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 derived and base class is referred to as “is-a” or more specifically as “isa-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. 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. 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.

Inheritance is an important facet of object-oriented programming, but it can be misused. 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.

---

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++.
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.

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.

```java
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 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:

```java
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:

```java
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:

```java
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 your 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, 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.
Figure 5.3: Mammalian 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

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. Animals would implement only the interfaces they could perform. The methods in the interfaces might be:
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:

```java
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:

```java
class Component
    set_position()
    abstract draw()

interface Textual
    set_text()
    get_text()
```
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 selections 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. In this organization, they are grouped based on behavior, not on appearance. This grouping of behavior can be coded with interfaces:

```java
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:

```java
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, Payton Manning would be an object of `Quarterback`. Based on the methods in the hierarchy, Payton 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:

```java
interface FootballPlayer
  run()
interface Blocker
  block()
interface PassReceiver
  catch()
interface BallCarrier
  run_with_ball()
  receive_handoff()
interface Snapper
  snap()
interface Leader
  throw_pass()
  handoff()
  receive_snap()
```

---

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

---

This excerpt is published by developer.* with the express permission of the publisher of the book Interface Oriented Design, The Pragmatic Programmers. developer.* is grateful to the publisher for granting permission for this publication, and to the author, Ken Pugh. If you enjoyed this excerpt, you will likely enjoy the book also. Buy direct from the publisher, or wherever books are sold. ©2005 Pragmatic Programmers, LLC.

---

7 This was Seneca Wallace in a Carolina Panthers/Seattle Seahawks game, for you trivia buffs.
About the Author

Ken Pugh has worked on software and hardware projects for over thirty years, from requirements gathering through testing and maintenance. He has a wide variety of experience with numerous operating systems, languages, and development processes. He has developed software systems extending from long-baseline interferometry to real-time goat serum process control, and embedded systems for signal processing to networked file storage.

As a teacher and mentor, he has trained thousands of students in subjects ranging from object-oriented design to UNIX operating system internals. He has presented at numerous conferences seminars on software development processes, programming techniques, and system architecture. Ken has written four books on programming and operating systems.