations that manipulate the new structure are implemented by making use of the existing operations provided for the earlier data type. For example, the implementation of the isEmpty operation for our stack data structure simply invokes the similarly named function already defined for vectors. The peek operation is known by a different name, but

```
class Stack {
    private Vector theData;

    public Stack ()
        { theData = new Vector(); }

    public boolean isEmpty ()
        { return theData.isEmpty(); }

    public Object push (Object newValue)
        { theData.addElement (newValue); }

    public Object peek ()
        { return theData.lastElement(); }

    public Object pop () {
        Object result = theData.lastElement();
        theData.removeElementAt(theData.size()-1);
        return result;
        }
};
```

Figure 10.1: A Stack created using Composition

the task is already provided by the lastElement operation in the Vector class. Similarly, the push operation is simply performed by executing an addElement on the vector.

The only operation that is slightly more complex is popping an element from the stack. This involves using two methods provided by the Vector class, namely obtaining the topmost element, and removing it from the collection. Notice that to remove the element we must first determine its index position, then remove the element by naming its position.

The important point to emphasize is the fact that composition provides a way to leverage off an existing software component in the creation of a new application. By use of the existing Vector class, the majority of the difficult work in managing the data values for our new component have already been addressed.

On the other hand, composition makes no explicit or implicit claims about substitutability. When formed in this fashion, the data types Stack and Vector are entirely distinct and neither can be substituted in situations where the other is required.

10.2.2 Using Inheritance

An entirely different mechanism for software reuse in object-oriented programming is the concept of inheritance; with which a new class can be declared a *subclass*, or *child class*, of an existing class. In this way, all data areas and functions associated with the original class are automatically associated with the new data abstraction. The new class can, in addition, define new data values or new functions; it can also *override* functions in the original class, simply by defining new functions with the same names as those of functions that appear in the parent class.

These possibilities are illustrated in the class description shown in Figure 10.2, which implements a different version of the Stack abstraction. By naming the class Vector in the class heading, we indicate that our Stack abstraction is an extension, or a refinement, of the existing class Vector. Thus, all data fields and operations associated with vectors are immediately applicable to stacks as well.

The most obvious features of this class in comparison to the earlier are the items that are missing. There are no local data fields. There is no constructor, since no local data values need be initialized. The method isEmpty need not be provided, since the method is inherited already from the parent class Vector.

Compare this function with the earlier version shown in Figure 10.1. Both techniques are powerful mechanisms for code reuse, but unlike composition, inheritance carries an implicit assumption that subclasses are, in fact, subtypes. This means that instances of the new abstraction should react similarly to instances of the parent class.

In fact, the version using inheritance provides more useful functionality than the version using composition. For example, the method size in the Vector class yields the number of elements stored in a vector. With the version of the Stack formed using inheritance we get this function automatically for free, while the composition version needs to explicitly add a new operation if we want to include this new ability. Similarly, the ability to access

```
class Stack extends Vector {

    public void push (Object value)
        { addElement (value); }

    public Object peek ()
        { return lastElement(); }

    public Object pop () {
        Object result = lastElement();
        removeElementAt(theData.size()-1);
        return result;
        }
};
```

Figure 10.2: A stack created using Inheritance

intermediate values directly using an index is in this version possible with a stack, but not permitted in the abstraction created using composition.

10.3 Composition and Inheritance Contrasted

Having illustrated two mechanisms for software reuse, and having seen that they are both applicable to the implementation of stacks, we can comment on some of the advantages and disadvantages of the two approaches.

- Inheritance carries with it an implicit, if not explicit, assumption of substitutability. That is, classes formed by inheritance are assumed to be subtypes of the parent class, and therefore candidates for values to be used when an instance of the parent class is expected. No such assumption of substitutability is associated with the use of composition.

- Composition is the simpler of the two techniques. Its advantage is that it more clearly indicates exactly what operations can be performed on a particular data structure. Looking at the declaration for the Stack data abstraction, it is clear that the only operations provided for the data type are the test for emptiness, push, peek and pop. This is true regardless of what operations are defined for vectors.

- In inheritance the operations of the new data abstraction are a superset of the operations of the original data structure on which the new object is built. Thus, to know exactly what operations are legal for the new structure the programmer must examine the declaration for the original. An examination of the Stack declaration, for example, does not immediately indicate that the size method can be legally applied to stacks. It is only by examination of the declaration for the earlier Vector data abstraction that the entire set of legal operations can be ascertained.

The difficulty that occurs when, to understand a class constructed using inheritance, the programmer must frequently flip back and forth between two (or more) class declarations has been labeled the "yo-yo" problem [Taenzer 1989].

- The brevity of data abstractions constructed with inheritance is, in another light, an advantage. Using inheritance it is not necessary to write any code to access the functionality provided by the class on which the new structure is built. For this reason, implementations using inheritance are almost always, as in the present case, considerably shorter in code than are implementations constructed with composition, and they often provide greater functionality. For example, the inheritance implementation makes available not only the size test for stacks but also the index related operations (inserting, modifying or removing elements by giving their index locations).

- Inheritance does not prevent users from manipulating the new structure using methods from the parent class, even if these are not appropriate. For example, when we use inheritance to derive the class Stack from the class Vector, nothing prevents users from adding new elements to the stack using the inherited method insertElementAt, and thereby placing elements in locations other than the top of the stack.

- In composition the fact that the class Vector is used as the storage mechanism for our stack is merely an implementation detail. With this technique it would be easy to reimplement the class to make use of a different technique (such as a linked list), with minimal impact on the users of the Stack abstraction. If users counted on the fact that a Stack is merely a specialized form of Vector, such changes would be more difficult to implement.

- A component constructed using inheritance has access to fields and methods in the parent class that have been declared as protected. A component constructed using composition can only access the public portions of the included component.

- Inheritance may allow us to use the new abstraction as an argument in an existing *polymorphic* function. We will investigate this possibility in more detail in Chapter 12. Because composition does not imply substitutability, it usually precludes polymorphism.

- Understandability and maintainability are difficult to judge. Inheritance has the advantage of brevity of code but not of protocol. Composition code, although longer, is the only code that another programmer must understand to use the abstraction. A programmer faced with understanding the inheritance version needs to ask whether any behavior inherited from the parent class was necessary for proper utilization of the new class, and would thus have to understand both classes.

- Data structures implemented through inheritance tend to have a very small advantage in execution time over those constructed with composition, since one additional function call is avoided (although optimization techniques, such as inline functions, can in theory be used to eliminate much of this overhead).

Of the two possible implementation techniques, can we say which is better in this case? One answer involves the substitution principle. Ask yourself whether, in an application that expected to use a Vector data abstraction, it is correct to substitute instead an instance of class Stack.

The bottom line is that the two techniques are very useful, and an object-oriented programmer should be familiar with both of them.

10.4 Combining Inheritance and Composition

The Java I/O system, which we will investigate in more detail in Chapter 14, provides an interesting illustration of the way in which inheritance and composition can interact with each other, and the particular problems each mechanism is designed to solve.

To begin with, there is an abstract concept, and several concrete realizations of the concept. For the file input system the abstract concept is the idea of reading a stream of bytes in sequence, one after the other. This idea is embodied in the class InputStream, which defines a number of methods for reading byte values. The concrete realizations differ in the source of the data values. Values can come from an array of bytes being held in memory, from an external file, or from another process that is generating values as needed. There is a different subclass of InputStream for each of these.

```
InputStream
    ├── ByteArrayInputStream
    ├── FileInputStream
    ├── PipedInputStream
    ├── SequenceInputStream
    └── StringBufferInputStream
```

Because each of these is declared as a subclass of InputStream, they can be substituted for a value of type InputStream. In this fashion procedures can be written to process a stream of byte values, without regard to where the values originate (whether in memory, or from an external file, or from another process).

However, there is an additional source of variation among input streams. Or rather, there is additional functionality that is sometimes required when using an input stream. Furthermore, this functionality is independent of the source for the byte values. One example is the ability to keep track of line numbers, so that the programmer can determine on which line of the input the current byte originates. Another useful function is the ability to buffer input, so as to have the possibility of rereading recently referenced bytes once again.

These features are provided by defining a subclass of InputStream, named FilterInputStream. A FilterInputStream is formed as a subclass of InputStream. Thus, using the principle of substitutability, a FilterInputStream can be used in places where an InputStream is expected. On the other hand, a FilterInputStream holds as a component another instance of InputStream, which is used as the source for data values. Thus, the class InputStream is both parent and component to FilterInputStream. As requests for values are received, the FilterInputStream will access the InputStream it holds to get the values it needs, performing whatever additional actions are required (for example, counting newline characters in order to keep track of the current line number).

```
class FilterInputStream extends InputStream {

    protected InputStream in;
    ...
}
```

Because the component held by the FilterInputStream can be any type of InputStream, this additional functionality can be used to augment and type of InputStream, regardless of where the byte values originate. This idea of *filters* (sometimes called *wrappers* in other object-oriented literature) can be quite powerful when there are orthogonal sources of variation. Here the two reasons of variation are the source of the input values, and the additional functionality that may or may not be needed in any particular application.

10.5 Novel Forms of Software Reuse

In this section we will describe several novel ways in which composition, or inheritance, or both, are used to achieve different effects.

10.5.1 Dynamic Composition

One advantage of composition over inheritance concerns the delay in binding time. With inheritance, the link between child class and parent class is established at compile time, and cannot later be modified. With composition, the link between the new abstraction and the

```
class Frog {
    private FrogBehavior behavior;

    public Frog () {
        behavior = new TadpoleBehavior();
        }

    public grow () { // see if behavior should change
        if (behavior.growUp())
            behavior = new AdultFrogBehavior();
        behavior.grow(); // behavior does actual work
        behavior.swim();
        }
}
```

Figure 10.3: Class Frog holds dynamically changing behavior component

older abstraction is created at run-time, and is therefore much weaker, since it can also be changed at run time. This is sometimes called *dynamic composition*.

To illustrate, imagine a class that is simulating the behavior of a Frog. Although the frog interface can be fixed throughout its life, the actual actions it performs might be very different when the frog is a tadpole or when it is an adult. One way to model this is to create a class Frog that uses composition to hold a value of type FrogBehavior (Figure 10.3). The Frog class is largely a facade, invoking the FrogBehavior object to do the majority of the real work.

The novel idea is that the variable holding the instance of FrogBehavior can actually be polymorphic. There might be more than one class that implements the FrogBehavior specification, such as TadpoleBehavior and AdultFrogBehavior. Figure 10.4 illustrates these classes. The parent class FrogBehavior is here declared *abstract*, which means that it *must* be overridden before any instances can be created. As the frog "grows", it can dynamically change behavior by reassigning the value behavior to a different value (for example, moving from a TadpoleBehavior to an AdultFrogBehavior). The user of the Frog abstraction need not know about this change, or even be aware when it occurs.

Dynamic composition is a useful technique if the behavior of an object varies dramatically according to some internal concept of "state", and the change in state occurs infrequently.

10.5.2 Inheritance of Inner Classes

We have seen another combination of inheritance and composition in some of the case studies presented in earlier chapters. For example, the "listener" classes that responded to events were constructed using inheritance, but were themselves components in the application class.

```
abstract class FrogBehavior {
    public boolean growUp () { return false; }

    public void grow ();

    public void swim ();
}

class TadpoleBehavior extends FrogBehavior {
    private int age = 0;

    public boolean growUp () { if (++age > 24) return true; }

    public void grow () { ... }

    public void swim () { ... }
}

class AdultFrogBehavior extends Behavior {

    public void grow () { ... }

    public void swim () { ... }
}
```

Figure 10.4: Class FrogBehavior can dynamically change

The following is a skeleton of the class PinBallGame from Chapter 7.

```
public class PinBallGame extends Frame {
    ...
    private class MouseKeeper extends MouseAdapter { ... }

    private class PinBallThread extends Thread { ... }
}
```

The classes MouseKeeper and PinBallThread are each constructed using inheritance. Each are then used to create a new component, that will be held as part of the state of the pin ball game. When used in this fashion, inner classes combine aspects of both inheritance and composition.

10.5.3 Unnamed Classes

In several of the earlier case studies we have seen situations where inheritance is used to override an existing class, yet only one instance of the new class is created. An alternative to naming the new class and then creating an instance of the named class is to use a class definition expression. Such an expression places the entire class definition inside the instance creation expression.

An example where this could be used is in the definition of an event listener. For example, the CannonBall game described in Chapter 6 contained the following code:

```
class CannonWorld extends Frame {
    ...

    private class FireButtonListener implements ActionListener {
        public void actionPerformed (ActionEvent e) {
            ...
            }
    }

    public CannonWorld () {
        fire.addActionListener(new FireButtonListener());
        ...
        }
}
```

Note how the constructor for the class CannonWorld creates an instance of the inner class FireButtonListener. This is the only instance of this class created. An alternative way to achieve the same effect would be as follows:

```
class CannonWorld extends Frame {
    ...

    public CannonWorld () {
        fire.addActionListener(new ActionListener(){
            public void actionPerformed (ActionEvent e) {
                ...
                }
            });
        ...
        }
}
```

Notice that in this example the object being created is declared only as being an instance of ActionListener. However, a class definition follows immediately the ending parenthesis, indicating that a new and *unnamed* class is being defined. This class definition would have exactly the same form as before, ending with a closing curly brace. The parenthesis that follows this curly brace ends the argument list for the addActionListener call. (The reader should carefully match curly braces and parenthesis to see how this takes place).

An advantage to the use of the class definition expression in this situation is that it avoids the need to introduce a new class name (in this case, the inner class name FireButtonListener). A disadvantage is that such expressions tend to be difficult to read, since the entire class definition must be wrapped up as an expression, and the close of the expressions occurs after the end of the class definition.

10.6 Chapter Summary

The two most common techniques for reusing software abstractions are inheritance and composition. Both are valuable techniques, and a good object-oriented system will usually have many examples of each.

A good rule of thumb for deciding when to use inheritance is the *is-a* rule. Form the sentence “An X is a Y”, and if it sounds right to your ear, then the two concepts X and Y can probably be related using inheritance. The corresponding rule for composition is the *has-a* rule.

The decision whether to use inheritance or not is not always clear. By examining one borderline case, one can more easily see some of the advantages and disadvantages of the two software structuring techniques.

The idea of *filters*, found in the portion of the Java library used for input and output, is an interesting technique that combines features of both inheritance and composition.

Study Questions

1. Explain in your own words the principle of substitutability.
2. What is the *is-a* rule? What is the *has-a* rule? How are these two rules related to inheritance and composition?
3. How are interfaces and inheritance related?
4. What are some of the advantages of composition over inheritance?
5. What are some of the advantages of using inheritance that are not available when composition is used?
6. How does a FilterStream combine inheritance and composition?

Exercises

1. A set is simply an unorganized collection of values. Describe how one could use a Vector to implement a set. Would inheritance or composition be the more appropriate mechanism in this case?
2. Modify the Stack data abstractions given in this chapter so that they will throw a EmptyStackException if an attempt is made to read or remove a value from an empty stack.