Chapter 8

Inheritance and Substitution

The first step in learning object-oriented programming is understanding the basic philosophy of organizing the performance of a task as the interaction of loosely coupled software components. This organizational approach was the central lesson in the case studies of Chapters 6 and 7.

The *next* step in learning object-oriented programming is organizing classes into a hierarchical structure based on the concept of inheritance. By *inheritance*, we mean the property that instances of a child class (or subclass) can access both data and behavior (methods) associated with a parent class (or superclass).

8.1 An Intuitive Description of Inheritance

Let us return to Chris and Fred, the customer and florist from the first chapter. There is a certain behavior we expect florists to exhibit, not because they are florists but simply because they are shopkeepers. For example, we expect Fred to request money for a transaction and in turn give back a receipt. These activities are not unique to florists, but are common to bakers, grocers, stationers, car dealers, and other merchants. It is as though we have associated certain behavior with the general category Shopkeeper, and as Florists are a specialized form of shopkeepers, the behavior is automatically identified with the subclass.

In programming languages, inheritance means that the behavior and data associated with child classes are always an *extension* (that is, a larger set) of the properties associated with parent classes. A subclass will have all the properties of the parent class, and other properties as well. On the other hand, since a child class is a more specialized (or restricted) form of the parent class, it is also, in a certain sense, a *contraction* of the parent type. This tension between inheritance as expansion and inheritance as contraction is a source for much of the power inherent in the technique, but at the same time it causes much confusion as to its proper employment. We will see this when we examine a few of the uses of inheritance in a subsequent section.

Inheritance is always transitive, so that a class can inherit features from superclasses many levels away. That is, if class Dog is a subclass of class Mammal, and class Mammal is a subclass of class Animal, then Dog will inherit attributes both from Mammal and from Animal.

8.1.1 The Is-a Test

As we noted in Chapter 2, there is a rule-of-thumb that is commonly used to test whether two concepts should be linked by an inheritance relationship. This heuristic is termed the *is-a* test. The is-a test says that to tell if concept A should be linked by inheritance to concept B, try forming the English sentence "A(n) A is a(n) B." If the sentence "sounds right" to your ear, then inheritance is most likely appropriate in this situation. For example, the following all seem like reasonable assertions:

A Bird is an Animal
A Cat is a Mammal
An Apple Pie is a Pie
A TextWindow is a Window
A Ball is a GraphicalObject
An IntegerArray is an Array

On the other hand, the following assertions seem strange for one reason or another, and hence inheritance is likely not appropriate:

A Bird is a Mammal
An Apple Pie is an Apple
An Engine is a Car
A Ball is a Wall
An IntegerArray is an Integer

There are times when inheritance can reasonably be used even when the is-a test fails. Nevertheless, for the vast majority of situations, it gives a reliable indicator for the appropriate use of the technique.

8.1.2 Reasons to Use Inheritance

Although there are many uses for the mechanism of inheritance, two motivations far outweigh all other concerns:

- Inheritance as a means of code reuse. Because a child class can inherit behavior from a parent class, the code does not need to be rewritten for the child. This can greatly reduce the amount of code needed to develop a new idea.

- Inheritance as a means of concept reuse. This occurs when a child class overrides behavior defined in the parent. Although no code is shared between parent and child, the child and parent share the definition of the method.

An example of the latter was described in the previous chapter. The variable that was declared as holding a GraphicalObject could, in fact, be holding a Ball. When the message draw was given to the object, the code from class Ball, and not from GraphicalObject, was the method selected. Both code and concept reuse often appear in the same class hierarchies.

8.2 Inheritance in Various Languages

Object-oriented languages can be divided into those languages that require every class to inherit from an existing parent class, and those languages that do not. Java, Smalltalk, Objective-C, and Delphi Pascal are examples of the former, while C++ and Apple Pascal are examples of the latter. For the former group we have already seen the syntax used to indicate inheritance, for example in Figure 4.3 of Chapter 4. In Figure 8.1 we reiterate some of these, and also show the syntax used for some of the languages in the second group.

One advantage given to those languages that insist that all classes inherit from an existing class is that there is then a single root that is ancestor to all objects. This root class is termed Object in Smalltalk and Objective-C, and termed TObject in Delphi Pascal. Any behavior provided by this root class in inherited by all objects. Thus, every object is guaranteed to possess a common minimal level of functionality.

The disadvantage of a single large inheritance tree is that it combines all classes into a tightly coupled unit. By having several independent inheritance hierarchies, programs in C++ and other languages that do not make this restriction are not forced to carry a large library of classes, only a few of which may be used in any one program. Of course, that means there is no programmer-defined functionality that *all* objects are guaranteed to possess.

In part, the differing views of objects are one more distinction between languages that use dynamic typing and those that use static typing. In dynamic languages, objects are characterized chiefly by the messages they understand. If two objects understand the same set of messages and react in similar ways, they are, for all practical purposes, indistinguishable regardless of the relationships of their respective classes. Under these circumstances, it is useful to have all objects inherit a large portion of their behavior from a common base class.

8.3 Subclass, Subtype, and Substitution

Consider the relationship of a data type associated with a parent class to a data type associated with a derived, or child, class in a statically typed object-oriented

Public, Private and Protected

In earlier chapters we have seen the use of the terms public and private. A public feature is accessible to code outside the class definition, while a private feature is accessible only within the class definition. Inheritance introduces a third alternative. In C++ (also in C#, Delphi, Ruby and several other languages) a protected feature is accessible only within a class definition or within the definition of any child classes. Thus a protected feature is more accessible than a private one, and less accessible than a public feature. This is illustrated by the following example:

```
class Parent {
private:
    int three;
protected:
    int two;
public:
    int one;
    Parent () { one = two = three = 42; }
    void inParent ()
        { cout << one << two << three; /* all legal */ }
};

class Child : public Parent {
public:
    void inChild () {
        cout << one; // legal
        cout << two; // legal
        cout << three; // error - not legal
    }
};

void main () {
    Child c;
    cout << c.one; // legal
    cout << c.two; // error - not legal
    cout << c.three; // error - not legal
}
```

The lines marked as error will generate compiler errors. The private feature can be used only within the parent class. The protected feature only within the parent and child class. Only public features can be used outside the class definitions.

Java uses the same keyword, but there protected features are legal within the same package in which they are declared.

C++	`class Wall : public GraphicalObject {`
	`    ...`
	`}`

C#	`class Wall : GraphicalObject {`
	`    ...`
	`}`

CLOS	`(defclass Wall (GraphicalObject) () )`

Java	`class Wall extends GraphicalObject {`
	`    ...`
	`}`

Object Pascal	`type`
	`    Wall = object (GraphicalObject)`
	`        ...`
	`    end;`

Python	`class Wall(GraphicalObject):`
	`    def __init__(self):`
	`    ...`

Ruby	`class Wall < GraphicalObject`
	`   ...`
	`end`

Figure 8.1: Syntax Used to Indicate Inheritance in Several Languages

language. The following observations can be made:

- Instances of the child class must possess all data members associated with the parent class.
- Instances of the child class must implement, through inheritance at least (if not explicitly overridden) all functionality defined for the parent class. (They can also define new functionality, but that is unimportant for the present argument).
- Thus, an instance of a child class can mimic the behavior of the parent class and should be *indistinguishable* from an instance of the parent class if substituted in a similar situation.

We will see later in this chapter, when we examine the various ways in which inheritance can be used, that this is not always a valid argument. Nevertheless, it is a good description of our idealized view of inheritance. We will therefore formalize this ideal in what is called the *principle of substitution*.

The principle of substitution says that if we have two classes, A and B, such that class B is a subclass of class A (perhaps several times removed), it should be possible to substitute instances of class B for instances of class A in *any situation* with *no observable effect*.

The term *subtype* is used to refer to a subclass relationship in which the principle of substitution is maintained, to distinguish such forms from the general *subclass* relationship, which may or may not satisfy this principle. We saw a use of the principle of substitution in Chapter 7. Section 7.4 described the following procedure:

```
procedure drawBoard;
var
    gptr : GraphicalObject;
begin
        (* draw each graphical object *)
    gptr := listOfObjects;
    while gptr <> nil do begin
        gptr.draw;
        gptr := gptr.link;
    end;
end;
```

The global variable listOfObjects maintains a list of graphical objects, which can be any of three types. The variable gptr is declared to be simply a graphical object, yet during the course of executing the loop it takes on values that are, in fact, derived from each of the subclasses. Sometimes gptr holds a ball, sometimes a hole, and sometimes a wall. In each case, when the draw function is invoked, the correct method for the current value of gptr will be executed–not the method in the declared class GraphicalObject. For this code to operate correctly it is imperative that the functionality of each of these subclasses match the expected functionality specified by the parent class; that is, the subclasses must also be subtypes.

All object oriented languages will support the principle of substitution, although some will require additional syntax when a method is overridden. Most support the concept in a very straight-forward fashion; the parent class simply holds a value from the child class. The one major exception to this is the language C++. In C++ only pointers and references truly support substitution; variables that are simply declared as value (and not as pointers) do not support substitution. We will see why this property is necessary in C++ in a later chapter.

8.3.1 Substitution and Strong Typing*

Statically typed languages (such as C++ and Object Pascal) place much more emphasis on the principle of substitution than do dynamically typed languages (such as Smalltalk and Objective-C). The reason for this is that statically typed languages tend to characterize objects by their class, whereas dynamically typed languages tend to characterize objects by their behavior. For example, a polymorphic function (a function that can take objects of various classes) in a statically typed language can ensure a certain level of functionality only by insisting that all arguments be subclasses of a given class. Since in a dynamically typed language arguments are not typed at all, the same requirement would be simply that an argument must be able to respond to a certain set of messages.

An example of this difference would be a function that requires an argument be an instance of a subclass of Measureable, as opposed to a function that requires an argument understand the messages lessThan and equal. The former is characterizing an object by its class, the latter is characterizing an object by its behavior. Both forms of type checking are found in object-oriented languages.

8.4 Overriding and Virtual Methods

In Chapter 1 we noted that child classes may sometimes find it necessary to *override* the behavior they would otherwise inherit from their parent classes. In syntactic terms, what this means is that a child class will define a method using the same name and type signature as one found in the parent class. When overriding is combined with substitution, we have the situation where a variable is declared as one class, but holds a value from a child class, and a method matching a given message is found in both classes. In almost all cases what we want to have happen in this situation is to execute the method found in the child class, ignoring the method from the parent class.

[0] Section headings followed by an asterisk indicate optional material.

C++	class GraphicalObject {
	public:
	    virtual void draw();
	};
	
	class Ball : public Graphicalobject {
	public:
	    virtual void draw(); // virtual optional here
	};

C#	class GraphicalObject {
	    public viritual void draw () { ... }
	}
	
	class Ball : Graphical Object {
	    public override void draw () { ... }
	}

Delphi	type
	    GraphicalObject = class (TObject)
	       ...
	       procedure draw; virtual;
	    end;
	
	    Ball = class (GraphicalObject)
	       ...
	      procedure draw; override;
	    end;

Object Pascal	type
	    GraphicalObject = object
	       ...
	       procedure draw;
	    end;
	
	    Ball = object (GraphicalObject)
	       ...
	       procedure draw; override;
	    end;

Figure 8.2: Overriding in Various Languages

In many object oriented languages (Smalltalk, Java) this desired behavior will occur naturally, as soon as a child class overrides a method in the parent class using the same type signature. Some languages, on the other hand, require the programmer to indicate that such a substitution is permitted. Many languages use the keyword virtual to indicate this. It may be necessary, as in C++, to place the keyword in the parent class[1] (indicating that overriding *may* take place; it does not indicate that it necessarily will take place) or, as in Object Pascal, in the child class (indicating that overridng *has* taken place). Or it may be required in both places, as in C# and Delphi. Figure 8.2 shows the syntax used for overriding in various langauges.

8.5 Interfaces and Abstract Classes

In Chapter 4 we briefly introduced the concept of an interface in Java and other languages. As with classes, interfaces are allowed to inherit from other interfaces and are even permitted to inherit from multiple parent interfaces. Although the specification that a new class inherits from a parent class and the specification that it implements an interface are not exactly the same, they are sufficiently similar that we will henceforth use the term *inheritance* to indicate both actions.

Several object-oriented languages support an idea, termed an *abstract method* that is mid-way between classes and interfaces. In Java and C#, for example, a class can define one or more methods using the keyword abstract. No body is then provided for the method. A child class *must* implement any abstract methods before an instance of the class can be created. Thus, abstract methods specify behavior in the parent class but the behavior itself must be provided by the child class:

```
abstract class Window {
    ...
    abstract public void paint ();  // draw contents of window
    ...
}
```

An entire class can be named as abstract, whether or not it includes any abstract methods. It is not legal to create an instance of an abstract class, it is only legal to use it as a parent class for purposes of inheritance.

In C++ the idea of an abstract method is termed a *pure virtual method*, and is indicated using the assignment operator:

```
class Window {
public:
    ...
```

[1] Virtual overriding in C++ is actually more complex, for reasons we will develop in the next several chapters.

```
    virtual void paint () = 0; // assignment makes it pure virtual
};
```

A class can have both abstract (or pure virtual) methods and non-abstract methods. A class in which all methods were declared as abstract (or pure virtual) would correspond to the Java idea of an interface.

Abstract methods can be simulated even when the language does not provide explicit support for the concept. In Smalltalk, for example, programmers frequently define a method so as to generate an error if it is invoked, with the expectation that it will be overwritten in child classes:

```
    writeTo: stream
        ↑ self error: 'subclass must override writeTo'
```

This is not exactly the same as a true abstract method, since it does not preclude the creation of instances of the class. Nevertheless, if an instance is created and this method invoked the program will quickly fail, and hence such errors are easily detected.

8.6 Forms of Inheritance

Inheritance is used in a surprising variety of ways. In this section we will describe a few of its more common uses. Note that the following list represents general abstract categories and is not intended to be exhaustive. Furthermore, it sometimes happens that two or more descriptions are applicable to a single situation, because some methods in a single class use inheritance in one way while others use it in another.

8.6.1 Subclassing for Specialization (Subtyping)

Probably the most common use of inheritance and subclassing is for specialization. In subclassing for specialization, the new class is a specialized form of the parent class but satisfies the specifications of the parent in all relevant respects. Thus, in this form the principle of substitution is explicitly upheld. Along with the following category (subclassing for specification) this is the most ideal form of inheritance, and something that a good design should strive for.

Here is an example of subclassing for specialization. A class Window provides general windowing operations (moving, resizing, iconification, and so on). A specialized subclass TextEditWindow inherits the window operations and *in addition*, provides facilities that allow the window to display textual material and the user to edit the text values. Because the text edit window satisfies all the properties we expect of a window in general (thus, a TextEditWindow window is a subtype of Window in addition to being a subclass), we recognize this situation as an example of subclassing for specialization.

8.6.2 Subclassing for Specification

Another frequent use for inheritance is to guarantee that classes maintain a certain common interface–that is, they implement the same methods. The parent class can be a combination of implemented operations and operations that are deferred to the child classes. Often, there is no interface change of any sort between the parent class and the child class–the child merely implements behavior described, but not implemented, in the parent.

This is in essence a special case of subclassing for specialization, except that the subclasses are not refinements of an existing type but rather realizations of an incomplete abstract specification. In such cases the parent class is sometimes known as an *abstract specification class*.

A class that implements an interface is always fulfilling this form of inheritance. However, subclassing for specification can also arise in other ways. In the billiards simulation example presented in Chapter 7, for example, the class GraphicalObject was an abstract class since it described, but did not implement, the methods for drawing the object and responding to a hit by a ball. The subsequent classes Ball, Wall, and Hole then used subclassing for specification when they provided meanings for these methods.

In general, subclassing for specification can be recognized when the parent class does not implement actual behavior but merely defines the behavior that will be implemented in child classes.

8.6.3 Subclassing for Construction

A class can often inherit almost all of its desired functionality from a parent class perhaps changing only the names of the methods used to interface to the class, or modifying the arguments in a certain fashion. This may be true even if the new class and the parent class fail to share the *is-a* relationship.

For example, the Smalltalk class hierarchy implements a generalization of an array called Dictionary. A dictionary is a collection of key-value pairs, like an array, but the keys can be arbitrary values. A *symbol table*, such as might be used in a compiler, can be considered a dictionary indexed by symbol names in which the values have a fixed format (the symbol-table entry record). A class SymbolTable can therefore be made a subclass of the class Dictionary, with new methods defined that are specific to the use as a symbol table. Another example might be forming a *set* data abstraction on top of a base class which provides *list* methods. In both these cases, the child class is not a more specialized form of the parent class, because we would never think of substituting an instance of the child class in a situation where an instance of the parent class is being used.

A common use of subclassing for construction occurs when classes are created to write values to a binary file, for example, in a persistent storage system. A parent class may implement only the ability to write raw binary data. A subclass is constructed for every structure that is saved. The subclass implements a save procedure for the data type, which uses the behavior of the parent type to do the actual storage.[2]

```
class Storable {
    void writeByte(unsigned char);
    };

class StoreMyStruct : public Storable {
    void writeStruct (MyStruct & aStruct);
    };
```

Subclassing for construction tends to be frowned upon in statically typed languages, since it often directly breaks the principle of substitution (forming subclasses that are not subtypes). On the other hand, because it is often a fast and easy route to developing new data abstractions, it is widely employed in dynamically typed languages. Many instances of subclassing for construction can be found in the Smalltalk standard library.

We will investigate an example of subclassing for construction in Chapter 9. We will also see that C++ provides an interesting mechanism, *private inheritance*, which permits subclassing for construction without breaking the principle of substitution.

8.6.4 Subclassing for Generalization

Using inheritance to subclass for generalization is, in a certain sense, the opposite of subclassing for specialization. Here, a subclass extends the behavior of the parent class to create a more general kind of object. Subclassing for generalization is often applicable when we build on a base of existing classes that we do not wish to modify, or cannot modify.

Consider a graphics display system in which a class Window has been defined for displaying on a simple black-and-white background. You could create a subtype ColoredWindow that lets the background color be something other than white by adding an additional field to store the color and overriding the inherited window display code that specifies the background be drawn in that color.

Subclassing for generalization frequently occurs when the overall design is based primarily on data values and only secondarily on behavior. This is shown in the colored window example, since a colored window contains data fields that are not necessary in the simple window case.

As a rule, subclassing for generalization should be avoided in favor of inverting the type hierarchy and using subclassing for specialization. However, this is not always possible.

[2] This example illustrates the blurred lines between categories. If the child class implements the storage using a different method name, we say it is subclassing for construction. If, on the other hand, the child class uses the same name as the parent class, we might say the result is subclassing for specification.

8.6.5 Subclassing for Extension

While subclassing for generalization modifies or expands on the existing functionality of an object, subclassing for extension adds totally new abilities. Subclassing for extension can be distinguished from subclassing for generalization in that the latter must override at least one method from the parent and the functionality is tied to that of the parent. Extension simply adds new methods to those of the parent, and the functionality is less strongly tied to the existing methods of the parent.

An example of subclassing for extension is a StringSet class that inherits from a generic Set class but is specialized for holding string values. Such a class might provide additional methods for string-related operations–for example, "search by prefix," which returns a subset of all the elements of the set that begin with a certain string value. These operations are meaningful for the subclass, but are not particularly relevant to the parent class.

As the functionality of the parent remains available and untouched, subclassing for extension does not contravene the principle of substitution and so such subclasses are always subtypes.

8.6.6 Subclassing for Limitation

Subclassing for limitation occurs when the behavior of the subclass is smaller or more restrictive than the behavior of the parent class. Like subclassing for generalization, subclassing for limitation occurs most frequently when a programmer is building on a base of existing classes that should not, or cannot, be modified.

For example, an existing class library provides a double-ended-queue, or *deque*, data structure. Elements can be added or removed from either end of the deque, but the programmer wishes to write a stack class, enforcing the property that elements can be added or removed from only one end of the stack.

In a manner similar to subclassing for construction, the programmer can make the Stack class a subclass of the existing Deque class, and can modify or override the undesired methods so that they produce an error message if used. These methods override existing methods and eliminate their functionality, which characterizes subclassing for limitation. (Overriding, by which a subclass changes the meaning of a method defined in a parent class, will be discussed in a subsequent chapter).

Because subclassing for limitation is an explicit contravention of the principle of substitution, and because it builds subclasses that are not subtypes, it should be avoided whenever possible.

8.6.7 Subclassing for Variance

Subclassing for variance is employed when two or more classes have similar implementations but do not seem to possess any hierarchical relationships between the abstract concepts represented by the classes. The code necessary to control a mouse, for example, may be nearly identical to the code required to control a graphics tablet. Conceptually, however, there is no reason why class Mouse should be made a subclass of class Tablet, or the other way. One of the two classes is then arbitrarily selected to be the parent, with the common code being inherited by the other and device-specific code being overridden.

Usually, however, a better alternative is to factor out the common code into an abstract class, say PointingDevice, and to have both classes inherit from this common ancestor. As with subclassing for generalization, this choice may not be available if you are building on a base of existing classes.

8.6.8 Subclassing for Combination

A common situation is a subclass that represents a *combination* of features from two or more parent classes. A teaching assistant, for example, may have characteristics of both a teacher and a student, and can therefore logically behave as both. The ability of a class to inherit from two or more parent classes is known as *multiple inheritance*; it is sufficiently subtle and complex that we will devote an entire chapter to the concept.

8.6.9 Summary of the Forms of Inheritance

We can summarize the various forms of inheritance by the following table:

- Specialization. The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class.

- Specification. The parent class defines behavior that is implemented in the child class but not in the parent class.

- Construction. The child class makes use of the behavior provided by the parent class, but is not a subtype of the parent class.

- Generalization. The child class modifies or overrides some of the methods of the parent class.

- Extension. The child class adds new functionality to the parent class, but does not change any inherited behavior.

- Limitation. The child class restricts the use of some of the behavior inherited from the parent class.

- Variance. The child class and parent class are variants of each other, and the class-subclass relationship is arbitrary.

- Combination. The child class inherits features from more than one parent class. This is multiple inheritance and will be the subject of a later chapter.

8.7 Variations on Inheritance*

In this section we will examine a number of mostly single-language specific variations on the themes of inheritance and overriding.

8.7.1 Anonymous Classes in Java

Occasionally a situation arises where a programmer needs to create a simple class, and knows there will never be more than one instance of the class. Such an object is often termed a *singleton.* The Java programming language provides a mechanism for creating such an object without even having to give a name to the class being used to define the object. Hence the name for this technique, *anonymous classes.*

In order to be able to create an anonymous class, several requirements must be met:

1. Only one instance of the anonymous class can be created.
2. The class must inherit from a parent class or interface, and not require a constructor for initialization.

These two conditions frequently arise in the context of user interfaces. For example, in Chapter 22 we will encounter a class named ButtonAdapter that is used to create graphical buttons. To give behavior to a button, the programmer must form a new class that inherits from ButtonAdapter, and overrides the method pressed. Since there is only one such object, this can be done with an anonymous class (also sometimes termed a *class definition expression*).

Graphical elements are added to a window using the method add. To place a new button in a window, all that is necessary is the following:

```
Window p = ...;

p.add (new ButtonAdapter("Quit"){
    public void pressed () { System.exit(0); }
    }
);
```

Study carefully the argument being passed to the add operator. It includes the creation of a new value, indicated by the new operator. But rather than ending the expression with the closing parenthesis on the argument list for new, a curly brace appears as if in a class definition. In fact, this is a new class definition. A subclass of ButtonAdapter is being formed, and a single instance of this class will be created. Any methods required by this new class are given immediately in-line. In this case, the new class overrides the method named pressed. The closing curly brace terminates the anonymous class expression.

[0] Section headings followed by an asterisk indicate optional material.

8.7.2 Inheritance and Constructors

A constructor, you will recall, is a procedure that is invoked implicitly during the creation of a new object value, and which guarantees that the newly created object is properly initialized. Inheritance complicates this process, since both the parent and the new child class may have initialization code to perform. Thus code from both classes must be executed.

In Java, C++ and other languages the constructor for both parent and child will automatically be executed as long as the parent constructor does not require additional parameters. When the parent does require parameters, the child must explicitly provide them. In Java this is done using the keyword super:

```
class Child extends Parent {
    public Child (int x) {
        super (x + 2); // invoke parent constructor
    ...
    }
}
```

In C++ the same task is accomplished by writing the parent class name in the form of an initializer:

```
class Child : public Parent {
public:
    Child (int x) : Parent(x+2) { ... }
};
```

In Delphi a constructor for a child class must always invoke the constructor for the parent class, even if the parent class constructor takes no arguments. The syntax is the same for executing the parent class behavior in any overridden method:

```
constructor TChildClass.Create;
begin
    inherited Create; // execute constructor in parent
end
```

Arguments to the parent constructor are added as part of the call:

```
constructor TChildClass.Create (x : Integer);
begin
    inherited Create(x + 2);
end
```

Similarly, an initialization method in Python does not automatically invoke the function in the parent, hence the programmer must not forget to do this task:

```
class Child(Parent):
    def __init__ (self):
        # first initialize parent
        Parent.__init__(self)
        # then do our initialization
        ...
```

8.7.3 Virtual Destructors

Recall from Chapter 5 that in C++ a destructor is a function that will be invoked just before the memory for a variable is recovered. Destructors are used to perform whatever tasks are necessary to ensure a value is properly deleted. For example, a destructor will frequently free any dynamically allocated memory the variable may hold.

If substitution and overriding are anticipated, then it is important that the destructor be declared as virtual. Failure to do so may result in destructors for child classes not being invoked. This following example shows this error:

```
class Parent {
public:
        // warning, destructor not declared virtual
    ~Parent () { cout << "in parent\n"; }
};

class Child : public Parent {
public:
    ~Child () { cout << "in child\n"; }
};
```

If an instance of the child class is held by a pointer to the parent class and subsequently released (say, by a delete statement), then only the parent destructor will be invoked.

```
    Parent * p = new Child();
    delete p;
in parent
```

If the parent destructor is declared as virtual, then both the parent and child destructors will be executed. In C++ it is a good idea to include a virtual destructor, even if it performs no action, if there is any possibility that a class may later be subclassed.

8.8 The Benefits of Inheritance

In this section we will describe some of the many important benefits of the proper use of inheritance.

8.8.1 Software Reusability

When behavior is inherited from another class, the code that provides that behavior does not have to be rewritten. This may seem obvious, but the implications are important. Many programmers spend much of their time rewriting code they have written many times before–for example, to search for a pattern in a string or to insert a new element into a table. With object-oriented techniques, these functions can be written once and reused.

Other benefits of reusable code include increased reliability (the more situations in which code is used, the greater the opportunities for discovering errors) and the decreased maintenance cost because of sharing by all users of the code.

8.8.2 Code Sharing

Code sharing can occur on several levels with object-oriented techniques. On one level, many users or projects can use the same classes. (Brad Cox [Cox 1986] calls these software-ICs, in analogy to the integrated circuits used in hardware design). Another form of sharing occurs when two or more classes developed by a single programmer as part of a project inherit from a single parent class. For example, a Set and an Array may both be considered a form of Collection. When this happens, two or more types of objects will share the code that they inherit. This code needs to be written only once and will contribute only once to the size of the resulting program.

8.8.3 Consistency of Interface

When two or more classes inherit from the same superclass, we are assured that the behavior they inherit will be the same in all cases. Thus, it is easier to guarantee that interfaces to similar objects are in fact similar, and that the user is not presented with a confusing collection of objects that are almost the same but behave, and are interacted with, very differently.

8.8.4 Software Components

In Chapter 1, we noted that inheritance provides programmers with the ability to construct reusable software components. The goal is to permit the development of new and novel applications that nevertheless require little or no actual coding.

Already, several such libraries are commercially available, and we can expect many more specialized systems to appear in time.

8.8.5 Rapid Prototyping

When a software system is constructed largely out of reusable components, development time can be concentrated on understanding the new and unusual portion of the system. Thus, software systems can be generated more quickly and easily, leading to a style of programming known as *rapid prototyping* or *exploratory programming*. A prototype system is developed, users experiment with it, a second system is produced that is based on experience with the first, further experimentation takes place, and so on for several iterations. Such programming is particularly useful in situations where the goals and requirements of the system are only vaguely understood when the project begins.

8.8.6 Polymorphism and Frameworks

Software produced conventionally is generally written from the bottom up, although it may be *designed* from the top down. That is, the lower-level routines are written, and on top of these slightly higher abstractions are produced, and on top of these even more abstract elements are generated. This process is like building a wall, where every brick must be laid on top of an already laid brick.

Normally, code portability decreases as one moves up the levels of abstraction. That is, the lowest-level routines may be used in several different projects, and perhaps even the next level of abstraction may be reused, but the higher-level routines are intimately tied to a particular application. The lower-level pieces can be carried to a new system and generally make sense standing on their own; the higher-level components generally make sense (because of declarations or data dependencies) only when they are built on top of specific lower-level units.

Polymorphism in programming languages permits the programmer to generate high-level reusable components that can be tailored to fit different applications by changes in their low-level parts. We will have much more to say about this topic in subsequent chapters.

8.8.7 Information Hiding

A programmer who reuses a software component needs only to understand the nature of the component and its interface. It is not necessary for the programmer to have detailed information concerning matters such as the techniques used to implement the component. Thus, the interconnectedness between software systems is reduced. We earlier identified the interconnected nature of conventional software as being one of the principle causes of software complexity.

8.9 The Costs of Inheritance

Although the benefits of inheritance in object-oriented programming are great, almost nothing is without cost of one sort or another. For this reason, we must consider the cost of object-oriented programming techniques, and in particular the cost of inheritance.

8.9.1 Execution Speed

It is seldom possible for general-purpose software tools to be as fast as carefully hand-crafted systems. Thus, inherited methods, which must deal with arbitrary subclasses, are often slower than specialized code.

Yet, concern about efficiency is often misplaced.[3] First, the difference is often small. Second, the reduction in execution speed may be balanced by an increase in the speed of software development. Finally, most programmers actually have little idea of how execution time is being used in their programs. It is far better to develop a working system, monitor it to discover where execution time is being used, and improve those sections, than to spend an inordinate amount of time worrying about efficiency early in a project.

8.9.2 Program Size

The use of any software library frequently imposes a size penalty not imposed by systems constructed for a specific project. Although this expense may be substantial, as memory costs decrease the size of programs becomes less important. Containing development costs and producing high-quality and error-free code rapidly are now more important than limiting the size of programs.

8.9.3 Message-Passing Overhead

Much has been made of the fact that message passing is by nature a more costly operation than simple procedure invocation. As with overall execution speed, however, overconcern about the cost of message passing is frequently penny-wise and pound-foolish. For one thing, the increased cost is often marginal–perhaps two or three additional assembly-language instructions and a total time penalty of 10 percent. (Timing figures vary from language to language. The overhead of message passing will be much higher in dynamically bound languages, such as Smalltalk, and much lower in statically bound languages, such as C++.) This increased cost, like others, must be weighed against the many benefits of the object-oriented technique.

A few languages, notably C++, make a number of options available to the programmer that can reduce the message-passing overhead. These include elim-

[3] The following quote from an article by Bill Wulf offers some apt remarks on the importance of efficiency: "More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason–including blind stupidity" [Wulf 1972].

inating the polymorphism from message passing (qualifying invocations of member functions by a class name, in C++ terms) and expanding inline procedures. Similarly, the Delphi Pascal programmer can choose dynamic methods, which use a run-time lookup mechanism, or virtual methods, which use a slightly faster technique. Dynamic methods are inherently slower, but require less space.

8.9.4 Program Complexity

Although object-oriented programming is often touted as a solution to software complexity, in fact, overuse of inheritance can often simply replace one form of complexity with another. Understanding the control flow of a program that uses inheritance may require several multiple scans up and down the inheritance graph. This is what is known as the *yo-yo* problem, which we will discuss in more detail in a later chapter.

Chapter Summary

In this chapter we begin a detailed examination of inheritance and substitution, a topic that will be continued through the next several chapters. When a child class declares that it inherits from a parent class, code in the parent class does not have to be rewritten. Thus inheritance is a powerful mechanism of code reuse. But this is not the only reason to use inheritance. In the abstract, a child class is a representative of the category formed by the parent class, and hence it makes sense that an instance of the child class could be used in those situations where we expect an instance of the parent class. This is known as the principle of *substitution*. But this is only an idealization. Not all types of inheritance support this idea behavior.

We have described various forms of inheritance, noting when they seem to support substitution and when they may not.

The chapter concludes with descriptions of both the benefits of inheritance and the costs incurred through the use of the technique.

Further Reading

Many of the ideas introduced in this chapter will be developed and explored in more detail in subsequent chapters. Overriding is discussed in detail in Chapter 16. We will discuss static and dynamic typing more in Chapter 10, and polymorphism in more detail in Chapter 14.

In Section 8.1.2 we noted that inheritance is used both as a mechanism of code reuse and concept reuse. The fact that the same feature is serving two different purposes is a frequent criticism levied against object-oriented languages. Many writers have advocated separating these two tasks, for example using inheritance of classes only for code reuse, and using inheritance of interfaces (as, for example, in Java) for substitution (concept reuse). While this approach has a theoretical appeal, from a practical standpoint it complicates the task of programming, and has not been widely adopted. See Exercise 5 for one way this could be accomplished.

The list describing the forms of inheritance is adopted from [Halbert 1987], although I have added some new categories of my own. The editable-window example is from [Meyer 1988a].

The principle of substitution is sometimes referred to as the *Liskov Substitution Principle*, since an early discussion of the idea was presented by Barbara Liskov and John Guttag [Liskov 1986].

Self Study Questions

1. In what ways is a child class an extension of its parent? In what ways is it a contraction?

2. What is the is-a test for inheritance?

3. What are the two major reasons for the use of inheritance?

4. What is the principle of substitution? What is the argument used to justify its application?

5. How is a class that contains abstract methods similar to an interface? If not all methods are abstract, how is it different?

6. What features characterize each of the following forms of inheritance?

   (a) Subclassing for Specialization
   (b) Subclassing for Specification
   (c) Subclassing for Construction
   (d) Subclassing for Generalization
   (e) Subclassing for Extension
   (f) Subclassing for Limitation
   (g) Subclassing for Variance

7. Why is subclassing for construction not normally considered to be a good idea?

8. Why is subclassing for limitation not a good idea?

9. How does inheritance facilitate software reuse?

10. How does it encourage consistency of interface?

11. How does it support the idea of rapid prototyping?

12. How does it encourage the principle of information hiding?

13. An anonymous class combines what two activities?

14. Why is the execution time cost incurred by the use of inheritance not usually important? What are some situations where it would be important?

Exercises

1. Suppose you were required to program a project in a non-object oriented language, such as Pascal or C. How would you simulate the notion of classes and methods? How would you simulate inheritance? Could you support multiple inheritance? Explain your answer.

2. We noted that the execution overhead associated with message passing is typically greater than the overhead associated with a conventional procedure call. How might you measure these overheads? For a language that supports both classes and procedures (such as C++ or Object Pascal), devise an experiment to determine the actual performance penalty of message passing.

3. Consider the three geometric concepts of a line (infinite in both directions), a ray (fixed at a point, infinite in one direction), and a segment (a portion of a line with fixed end points). How might you structure classes representing these three concepts in an inheritance hierarchy? Would your answer differ if you concentrated more on the data representation or more on the behavior? Characterize the type of inheritance you would use. Explain the reasoning behind your design.

4. The following appeared as an illustration of inheritance in a popular journal:

   > "Perhaps the most powerful concept in object-oriented programming systems is inheritance. Objects can be created by inheriting the properties of other objects, thus removing the need to write any code whatsoever! Suppose, for example, a program is to process complex numbers consisting of real and imaginary parts. In a complex number, the real and imaginary parts behave like real numbers, so all of the operations (+, -, /, *, sqrt, sin, cos, etc.) can be inherited from the class of objects call REAL, instead of having to be written in code. This has a major impact on programmer productivity."

   (a) The quote seems to indicate that class Complex could be a child class of Real. Does the assertion that the child class Complex need not write any code seem plausible?

   (b) Does this organization make sense in terms of the data members each class must maintain? Why or why not?

(c) Does this organization make sense in terms of the methods each class must support? Why or why not?

(d) Can you describe a better approach for creating a class Complex using an existing class Real? What benefit does your new class derive from the existing class?

5. In Section 8.1.2 we noted how inheritance is used for two different purposes; as a vehicle for code reuse, and a vehicle for substitution. Among the major object-oriented langauges, Java comes closest to separating these two purposes, since the language supports both classes and interfaces. But it confuses the two topics by continuing to allow substitution for class values. Suppose we took the next step, and changed the Java language to eliminate substitution for class types. This could be accomplished by making the following two modifications to the language:

   - A variable declared as a class could hold values of the class, but not of child classes.
   - If a parent class indicates that it supports an interface, the child class would not automatically support the interface, but would have to explicitly indicate this fact in its class heading.

   We maintain inheritance and substitution of interfaces; a variable declared as an interface could hold a value from any class that implemented the interface.

   (a) Show that any class hierarchy, and any currently legal assignment, could be rewritten in this new framework. (You will need to introduce new interfaces).

   (b) Although the resulting system is much cleaner from a theoretical standpoint, what has been lost? Why did the designers of Java not follow this approach?