Chapter 5

Inheritance and Interfaces

Finding commonality among classes makes for effective object-oriented programming. Often, programmers express that commonality using an inheritance hierarchy, since that is one of the first concepts taught in object-oriented programming.

We're going to go to the other extreme in this chapter to explore the difference between using inheritance and using interfaces. An emphasis on interfaces guides you in determining what is the real essence of a class; once you have determined the essence, then you can look for commonalities between classes.

Creating an inheritance hierarchy prematurely can cause extra work when you then need to untangle it. If you start with interfaces and discover an appropriate hierarchy, you can easily refactor into that hierarchy. Refactoring into an inheritance hierarchy is far easier than refactoring out of an existing hierarchy.

We will look at examples of alternative designs that emphasize either inheritance or interfaces, so you can compare the two approaches. An interface-oriented alternative of a real-world Java inheritance hierarchy demonstrates the differences in code.

5.1 Inheritance and Interfaces

You probably learned inheritance as one of the initial features of object-oriented programming. With inheritance, a derived class receives the attributes and methods of a base class. The relationship between the

derived and base class is referred to as "is-a" or more specifically as "is-a-kind-of." For example, a mammal "is-a-kind-of" animal. Inheritance creates a class hierarchy.

You may hear the term *inherits* applied to interfaces. For example, a PizzaShop that implements the PizzaOrdering interface is often said to inherit the interface.[1] However, it is a stretch to say that a PizzaShop "is-a" PizzaOrdering. Instead, a more applicable relationship is that a PizzaShop "provides-a" PizzaOrdering interface.[2] Often modules that implement PizzaOrdering interfaces are not even object-oriented. So in this book, we use the term *inherits* only when a derived class inherits from a base class, as with the **extends** keyword in Java. A class "implements" an interface if it has an implementation of every method in the interface. Java uses the **implements** keyword precisely for this concept.[3]

Inheritance is an important facet of object-oriented programming, but it can be misused.[4] Concentrating on the interfaces that classes provide, rather than on their hierarchies, can help prevent inheritance misuse, as well as yield a more fluid solution to a design. Let's look at some alternate ways to view example designs using both an inheritance-style approach and an interface-style approach. Both inheritance and interfaces provide polymorphism, a key feature of object-oriented design, so let's start there.

5.2 Polymorphism

A common form of polymorphism consists of multiple classes that all implement the same set of methods. Polymorphism of this type can be organized in two ways. With inheritance, a base class contains a set of methods, and derived classes have the same set of methods. The derived classes may inherit implementations of some methods and contain their own implementations of other methods. With interfaces, multiple classes each implement all the methods in the interface.

---

[1] Using a single term to represent two different concepts can be confusing. For example, how many different meanings are there for the keyword **static** in C++?

[2] You may see adjectives used for interface names, such as Printable;. With an adjective, you may see a reference such as a Document "is" Printable. The "is" in this case really means that a Document "provides-a" Printable interface.

[3] See the examples in Chapter 1 for how to code interfaces in C# and C++.

[4] See *Designing Reusable Classes* by Ralph E. Johnson and Brian Foote, `http://www.laputan.org/drc/drc.html`.

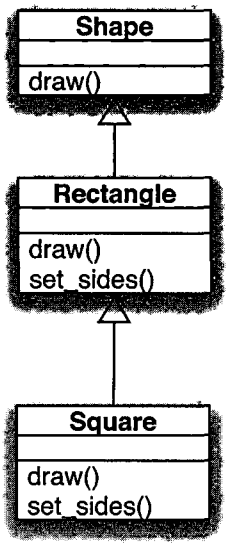

Figure 5.1: SHAPE HIERARCHY

With inheritance, the derived classes must obey the contract (of Design by Contract) of the base class. This makes an object of a derived class substitutable for an object of the base class. With interfaces, the implementation must also obey the contract, as stated in the First Law of Interfaces (see Chapter 2).

An example of an inheritance that violates a contract is the Shape hierarchy. The hierarchy looks like Figure 5.1.

```
class Shape
    draw()
class Rectangle extends Shape
    set_sides(side_one, side_two)
    draw()
class Square extends Rectangle
   set_sides(side_one, side_two)
   draw()
```

A Rectangle is a Shape. A Square is a Rectangle. Square inherits the set_sides() method from Rectangle. For a Rectangle, any two positive values for `side_one` and `side_two` are acceptable. A Square can accept only two equal values. According to Design by Contract, a derived class can have less strict preconditions and stricter postconditions. This

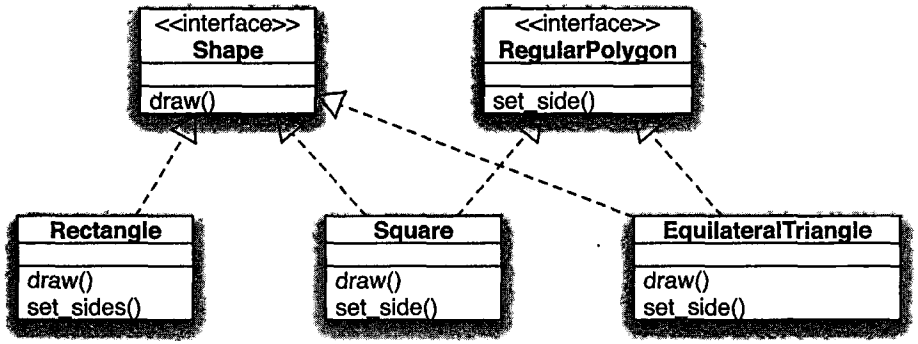

Figure 5.2: DIAGRAM OF INTERFACES

situation violates that rule, and thus the hierarchy is not ideal.

Although a Square is a Rectangle from a geometric point of view, it does not have the same behavior as a Rectangle. The error in this example comes from translating the common statement that "a square is a rectangle" into an inheritance hierarchy.

An alternative organization (Figure 5.2) using interfaces is as follows:

```
interface Shape
    draw()
Rectangle implements Shape
    set_sides(side_one, side_two)
    draw()
interface RegularPolygon
    set_side(measurement)
Square implements Shape, RegularPolygon
    set_side(measurement)
    draw()
EquilateralTriangle implements Shape, RegularPolygon
    set_side(measurement)
    draw()
```

With these interfaces, Square provides the Shape methods, but it also provides the methods in RegularPolygon. Square can obey the contract in both of these interfaces.

One difficulty with interfaces is that implementations may share common code for methods. You should not duplicate code; you have two ways to provide this common code. First, you can create a helper class and delegate operations to it. For example, if all RegularPolygons need to

compute the perimeter and to compute the angles at the vertices, you could have this:

```
class RegularPolygonHelper
    set_side(measurement)
    compute_perimeter()
    compute_angle()
```

Implementers of RegularPolygon would delegate operations to this class in order to eliminate duplicate code.

Second, you could create a class that implemented the interface and provided code for many, if not all, of the methods (such as the Java Swing adapter classes for event listeners shown in Chapter 3). You would then derive from that class instead of implementing the interface. For example:

```
interface RegularPolygon
    set_side(measurement)
    compute_perimeter()
    compute_angle()
class DefaultRegularPolygon implements RegularPolygon
    set_side(measurement)
    compute_perimeter()
    compute_angle()
class Square extends DefaultRegularPolygon, implements Shape
    set_side(measurement)
    compute_perimeter()
    compute_angle()
    draw()
```

In the case of single-inheritance languages, you need to decide which of the two potential base classes (Shape or RegularPolygon) is the more important one. If you decide on Shape, then you'll still need RegularPolygonHelper. Determining which one is important can be difficult until you have more experience with the classes. Starting with interfaces allows you to postpone that decision until you have that experience.

USING INTERFACES
: Advantage—delay forming hierarchy until usage known

USING INHERITANCE
: Advantage—less delegation of common operations

5.3 Hierarchies

The animal kingdom is a frequently used hierarchy example. The hierarchy starts with Animal on top. Animal breaks down into Mammals,

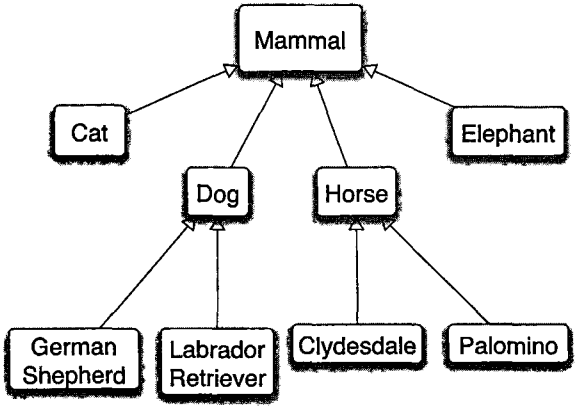

Figure 5.3: MAMMALIAN HIERARCHY

Fishes, Birds, Reptiles, Amphibians, etc. The relationships parallel those of an object-oriented hierarchy: a cow "is-a" Mammal. The subclasses (derived classes) have attributes in common with the superclasses (base classes). This zoological classification is based on characteristics used to identify animals; Figure 5.3 shows a portion of the standard hierarchy.

The animal hierarchy is useful for identification, but it does not necessarily represent behavior. The hierarchy represents data similarities. Mammals all have hair (except perhaps whales and dolphins), are warm-blooded, and have mammary glands. The organization does not refer to services—things that animals do for us. Depending on your application that uses animals, a service-based description of animals may be more appropriate. The service-based description cuts across the normal hierarchy. Looking at what these animals do for us, we might have the following:

- Pull a Vehicle: Ox, Horse
- Give Milk: Cow
- Provide Companionship: Cat, Dog, Horse
- Race: Horse, Dog
- Carry Cargo: Horse, Elephant
- Entertain: Cat, Dog, Tiger, Lion, Elephant

Linnaean Taxonomy

Carolus Linnaeus developed the standard biological classification system. The system classifies species based on similarities in their forms and other traits that usually, but not always, reflect evolutionary relationships.

A problem with Linnaean taxonomy is that reclassification of an existing species or discovery of a new one can lead to changes in rank. Rank (i.e., in the Kingdom/Phylum/Class/Order/Family/Genus/Species breakdown) is denoted by suffices (e.g., "ae" as in "Hominidae"). A rank change requires renaming whole suites of taxonomic groups. (This need to reorganize an inheritance scheme may seem familiar to programmers.)

Phylocode is another biological classification system. It is based on common ancestry and the branching of the evolutionary tree. It is organized by species and clades—group of organisms sharing a particular ancestor. It is more immune to the need for reorganization. Just as in programming, picking the appropriate inheritance hierarchy can make changes simpler.*

*(See "Attacks on Taxonomy," *American Scientist*, July–August, 2005)

We could organize these methods in the same way we did printers in Chapter 3; e.g., each animal could have a "can you do this for me" method, such as can_you_carry_cargo(). Alternatively, we could have a set of interfaces as shown in Figure 5.4, on the next page. Animals would implement only the interfaces they could perform. The methods in the interfaces might be:

```
interface Pullers
    hook_up
    pull_hard
    pull_fast

interface MilkGivers
    give_milk
    give_chocolate_milk

interface CompanionshipGivers
    sit_in_lap
    play_for_fun
```

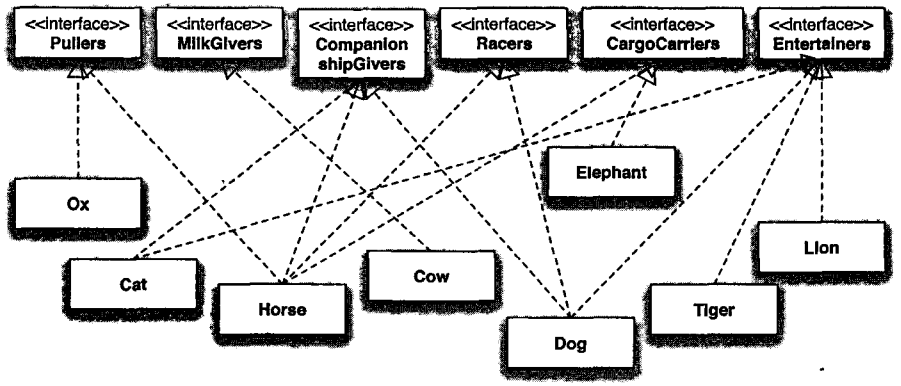

Figure 5.4: ANIMAL INTERFACES

```
interface Racers
    run_fast
    run_long

interface CargoCarriers
    load_up
    get_capacity

interface Entertainers
    jump_through_hoop
    stand_on_two_legs
```

- Depending on the application, you may employ both a hierarchy and service-based interfaces. For example, you might have a Dog hierarchy whose base class implemented the methods for CompanionShipGivers, Racers, and Entertainers. Particular breeds of dogs could inherit from Dog to obtain a default implementation.

  You might also have a need for interfaces based on common characteristics that cross hierarchies, such as LiveInWater, Vegetarian, etc. These interfaces could each have a helper class that provided common implementations. Classes such as Cow, Horse, and Ox could delegate to a VegetarianHelper class.

- USING INTERFACES

  Advantage—can cross hierarchies

USING INHERITANCE

Advantage—captures common attributes

Inheritance and Methods

Inheritance delineates a hierarchy of classes that all implement methods of the base class. The base class represents a general type, such as Mammal. The derived classes represent more specialized types, such as Cow and Horse. The derived classes may not necessarily offer additional methods.

On the other hand, derived classes can extend the base class and offer more methods. For example, for the Printer class in Chapter 4, a ColorPrinter represents more services than a Printer. When a derived class adds more methods to the base class, those additional methods can be considered an additional responsibility for the derived class. An interface could represent this additional responsibility.

For example, GUI components are usually organized as an inheritance hierarchy, like this:

```
class Component
    set_position()
    abstract draw()
class TextBox extends Component
    draw()
    set_text()
    get_text()
class CheckBox extends Component
    draw()
    set_state()
    get_state()
```

Here TextBox and CheckBox have additional methods that represent additional services for each derived class. Those additional methods could be denoted as interfaces, like this :

```
class Component
    set_position()
    abstract draw()
interface Textual
    set_text()
    get_text()
class TextBox extends Component, implements Textual
    draw()
    set_text()
    get_text()
```

```
interface Checkable
    set_state()
    get_state()
class CheckBox extends Component, implements Checkable
    draw()
    set_state()
    get_state()
```

- If each derived class has its own unique set of additional methods, there is no advantage to organizing the hierarchy with interfaces. However, if many of the derived classes do have a common set of services, you may make those commonalities more apparent by using interfaces.

  For example, a drop-down box and a multiple selection list are usually on one branch of a GUI hierarchy. Radio buttons and check boxes are on another branch of the hierarchy. These two separate branches are based on their relative appearances. Another way to group commonality is to put radio buttons and drop-down lists together and multiple-selection lists and check boxes together. Each of those groups has the same functionality. In the first group, the widgets provide selection of a single value. In the second group, the widgets provide the option of multiple values.[5] In this organization, they are grouped based on behavior, not on appearance. This grouping of behavior can be coded with interfaces:

```
interface SingleSelection
    get_selection()
    set_selection()
interface MultipleSelection
    get_selections()
    set_selections()
class RadioButtonGroup implements SingleSelection
class CheckBoxGroup implements MultipleSelection
class DropDownList implements SingleSelection
class MultipleSelectionList implements MultipleSelection
```

- USING INTERFACES

  Advantage—can capture common set of usage

- USING INHERITANCE

  Advantage—captures set of common behavior

[5] You might also put a list that allows only a single selection into this group.

Football Team

The members of a football team can be depicted with either inheritance or interfaces. If you represented the positions with inheritance, you might have an organization that looks like this:[6]

```
FootballPlayer
    run()

        DefensivePlayer extends FootballPlayer
            tackle()

                DefensiveBackfieldPlayer extends DefensivePlayer
                    cover_pass()

        OffensivePlayer extends FootballPlayer
            block()

                Center extends OffensivePlayer
                    snap()

                OffensiveReceiver extends OffensivePlayer
                    catch()
                    run_with_ball()

                OffensiveBackfieldPlayer extends OffensivePlayer
                    catch()
                    receive_handoff()
                    run_with_ball()

                Quarterback extends OffensivePlayer
                    handoff()
                    pass()
```

An object of one of these classes represents a player. So, Peyton Manning would be an object of Quarterback. Based on the methods in the hierarchy, Peyton can run, block, hand off, and pass. This hierarchy looks pretty good. On the other hand, we can make our organization more fluid by using interfaces, like this:

```
interface FootballPlayer
    run()
interface Blocker
    block()
interface PassReceiver
    catch()
```

[6]The services listed for each position are the required ones for each position. You could require that all FootballPlayers be able to catch and throw. The base class FootballPlayer would provide a basic implementation of these skills.

```
interface BallCarrier
    run_with_ball()
    receive_handoff()
interface Snapper
    snap()
interface Leader
    throw_pass()
    handoff()
    receive_snap()
interface PassDefender()
    cover_pass_receiver()
    break_up_pass()
    intercept_pass()
```

- A *role* combines one or more interfaces. We might come up with the following roles for team members:

```
Center implements FootballPlayer, Blocker, Snapper
GuardTackle implement FootballPlayer, Blocker
EndTightOrSplit implements FootballPlayer, Blocker, PassReceiver
RunningBack implements FootballPlayer, BallCarrier, PassReceiver
Fullback implements Blocker, FootballPlayer, BallCarrier, PassReceiver
WideReceiver implements FootballPlayer, PassReceiver
Quarterback implements FootballPlayer, Leader, BallCarrier
```

- Now along comes Deion Sanders, who plays both offense and defense. To fit Deion into the previous hierarchy, you need to create two objects: one an OffensivePlayer and the other a DefensivePlayer. Or you'd need to come up with some other workaround that does not fit cleanly into the hierarchy. With interfaces, Deion simply fulfills another role, like this:

```
SwitchPlayer implements FootballPlayer, PassReceiver, PassDefender
```

Roles can even be more fluid. For example, in one professional game, a backup quarterback lined up as a wide receiver.[7] Trying to fit such a role into a hierarchy can be daunting. With interfaces, he would have simply implemented PassReceiver, or he could take on a role like this:

```
ReceiverQuarterback implements FootballPlayer, PassReceiver, Quarterback
```

USING INTERFACES

Advantage—give more adaptability for roles that cross hierarchies

Disadvantage—may have duplicated code without helper classes to provide common functionality

[7]This was Seneca Wallace in a Carolina Panthers/Seattle Seahawks game, for you trivia buffs.

USING INHERITANCE

Advantage—base classes can provide common implementations

Disadvantage—difficult to adapt to new situations.

5.4 An Interface Alternative for InputStream

What do football players, mammals, and geometric shapes have in common? We've used them as examples to show the differences between inheritance and interfaces. Let's look at a real-life class hierarchy and see how an alternative organization with interfaces would appear. This is a concrete example of the concepts discussed in the previous sections.

Java's java.io.InputStream class is an abstract class.[8] InputStream contains many methods defined as abstract, such as read(). Other methods are concrete but contain this statement in the documentation: "This method should be overridden by subclasses." A few other methods only suggest that they should be overridden. For example, a method that reads an array of bytes is provided. Its code simply calls the read() method for each byte, but the documentation suggests that a concrete implementation could code this better. Many methods in the class have an implementation that does nothing (e.g., close()).[9]

To contrast an inheritance approach with an interface approach in a real-code example, we will transform the InputStream hierarchy into an interface-based design. This transformation follows the concepts of the "Replace Inheritance with Delegation" refactoring.[10]

InputStream Interface

Suppose we have a CustomInputStream we developed by inheriting from InputStream such as in Figure 5.5, on the following page. We start our transformation by extracting an interface from the current methods of the abstract InputStream class:

[8] See `http://www.docjar.com/html/api/java/io/InputStream.java.html`

[9] This discussion ignores the additional methods that InputStream inherits from the Object. InputStream does not override any of those methods. Any implementation of the InputStream interface will also inherit from Object and thus have those same additional methods.

[10] See *Refactoring* by Martin Fowler, et al.

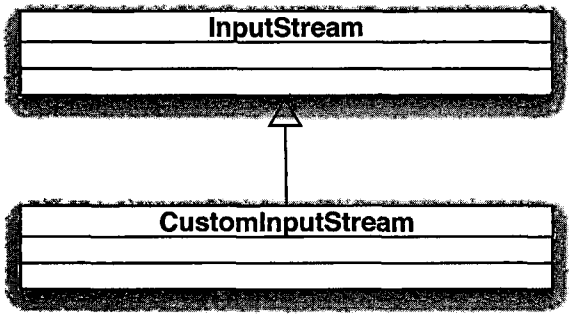

Figure 5.5: INHERITANCE

```
public interface InputStream
    {
    public int available() throws IOException;
    public void close()throws IOException;
    public void mark(int readlimit)
    public boolean markSupported()
    public int read()throws IOException;
    public int read(byte [] bytes) throws IOException;
    public int read(byte [] bytes, int offset, int length)
        throws IOException;
    public void reset()throws IOException;
    public  long skip(long n) throws IOException;
    }
```

- The current implementation code in InputStream is put into an InputStream Default class as shown in Figure 5.6, on the next page. A particular InputStream, say CustomInputStream, inherits from InputStreamDefault:

```
public class InputStreamDefault implements InputStream
    {
    // same as current source of java.io.InputStream class
    }
public class CustomInputStream extends InputStreamDefault
    {
    }
```

At this point, we now have an interface that classes in other hierarchies can implement. We can transform CustomInputStream so that it implements InputStream directly, by taking the method bodies in InputStreamDefault and moving them to CustomInputStream. But we would end up with duplicate code if classes in other hierarchies implemented InputStream. So, we remove the methods from InputStreamDefault and place

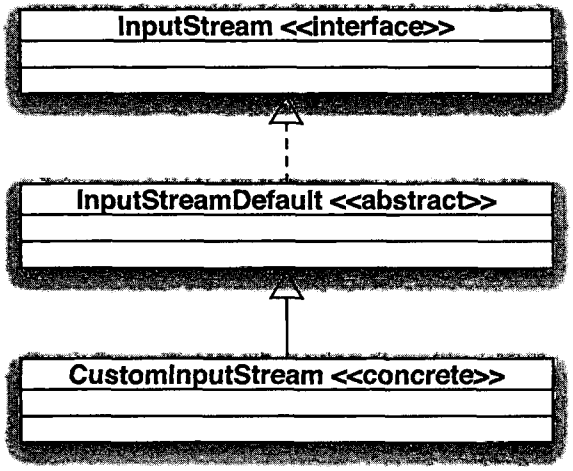

Figure 5.6: INHERITANCE

them in a helper class InputStreamHelper. CustomInputStream or other classes can delegate operations to this helper class. (See Figure 5.7, on the following page.)

```
public class InputStreamHelper
    {
    private InputStream inputStream;
    public InputStreamHelper(InputStream input)
        {
        inputStream = input;
        }
    public int read(byte [] bytes) throws IOException
        {
        read(bytes[], 0, byte.length);
        }
    public int read(byte [] bytes, int offset, int length)
        throws IOException;
        {
        // calls inputStream.read() and places bytes into array
        }
    public long skip(long n) throws IOException
        {
        // Calls inputStream.read() to skip over n bytes
        }
    }
```

The code for CustomInputStream now looks like the following. We show details only for the methods where we delegate operations to the helper class.

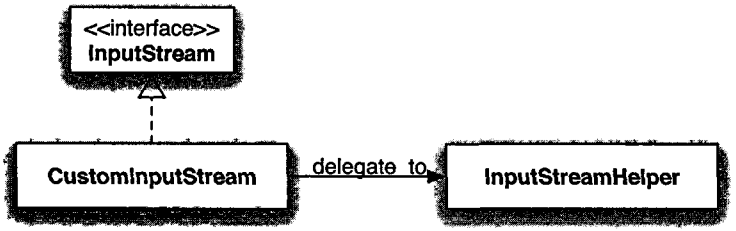

Figure 5.7: DELEGATION

```
class CustomInputStream implements InputStream
    {
    private InputStreamHelper inputHelper;
    public CustomInputStream()
        {
        inputHelper = new InputStreamHelper(this);
        }
    public int available() throws IOException {...}
    public void close()throws IOException {...}
    public void mark(int readlimit) {...}
    public boolean markSupported() {...}
    public int read()throws IOException {...}
    public int read(byte [] bytes) throws IOException
        {
        return inputHelper.read(bytes);
        }
    public int read(byte [] bytes, int offset, int length)
        throws IOException
        {
        return inputHelper.read(bytes, offset, length);
        }
    public void reset()throws IOException
    public long skip (long n) throws IOException
        {
        return inputHelper.skip(n) ;
        }
    }
```

The Input Marker

The mark methods (mark(), markSupported(), and reset()) in the java.io.InputStream class are examples of "tell me if you do this" interfaces. For those not familiar with the Java I/O library, these methods work as follows.

If markSupported() returns true, then mark() can be used to mark a position in the input stream. When reset() is called, the next call to read() returns bytes starting at the marked position. So, the same bytes are returned again by read(). If the number of bytes read between the call to mark() and the call to reset() exceeds the readlimit number of bytes, then reset() does not have to work. Only classes that override markSupported() to return true need to implement mark() and reset().

If a method receives a reference to a java.io.InputStream object, it should call markSupported() before using mark() and reset().[11] The code for this method could look something like this:

```
void method(java.io.InputStream input)
    {
    if (input.markSupported())
        mark();
    //.... and other stuff
    }
```

The InputStream interface that we created includes the markSupported() method as well as the mark() and reset() methods. Instead of using the "can you do this" method approach, we can add another interface:

```
interface Markable
    {
    public void mark(int readlimit);
    public void reset()throws IOException;
    }
```

Only classes that support mark() and reset() need to implement this interface. In a method, you can tell whether an InputStream supports marking by performing a cast. If an exception is thrown, you could pass the exception back to the caller as a violation of the interface contract. ("You must call this method with a Markable InputStream.") The code looks like this:

```
a_method(InputStream input) throws ClassCastException
    {
    Markable marking = (Markable) input;
    marking.mark();                //...and  marking.reset()
     }
```

Alternatively, if the method can perform its operation in an alternative way without requiring the Markable interface, it does not need to throw the exception.

[11] I'm sure there is some application where calling mark() or reset() without checking markSupported() makes sense. I just can't think of one at the moment.

By separating the Markable interface, you're simplifying the InputStream interface. Also, it becomes clear from just looking at the class definition which classes support marking. For example:

```
class ByteArrayInputStream implements InputStream, Markable
```

This definitively shows you can mark an object of ByteArrayInputStream.[12]

FileInputStream: Yet Another Interface

We're still not done with evolving some more interfaces for InputStreams. The java.io.FileInputStream class is derived from the java.io.InputStream class. If we used interfaces as shown previously, we would state that FileInputStream implements InputStream and does not implement Markable. The java.io.FileInputStream class has additional methods that can be grouped as an interface, say FileInterface. This interface includes two methods:

```
FileInterface
    FileChannel getChannel()
    FileDescriptor getFD()
```

The java.io.FileDescriptor class refers to the underlying operating system file descriptor (equivalent to the Unix file descriptor shown in the Unix section in Chapter 1). The java.nio.channels.FileChannel class represents a way to asynchronously access a file.[13]

With FileInterface, FileInputStream can be described as follows:

```
class FileInputStream implements InputStream, FileInterface
```

java.io.FileOutputStream is the corresponding class for file output. It is derived from java.io.OutputStream in standard Java. Using interfaces, we could describe this class as follows:

```
class FileOutputStream implements OutputStream, FileInterface
```

Note that FileInterface is implemented by both FileInputStream and FileOutputStream. This shows the relationship between these two classes that

[12] "Definitively" may be too strong a word. As one reviewer noted, "Just because you said you're going to do it, doesn't mean you really are doing it." Declaring the class to implement the interface really should imply that the class will honor the contract for the interface.

[13] FileChannel provides additional methods for reading and writing to a file. FileChannel is an abstract class that implements several interfaces. They include ByteChannel, Channel, GatheringByteChannel, InterruptibleChannel, ReadableByteChannel, ScatteringByteChannel, and WritableByteChannel. Without going into details about all these interfaces, suffice it to say that the structure of the channel classes reflects an interface-based approach to design.

appear on different branches in the standard Java hierarchy. With the standard library, there are separate implementations for the methods in each class. Seeing a common interface may prompt you to provide a common helper class for FileInterface that could eliminate this code duplication.

InputStream Review

When you have multiple implementations of an interface such as InputStream, you may duplicate logic in each implementation. If there are common implementation methods and you do not use a helper class, you may find yourself copying and pasting a lot. If you can create a well-defined hierarchy with many inheritable implementations, you are far better off using inheritance, rather than interfaces. But you may find that starting with interfaces and then refactoring to inheritance allows you to discover what is the best hierarchy.

INHERITANCE APPROACH

Advantage—easy to inherit an implementation

Disadvantage—may be difficult to adapt to changing roles

INTERFACE APPROACH

Advantages—can be clearer what methods must be implemented.

A class in another inheritance hierarchy can provide the services of an interface.

Disadvantage—may end up with lots of helper classes.

5.5 Things to Remember

You can design with an emphasis on either inheritance or interfaces:

- Interfaces show commonality of behavior.
- Inheritance shows commonality of behavior along with commonality of implementation.
- Interfaces are more fluid than inheritance, but you need to use delegation to share common code.
- Inheritance couples the derived classes to the base class, but it allows them to easily share the common code of the base class.

What if it rings but no one answers? Do you try again, thinking you may have dialed the wrong number? Do you assume that they aren't open?

Suppose you get cut off in the middle of the call. Do you call back?

External Interfaces

The problems of pizza ordering exist in any external interface. An external interface is one called by other processes (either local or remote). External interfaces differ from local interfaces by network considerations, by nonhomogeneity of hosts, and by multiple process interactions.[1]

- If an entire software system is contained within a single process, the system fails if the process fails. With a system consisting of multiple processes, a calling process (e.g., a client) has to handle the unavailability of other processes (e.g., a server). The client usually continues to run in spite of the failure of servers, but it needs either to communicate the server failure to the user or to act on that failure in a transparent manner, as per the Third Law of Interfaces.
- Remote interfaces are external interfaces that are accessed over a network. In addition to server failure, with a network you may have a network delay or a network failure. Note that if you are unable to connect to a server, it is difficult to determine whether the network is down or whether the server is down. Likewise, a delay may be due to an overloaded network or an overloaded server that is handling too many clients. In either case, the client needs to handle the failure.
- With nonhomogeneity, the client and the server may be different processor types (e.g., IBM mainframe versus PC). Even on a local machine, where processor types are not a consideration, the caller and server may be written in different programming languages.

Network Disruption

What would you do if you were ordering a pizza by phone and the call dropped before you heard how long it was going to take? You'd call back. You'd try to continue to describe the pizza you were ordering. But

[1] A local interface is usually called by only one process, although it may be called by multiple threads within that process. A remote interface can typically be concurrently called by multiple remote processes.

the pizza shop says, "I've had hundreds of orders in the last minute for just that type of pizza." You don't really want to order a second pizza. And the store owner doesn't want to make an unclaimed pizza. How would we change the pizza-ordering interface to avoid this?

The store owner could take some identification at the beginning of a call—a name or a phone number. If the circuit is broken, you call back and give the same name or phone number. If the order taker determines the name corresponds to one of the uncompleted orders, he pulls it off the shelf and resumes at the appropriate place in the order sequence.[2]

Getting initial identification is a form of planning for the possibility of communication disruption. The interface protocol should assume that the network may go down. In a manner similar to the pizza shop, interfaces can use client-generated IDs to ensure that service invocations are not duplicated. For example, when credit card transactions are submitted to a bank, the merchant identifies each transaction with a unique ID. If the connection is broken in the middle of transmitting a transaction, the merchant resubmits transactions that have not been explicitly confirmed. The bank knows by the ID for the particular merchant whether a transaction has already been posted. If the transaction has been posted, the bank merely confirms the transaction without reposting it.

6.2 Procedural and Document Interfaces

In our example in Chapter 1, you called the pizza shop over the phone. Your pizza shop may also accept fax orders. What is different about making a phone call versus sending a fax order? In either case, the order needs to be put into terms both you and the pizza shop understand. With the voice system, you execute a series of operations to create an order. With the fax, you have an order form that defines the required and optional data.

Problems are discovered immediately in the voice system. For example, you can ask for anchovies and get an immediate response. The voice on the other end can say "nope," meaning either they never have anchovies

[2]Some readers might note that a name such as Jim might be given for different orders. If the given name matches a previous name, the order taker may inform you that you have to give a different name. A phone number is not only good for identification but also for verification. The store owner can check the caller ID against the phone number to see whether it's the same one you said.

(a permanent error) or they don't have any today (a temporary error). In either case, you can make up your mind whether you want to have an anchovyless pizza or find some other pizza place.

With the a fax-based system, you fill out an order and await a response. The response may be a fax with the time till delivery or a fax saying, "Don't have any." If the latter, you need to alter your order and resubmit it. You may wonder whether your order was received. Since you may have to wait a while to get a fax back, it is harder to determine when to resend the order. The pizza parlor's fax may be out of paper. The scanner for the return fax may not be working. The order may have been put onto a pile. Only when the order is retrieved from the pile is a fax returned. We shall see how these issues of ordering by fax have parallels in remote interfaces.

External interfaces can use either procedural style or document style. A procedural interface looks like the interfaces we've been describing in this book. On the other hand, document-style interfaces use sets of data messages, similar to the fax-based pizza order.

For the most flexibility, the client (interface user) and the server (interface implementation provider) should be loosely coupled in terms of platform and language. A client written in any language should be able to access the server. You can accomplish this decoupling with either style.

Procedural Style

You can use Common Object Request Broker Architecture (CORBA) to define procedural-style interfaces that are both language and platform independent.[3] With CORBA, you specify the interface with the Interface Definition Language (IDL).[4] IDL looks a lot like a C++ header or a Java interface. A transformation program turns an IDL declaration into code stubs appropriate for a particular language and platform. An example of an interface defined in IDL is as follows:

```
enum Size {SMALL, MEDIUM, LARGE};
enum Toppings {PEPPERONI, MUSHROOMS, PEPPERS, SAUSAGE};
```

[3] You can define remote interfaces in a language-dependent manner, such as Java's Remote Method Invocation. You could also define them in a platform-dependent manner, such as Window's Distributed Component Object Model (DCOM).

[4] See `http://www.omg.org` for more information about CORBA and IDL.

```
interface PizzaOrdering
    {
    exception UnableToDeliver(string explanation);
    exception UnableToMake(string explanation);
    typedef Toppings ToppingArray[5];
    set_size(in Size the_size) raises (UnableToMake);
    set_toppings(ToppingArray toppings) raises (UnableToMake);
    set_address(in string street_address);
    TimePeriod get_time_till_delivered() raises (UnableToDeliver);
    }
```

Procedural-style remote interfaces look familiar to programmers. Calls to methods in remote interfaces (a Remote Procedure Call [RPC]) appear in your code as if they were calls to local interfaces. The only major difference is that the code must handle communication failure situations. RPCs are typically used for an immediate request/response in interactive situations. A client that called the PizzaOrdering interface can find out immediately whether the shop cannot make the pizza.

Procedural-style interfaces tend to be fine-grained. For example, they frequently contain operations for accessing individual values such as set_size() in the PizzaOrdering interface.

Document Style

With document style, the client and server interchange a series of data messages (documents). For a pizza order, the sequence might start with a document:

```
Document: PizzaOrder
    Size
    Toppings
    Address
```

The response could be either like this:

```
Document: SuccessResponse
    TimePeriod time_to_deliver
```

or like this:

```
Document: ErrorResponse
    String error_explanation
```

You may be less familiar with document-style interfaces. The documents represent a series of messages transmitted between the client and the service provider. The protocol is defined by the sequence of messages. We'll explore a typical sequence later in this chapter. Messages are not necessarily processed immediately. Response documents,

such as SuccessResponse, may come almost immediately. However, they may also be delayed. A client using the document interface to order pizzas may not instantly find out whether the requested pizza can be made.

- A document-style interface tends to be very coarse-grained. For example, a PizzaOrder document that contains the size and toppings is sent in a single operation, like this:[5]

```
interface Ordering
    submit_order(PizzaOrder)
```

- PROCEDURAL STYLE

  Advantage—remote and local interfaces can appear the same

  Disadvantage—can require more communication (especially if fine-grained)

- DOCUMENT STYLE

  Advantage—can require less communication

  Disadvantages—style is less familiar to programmers

6.3 Facets of External Interfaces

We discussed several facets of interfaces in Chapter 3. Now we'll examine some additional facets of external interfaces.

Synchronous versus Asynchronous

In Chapter 3, we described asynchronous event handling using the Observer pattern. Likewise, communication between a client and a server can be either synchronous or asynchronous. With synchronous interfaces, the client does not end communication until a result is returned.

- With asynchronous interfaces, a result is returned at some other time after the client has ended the original communication. For example, documents are often placed on message queues. The client creates a

[5] The most general document interface consists of three operations:
Request/response—`Document send_document_and_get_response(Document)`
Request—`void send_document(Document)`
Response—`Document receive_document()`
That's so coarse-grained, you can transmit anything. (OK, maybe not anything, but almost anything).

document (message) and puts it onto a message queue. The client usually continues processing, handling other events. At some time, the server retrieves the message from the queue and processes the document. It then returns a response document, either directly to the client or back onto the queue for retrieval by the client.

Two typical combinations of modes for applications that use external interfaces are asynchronous/document (e.g., message queues) and synchronous/procedural (e.g., RPCs). You could consider the World Wide Web to be an synchronous/document interface: you send a document (e.g., a filled-in form) and receive a document (a new web page) in return. The least frequently used combination is asynchronous/procedural.

SYNCHRONOUS

Advantage—practically immediate response

Disadvantage—cannot scale up as well

ASYNCHRONOUS

Advantage—can scale well, especially with document queues

Disadvantage—documents should be validated on client before transmitting

Stateful versus Stateless

With remote interfaces, the distinction between stateful and stateless interfaces is more critical. A server that keeps state for clients may not be able to handle as many clients as a server that does not keep state.

For example, many web sites have shopping carts. In a stateful interface, the contents of the shopping cart are kept in a semipersistent storage on the server. Each new connection from a client (i.e., a browser) creates a new shopping cart that is assigned a SessionID. The SessionID is the key that identifies the data for a particular client on the server. The browser returns this SessionID with each request for another web page. The server uses the SessionID to retrieve current contents of the shopping cart.

In a stateless interface, the server does not keep any state. For example, with a Google search, the URL passes the search parameters every time. If Google keeps any information on a search, it is for performance reasons, rather than for interface reasons.

A stateful interface can be turned into a stateless interface in a manner similar to that shown in Chapter 3. All state information can be kept on the client and passed every time to the server. The server updates the state information and passes it back to the client. For a stateless shopping cart, the entire contents of the shopping cart are transmitted to and returned from the server for each web page.

REMOTE STATELESS

Advantages—servers can be easily scaled. If you have multiple servers processing client requests, any server can handle any client.

The service has redundancy. Any server could go down and the client could continue with any other server.

Disadvantage—amount of state information passed between client and server can grow, especially for a full shopping cart. In most cases, this amount will be less than the size of the web pages for which the state information is transferred.

REMOTE STATEFUL

Advantage—less information to communicate between client and server

Disadvantage—if using central database where the state information is stored, the amount of simultaneous connections to that database could be a limiting factor to scalability

Stability versus Flexibility

Stability is a needed trait for external interfaces. You typically have no knowledge of who is accessing the interface, so you can't change it willy-nilly. However, flexibility is also needed for interfaces, since you may want to add features to them in the future. Both procedural and document interfaces can be flexible.

You should follow a few guidelines in being flexible in interfaces. First, never alter the semantics of existing methods or data. Second, you can add methods or data; just don't expect that users will add calls to the new methods or provide the new data. For example, you could add a country field to an address. But you still should handle a document that does not contain a country, by creating a reasonable default value for country.

But although adding functionality is easy, deleting methods or data is hard. Callers may still expect that they still exist and call the methods

or supply the data. Methods in some computer language libraries that have been designated as "deprecated" may still be called years later. Only if you have control of both sides of the interface (the client and the server), or you have a sufficiently desired interface, do you have a possibility of deleting deprecated methods.[6]

All interfaces should be identified by an explicit version ID. An ID provides a simpler way to determine the version of the client than by ascertaining which particular methods or data a client uses. Knowing a client's version aids the server in dealing with older versions. A server should be able to handle not only the current version of a service but also older versions. Obviously at some point, you would like to be able to remove support for really old versions. Depending on the importance of your service to the clients and your relative political strength, you may need to continue to handle numerous versions for a long time.[7]

6.4 Discovery of Services

You've come to a new city. You're dying for a pizza. You look up in the Yellow Pages[8] for a pizza place. (Ok, maybe you look on Google Maps to find a close one.) You look under *Pizza*. You know that any listed pizza shop should implement the PizzaOrdering interface.[9] How do you pick one from the hundreds of pizza places listed? You can choose the random method, the first on the list, the closest, or you can go for a known brand name.

You probably did this lookup once in your hometown and posted the phone number on your bulletin board. That parallels how Microsoft's DCOM works. You make an entry in the registry to indicate where a

[6]The situation of telling users that methods that have been marked as deprecated are finally being deleted is similar to that of passengers boarding a plane. Even though all the passengers know when the flight is leaving, some passengers still need to be informed over the loudspeakers. Even then, the gate agent needs to decide when to finally let the plane depart without the missing passengers.

[7]To help you determine when it's appropriate to remove support, you can log the version IDs that clients use to access an interface. When the log shows that the only clients using the old version are those for whom you have blackmailable information, you can remove the old version.

[8]It's always amazing what you learn when writing a book: "Yellow Pages" is a trademark in the public domain.

[9]How do you know it's really a pizza shop? Maybe the pizza is made on a truck as it's on its way to deliver the pizza to you.