Chapter 7

Inheritance

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 the previous two chapters.

The *next* step in learning how to program in an object-oriented manner involves acquiring the skill of 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).

7.1 An Intuitive Description of Inheritance

Let us return to Flo the florist from the first chapter. There is a certain behavior we expect florists to perform, not because they are florists but simply because they are shopkeepers. For example, we expect Flo to request money for the transaction, and in turn expect to be given a receipt. These activities are not unique to florists, but rather are common to bakers, grocers, stationers, toystore owners, car dealers, and other merchants. Mentally, 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 strictly larger set) of the properties associated with parent classes. A subclass must have all the properties of the parent class, and others 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 for much of the 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, in turn, a subclass of class `Animal`, then `Dog` will inherit attributes both from `Mammal` and from `Animal`.

A complicating factor in our intuitive description of inheritance is the fact that subclasses can *override* behavior inherited from parent classes. For example, the class `Platypus` overrides the reproduction behavior inherited from class `Mammal`, since platyupus's lay eggs. We will briefly mention the mechanics of overriding in this chapter, then return to a more detailed discussion of the semantics of overriding in Chapter 11.

7.2 Subclass, Subtype, and Substitutability

Consider the relationship of the data type associated with a parent class to the data type associated with a derived, or child class. The following argument can be made:

- Instances of the subclass must possess all data areas associated with the parent class.
- Instances of the subclass 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 argument is not always valid. 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 substitutability.*

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

As we will see in Chapter 10, the term *subtype* is often used to denote a subclass relationship in which the principle of substitutability is maintained, in order to distinguish such forms from the general *subclass* relationship, which may or may not satisfy this principle.

We have already seen a use of the principle of substitutability in the previous chapter. Section 6.4 described the following procedure:

```
procedure drawBoard;
```

```
var
    gptr : GraphicalObject;
begin
    SetPort (theWindow);

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

The global variable `listOfObjects` maintained a list of graphical objects, which could be any of three different 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 different subclasses. Sometimes `gptr` holds a ball, sometimes a hole, and sometimes a wall. In each case, when the `draw` function is invoked, it is the correct method for the current value of `gptr` that will be executed, and 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, that the subclasses must also be subtypes.

7.2.1 Subtypes and Strong Typing

It should be noted that much more emphasis is placed on the principle of substitutability in statically typed languages (such as C++ and Object Pascal) than in 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 more by their behavior. For example, a polymorphic function (that is, a function that can take objects of various different classes) in a statically typed language can only ensure a certain level of functionality 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. We will discuss static and dynamic typing more in Chapter 10, and polymorphism in more detail in Chapter 14.

7.3 Forms of Inheritance

The mechanism of inheritance is used in a surprisingly large variety of ways. In this section we will describe a few of the more common uses. Note that the following list represents general abstract categories, and are not intended to be exhaustive. Furthermore, it will sometime happen that two or more descriptions may be applicable to a single situation, as some methods in a single class use inheritance in one fashion while other methods in the same class use the same mechanism for a different purpose.

In reading the following list,[1] pay careful attention to which uses of inheritance support the subtyping relationship, and which do not.

7.3.1 Subclassing for Specialization (Subtyping)

Probably the most common use of inheritance and subclassing is for the purposes of specialization. In subclassing for specialization, the new class is a specialized form of the parent class, but satisfies in all relevant respects the specifications of the parent. Thus, in this form the principle of substitutability 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 towards.

An example of subclassing for specialization might occur if you have a class `Window` which 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 allow the user to edit such text values. Because the text edit window satisfies all the properties we expect of a window in general (and thus, a `TextEditWindow` window is a subtype of a `Window` in addition to being a subclass), we recognize this situation as an example of subclassing for specialization.

7.3.2 Subclassing for Specification

Another frequent use for inheritance is to guarantee that classes maintain a certain common interface; that is, that 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 actually a special case of subclassing for specialization, except that the subclasses are not refinements of an existing type, but rather are realizations of an incomplete abstract specification. In such cases the parent class is sometimes known as an *abstract specification class*.

In the Billards simulation example presented in the previous chapter, 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.

[1] The categories described here are adopted from [Halbert 87], although I have added some new categories of my own. The editable-window example is from [Meyer 88a].

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. This form of subclassing is common in languages that make extensive use of *virtual functions*, a topic we will subsequently discuss in more detail.

7.3.3 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 a `Dictionary`. A dictionary is a collection of key-value pairs, like an array, only the keys can be arbitrary values. A *symbol table*, such as might be used in a compiler, can be considered to be a dictionary indexed by symbol names in which the values have a fixed format (the symbol-table entry record). A class `SymbolTable` could therefore be made a subclass of the class `Dictionary`, with new methods being 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. Note that, in both these cases, the child class is not a more specialized form of the parent class, in as much as we would never think of substituting an instance of the child class in a situation where an instance of the parent class was being used.

A common use of subclassing for construction occurs when classes are created to write values to a binary file, as for example in a persistant storage system. A parent class may implement only the ability to write raw binary data. A subclass is constructed for every structure that will be 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 substitutability (and thus, forms subclasses that are not subtypes). On the other hand, because it is often a fast and easy route to developing new

[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.

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 the next chapter. We will also see that the C++ language provides an interesting mechanism, named *private inheritance*, which permits using subclassing for construction without breaking the principle of substitutability.

7.3.4 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. Subtyping for generalization is often applicable when you are building on a base of existing classes that you do not wish to modify, or cannot modify.

For example, 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 that the background is drawn in that color.

Subtyping for generalization will frequently occur when the overall design is based first on a consideration of data values, and only secondarily on questions of behavior. This is shown in the colored window example, since a colored window will contain 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 subtyping for specialization. However, this is not always possible.

7.3.5 Subclassing for Extension

While subclassing for generalization modifies or expands on 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, whereas 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 might occur if it was desired to maintain instances of a certain class in a list. To do this a new class could be constructed which is subclassed from the original (thus providing all the behavior of the original) and in addition provided the link fields necessary for constructing the linked list.

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

7.3.6 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.

Suppose, for example, that 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. The programmer, however, 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. It is the presence of these methods that override existing methods, and eliminate their functionality, which characterizes subclassing for limitation. (The process of 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 substitutability, and builds subclasses which are not subtypes, it should be avoided whenever possible.

7.3.7 Subclassing for Variance

Subclassing for variance is being 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 particular reason why either class `Mouse` should be made a subclass of class `Tablet`, or the other way around. 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.

7.3.8 Subclassing for Combination

One common situation occurs when a subclass represents a *combination* of features from two or more parent classes. A TeachingAssistant, for example, may have characteristics which are representative of both a Teacher and a Student, and could therefore logically behave as both. The ability of a class to inherit from two or more parent classes is known as *multiple* *inheritance*, and is sufficiently subtle and complex that we will subsequently devote an entire chapter to the concept.

7.3.9 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, or in other words, the child class is a subtype of the parent class.
- **Specification**. The parent class defines behavior which 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.

7.4 Inheritance in Various Languages

In the following sections we will describe in detail the syntax used to describe inheritance in each of the various languages we are considering. Among the features the reader should note are differences between languages which require all classes to be subclasses of a common parent class (often named `Object`, as in Smalltalk and Objective-C), versus languages which permit a number of different independent class hierarchies.

The advantage of the single inheritance structure is that the functionality provided in the root of the tree, in class `Object`, is inherited by all objects. Thus, every object is guaranteed to possess a common minimal level of functionality. On the other hand, it has the disadvantage of combining together all classes into a tightly coupled unit.

The advantage of having several independent inheritance hierarchies is that an application is not forced to carry along with it a large library of classes, only a few of which may be used in any one program. The disadvantage, of course, mirrors the advantage of the alternative. Namely, there is no programmer-defined functionality that *all* objects are guaranteed to possess.

In part, the differing views of objects are yet one more example of the distinction between languages that use dynamic typing and those that employ static typing (a topic we will return to in Chapter 12). 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.

7.4.1 Inheritance in Object Pascal

In Apple Object Pascal, inheritance from a parent class is indicted by placing the name of the parent class inside a parenthesis after the `object` keyword. For example, suppose we had decided to make the classes `Ball`, `Wall` and `Hole` in our billiard simulation inherit from a common `GraphicalObject` class. We would have indicated this as shown in Figure 7.1

As illustrated by this example, child classes can add both new data fields and new behavior. They can also override existing behavior, by redefining methods. This is indicated by the keyword `override`, as in the `draw` method. The arguments to an overridden method must match in type and number the arguments for the parent class.

The Delphi version of Object Pascal differs in two important respects. First, as we have seen already in earlier chapters, the keyword `class` is used in place of the keyword `object` in the class definition, and classes must always inherit from another class. The class `TObject` is ultimately the root class for all objects. Second, in addition to the keyword `override`, the keyword `virtual` is appended in the parent class to those methods that are allowed to be overridden, in much the same fashion as in C++ (to be described shortly). An example illustrating these changes in shown in Figure 7.2. (Omitting the `override` is a common source of error, since the declaration is legal but the interpetation is incorrect. We will discuss this in more detail in Chapter 10.)

A more radical difference between Delphi Pascal and Apple Object Pascal is the introduction of `dynamic` methods. A dynamic method uses a different run-time lookup mechanism (closer to that used by Objective-C than to C++, see Chapter 21 for an explanation of these mechanisms). This makes dynamic methods slightly slower than virtual methods, but they also use less space. The keyword `dynamic` in place of `virtual` is an indication that a dynamic method is being declared. Many of the methods associated with operating-system windows-related activity is implemented using dynamic methods. The term *message* is often restricted to mean only these windows-related actions.

```
type
    GraphicalObject = object
            (* data fields *)
        region : Rect;
        link : GraphicalObject;
            (* operations *)
        procedure draw;
        procedure update;
        procedure hitBy (aBall : Ball);
    end;

    Ball = object (GraphicalObject)
            (* data fields *)
        direction : real;
        energy : real;

            (* initialization *)
        procedure initialize (x, y : integer);

            (* overridden methods *)
        procedure draw; override;
        procedure update; override;
        procedure hitBy (aBall : Ball); override;

            (* ball specific methods *)
        procedure erase;
        procedure setCenter (newx, newy : integer);
        function x : integer;
        function y :integer;
    end;
```

Figure 7.1: An example of inheritance in Apple Object Pascal

```
type
    GraphicalObject = class (TObject)
            (* data fields *)
        region : Rect;
        link : GraphicalObject;
            (* operations *)
        procedure draw; virtual;
        procedure update; virtual;
        procedure hitBy (aBall : Ball); virtual;
    end;

    Ball = class (GraphicalObject)
            (* data fields *)
        direction : real;
        energy : real;

            (* initialization *)
        procedure initialize (x, y : integer);

            (* overridden methods *)
        procedure draw; override;
        procedure update; override;
        procedure hitBy (aBall : Ball); override;

            (* ball specific methods *)
        procedure erase;
        procedure setCenter (newx, newy : integer);
        function x : integer;
        function y :integer;
    end;
```

Figure 7.2: An example of inheritance in Delphi Pascal

7.4.2 Inheritance in Smalltalk

As we noted in Chapter 3, in Smalltalk inheritance is an intrinsic part of creating new classes. A new class cannot be defined without first specifying an existing class from which it will inherit. In fact, the creation of a new class is performed through the act of sending a *message* to the parent class from which the new class will inherit.

```
List subclass: #Set
    instanceVariables: #( )
    classVariables: #( )
```

There is a single ancestor class, named `Object`, from which all other classes ultimately descend. Class `Object` provides all objects with a consistent and common functionality. Examples of methods provided by this class include the ability to compare one object against another, the ability to print a string representation of an object, and so on.

Only single inheritance is supported by Smalltalk. That is, each class inherits from only one parent class. A method can override a method from a parent class by simply being named the same.

7.4.3 Inheritance in Objective-C

As in Smalltalk, in Objective-C inheritance is an intrinsic part of forming a new class. The interface description of every new class must define the parent class from which the new class will inherit. The following, for example, indicates that class `Card` will inherit from class `Object`.

```
@interface Card : Object
{
 ...
}
...
@end
```

As in Smalltalk, there is a single ancestor class, named `Object`, from which all other classes ultimately descend. Class `Object` provides all objects with a consistent and common functionality, and is frequently used as a parent for new classes.

Only single inheritance is permitted. That is, a class cannot inherit from two or more parent classes. As in Smalltalk, a method having the same name as one found in a parent class will override the inherited method.

7.4.4 Inheritance in C++

In C++, unlike Smalltalk and Objective-C, a new class need not inherit from an existing class. Should inheritance be desired, it is indicated in the class heading by using the keyword `public` followed by the name of the parent class. The following example, from a program similar to the one we will examine in Chapter 8, illustrates this syntax. Here the new class, named `TablePile`, inherits from the more general class `CardPile`, which represents an abstraction for a pile of cards.

```
class TablePile : public CardPile {
  ...
};
```

The keyword `public` can be replaced by the keyword `private` to indicate subclassing for construction – that is, the use of inheritance that does not create a subtype relationship. We will see an example of this in Chapter 11.

As we noted in an earlier chapter, an advantage of object oriented languages is that they tend to tie the process of creation of a new value and the process of initialization for the value. Inheritance complicates this process somewhat, since both the parent class and a new class may have initialization code to perform. To accomplish this, the constructor in the child class must explicitly invoke the constructor in the parent class. This is done as an initialization clause in the constructor for the child class, as in the following:

```
TablePile::TablePile(int x, int y, int c)
    : CardPile(x, y) // initialize parent
{
    // then initialize child class
    ...
}
```

C++ supports multiple inheritance. That is, a new child class can be specified as inheriting from two or more parent classes. We will investigate the implications of this in more detail in a later chapter.

In an earlier chapter we introduced the keywords `private` and `public`, indicating that the one was used to describe the interface for the class while the other described implementation details. A third keyword, `protected`, can be used with classes are formed using inheritance. Protected fields are part of the implementation, but are accessible to subclasses as well as to the implementation of the class itself.

The semantics of overriding in C++ are subtle, and depend upon how the receiver has been declared. We will postpone a discussion of this point until Chapter 11.

7.4.5 Inheritance in Java

Of all the languages we are considering in this book, Java goes the furthest in separating the concepts of subclass and subtype. Subclasses are declared using the keyword `extends`, as in the following example:

```
class window {
    // ...
    };

class textEditWindow extends window {
    // ...
    };
```

Subclasses are assumed to be subtypes (although, as with C++, this assumption need not always be valid). This means that an instance of a subclass can be assigned to a variable declared as the parent class type. Methods in the child class that have the same name as methods in the parent class will override the inherited behavior. As in C++, the keyword `protected` can be used to designate methods and data fields which are accessible only within the class or within subclasses, but not part of the more general interface.

All classes are derived from a single root class, named `Object`. If no parent class is explicitly provided the class `Object` is assumed. Thus, the class declaration for `window` shown above is the same as the following:

```
class window extends Object {
    // ...
    };
```

An alternative form of subtyping involves the concept of an *interface*. An interface defines the protocol for certain behavior, but no implementation. In this respect it is similar to an abstract parent class. An example interface might be the following, used to describe objects that can read from and write to an input/output stream.

```
public interface Storing {
    void writeOut(Stream s);
    void readFrom(Stream s);
    };
```

An interface defines a new type. This means that variables can be declared as instances of an interface. A class can then indicate that it implements the protocol defined by an interface. Instances of the class can be assigned to variables declared as the interface type, just as instances of a child class can be assigned to variables declared using a parent class type.

```
public class BitImage implements Storing {
    void writeOut (Stream s) {
        // ...
        };
    void readFrom (Stream s) {
        // ...
        }:
    };
```

Although Java only support single inheritance (inheritance from only one parent class), a class can indicate it supports multiple different interfaces. Many problems for which multiple inheritance might be used in C++ can be solved in Java using multiple interfaces. Interfaces are allowed to `extend` other interfaces, and are even permitted to `extend` multiple interfaces.

The idea of subclassing for specification is formalized in Java through the use of the `abstract` modifier. If a class is declared as `abstract`, then it must be subclassed. It is not permitted to create an instance of an abstract class, only subclasses. Methods as well can be declared to be `abstract`, and if so need not have an implementation. Thus, defining a class as `abstract` ensures that it will be used only as a specification of behavior, not an implementation.

```
abstract class storable {

    public abstract writeOut();

    }
```

An alternative modifier, `final` indicates that the class (or method) cannot be subclassed or modified. Thus, the user is guaranteed that the behavior of the class will be as defined, and not modified by a later subclass.

```
final class newClass extends oldClass {
    ...
    }
```

7.5 The Benefits of Inheritance

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

7.5.1 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 a large percentage of their time rewriting code that 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. Using 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 has been reused, the greater will have been the opportunities for discovering errors), and the decreased maintenance cost because of sharing by all users of the code.

7.5.2 Code Sharing

Code sharing can occur on several levels using object-oriented techniques. On one level, many separate users or projects can use the same classes. We can refer to these as software components. (Brad Cox [Cox 86] calls these software-ICs, in analogy to the integrated circuits used in hardware design). Another form of sharing occurs when two or more different 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, it means two or more different 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.

7.5.3 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.

7.5.4 Software Components

In the first chapter, we noted that inheritance will provide programmers with the ability to construct reusable software components. The goal is to permit the development of new and novel applications which nevertheless require little or no actual coding. Already, several such libraries have become commercially available, and we can expect many more specialized systems to appear in time.

7.5.5 Rapid Prototyping

When a software system can be constructed largely out of reusable components, development time can be concentrated on understanding the portion of the system that is new and unusual. 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 based on experience with the first, further experimentation takes place, and so on for several iterations. Such a style of programming is particularly useful in situations where the goals and requirements of the system are only vaguely understood when the project begins.

7.5.6 Polymorphism and Frameworks

Software produced in a conventional manner is generally written in a bottom-up fashion, although it may be *designed* in a top-down fashion. 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 underlying 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 changing their low level parts. We will have much more to say about this topic in subsequent chapters.

7.5.7 Information Hiding

When a programmer reuses a software component, he or she needs only to understand the nature of the component and its interface. It is not necessary for the programmer to have detailed information concerning such matters 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.

7.6 The Costs of Inheritance

Although the benefits of inheritance in object-oriented programming are great, it is true that there is almost nothing that is without cost of one sort or another. For this reason, it is only honest for us to consider the cost involved in the use of object-oriented programming techniques, and in particular the cost associated with the use of inheritance.

7.6.1 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 be prepared to deal with arbitrary subclasses, are often slower than specialized code.

Concern about efficiency, however, is often misplaced.[3] For one, the difference is often small. For another, the reduction in the speed of execution may be balanced by an increase in the speed of software development. Finally, most programmers actually have little idea how execution time is being used in their programs. It is far better to develop a working system, and then to 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.

7.6.2 Program Size

The use of any software library frequently imposes a size penalty over the use of systems especially constructed for a specific project. Although this expense may in some cases be substantial, it is also true that, as memory costs decrease, the size of programs is becoming less important. Containing development costs and producing high-quality and error-free code in a rapid fashion are now more important than is limiting the size of programs.

7.6.3 Message-Passing Overhead

Much has been made out of the fact that message passing is, by its nature, a more costly operation than is simple procedure invocation. As with the question of overall execution speed, however, over concern about the cost of message passing is frequently penny-wise and pound-foolish. For one, 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 weighted against the many benefits of the object-oriented technique.

[3] The following quote from an article by Bill Wulf provides some apt remarks on the importance of effiiciency: "More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity" [Wulf 72].

A few languages, notably C++, make available to the programmer a number of options that can reduce the message-passing overhead. These include eliminating 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 chose to use `dynamic` methods, which use a run-time lookup mechanism, or `virtual` methods, which use a slightly faster technique. Dynamic methods are inheritantly slower, but require less space.

7.6.4 Program Complexity

Although object-oriented programming is often touted as a solution to the problem of software complexity, overuse of inheritance can often, in fact, 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, and we will discuss this in more detail in a later chapter.

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? Can 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 on the behavior? Characterize the type of inheritance you are using. Explain the reasoning behind your design.

4. Why is the example used in the following explanation not a valid illustration of inheritance?

   > 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.