Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages

Josephine Micallef

Columbia University

Department of Computer Science

New York, NY 10027

August, 1987

CUCS-285-87

Abstract

The object-oriented paradigm, first introduced in the language Simula, has been the central design principle of many new programming languages, and has also resulted in object-oriented extensions to existing conventional languages. This paper is a comparative survey of object-oriented programming languages (OOPLs). The framework for the comparison is the suitability of OOPLs for the development of large software systems. We therefore limit our discussion to their support for (1) *encapsulation* — the strict enforcement of information-hiding; (2) *reusability* — the ability to reuse a system, or parts thereof, in the construction of a new system; and (3) *extensibility* — the ease with which a software system may be changed.

Table of Contents

1. Introduction 1
2. Basic Concepts and Terminology 5
    2.1. Objects 5
    2.2. Classes 6
    2.3. Inheritance and Subclasses 8
    2.4. Synonyms for Terminology in Other Languages 9
3. Encapsulation 10
    3.1. Interface to Instantiating Clients 12
        3.1.1. Object Creation 13
        3.1.2. Access to Object's Representation 19
        3.1.3. Distinction between Public and Private Operations 24
        3.1.4. Discussion 25
    3.2. Interface to Inheriting Clients 25
        3.2.1. Visibility of Superclass's Representation and Operations 26
        3.2.2. Conflict Resolution of Inherited Instance Variables 30
        3.2.3. Discussion 32
    3.3. Visibility of Inheritance 33
        3.3.1. Excluding operations 33
        3.3.2. Subtyping 35
        3.3.3. Reference to ancestors 38
        3.3.4. Multiple inheritance 39
        3.3.5. Discussion 41
4. Reusability and Extensibility 42
    4.1. Inheritance 43
        4.1.1. Single Inheritance 44
        4.1.2. Multiple Inheritance 48
        4.1.3. Dynamic Combination of Behavior 51
    4.2. Polymorphism 53
        4.2.1. Simple Polymorphism 53
        4.2.2. Multiple Polymorphism 55
    4.3. Comparison to Reusability and Extensibility in ADA 59
5. Conclusion 62

1. Introduction

This paper is a comparative survey of object-oriented programming languages (OOPLs). The notion of *objects* was first introduced in the language Simula, designed in the late 60's. However, object-oriented programming did not emerge as a new programming paradigm until Smalltalk came along in the late 70's. The radical difference between Smalltalk and previous languages, including Simula, is that in Smalltalk everything is an object, from the primitive language types like integers and characters, to application-dependent types like windows in a window management application. Many object-oriented languages are in wide use today. While a few of these languages have been designed from scratch (for example, Traits [Curry 84, Curry 82], Trellis/Owl [Schaffert 86, Schaffert 85, O'Brien 85], Eiffel [Meyer 85]), most of them are hybrid languages, that is, conventional languages with object-oriented concepts built on top. Examples include Objective-C [Cox 84] and C++ [Stroustrup 86a] (extensions to C), Flavors [Moon 86, Keene 85] and Loops [Stefik 86, Bobrow 86] (extensions to Lisp) and Clascal [Schmucker 86, Doyle 86] (modified front end to Pascal).

The framework for the comparison is the use of OOPLs for the development of large software systems. Although it is hard to pin down exactly what constitutes a "large" system, it is usually made up of a large number of units, thereby involving many people in its development (usually more than ten). Typically, the problem solved by the system is itself very complex. Within this framework, we identify three objectives that a programming paradigm should provide support for: (1) encapsulation, (2) reusability and (3) extensibility. We evaluate several object-oriented languages to see how well they meet these objectives, and show that the object-oriented programming paradigm, in general, has better support for these objectives than conventional programming techniques (like structured programming). In this paper, the emphasis is on the constructs that are provided for programming-in-the-large [DeRemer 76], such as interfaces of units and the means for composing the units into a system, rather than on the constructs available for programming-in-the small, such as the kinds of statements, declarations, and so on.

In an object-oriented programming language, an "object" is a self-contained entity which has:

- its own private data, and
- a set of operations to manipulate that data.

The set of operations defined for an object constitutes a *uniform external interface* to the rest of the system. Interaction with an object occurs through requests for the object to execute one of the operations in its interface. A computation is defined as a sequence of interactions among objects in the system. New objects can be defined as extensions of existing ones by a technique called *inheritance.* The specification "B inherits A" in the definition of object B means that B contains the data and operations defined for object A in addition to those defined for B. Inheritance may be viewed as an abbreviation mechanism that avoids redefining the attributes of an object in the definition of another. In this paper, languages which have abstract data types with uniform external interfaces (for example, the ADA™ package [Habermann 83] and CLU cluster [Liskov 81]) but do not have inheritance are not considered object-oriented.

One advantage of using object-oriented programming languages for building large systems is that they facilitate the creation of software components that closely parallel the application domain, an important feature for building inexpensive, understandable systems. Procedural languages, such as Pascal and Ada, often lead to program structures radically different from the structure of the problem domain. The reason is that in such languages there are two kinds of entities: data, which is passive and represents the information of the system, and functions, which can manipulate the data. The designer of a system written in a procedural language can either map the problem domain into a set of functions, and then define the data structures needed by the functions, or he can map the problem domain to the data, and then define functions that transform the input data to the output data. By contrast, object-oriented design treats functions and data as indivisible aspects of objects in the problem domain. Many applications can be designed by straightforwardly identifying the objects in the problem domain, and deciding how to implement the objects' behavior in the computer.

For example, in a window management system there would be objects corresponding to the windows on the display. The local data for each window object would contain information about the size of the window, its location on the display, its contents, and so on. Additionally, the window object would have operations for its manipulation, such as operations for moving it, changing its size or contents, deleting it, etc. A user interacts with the system by specifying a window object and the operation that he wants performed on that window through some interface, such as a menu.

Three additional advantages of object-oriented programming languages are their support for encapsulation, reusability and extensibility. *Encapsulation* is the strict enforcement of information-hiding. *Reusability* is the ability of a system to be reused, in whole or in parts, for the construction of new systems. *Extensibility* is the ease with which a software system may be changed to account for modifications of its requirements. These three are particularly relevant to the development of large software systems.

The characteristics of large software systems impose several requirements both on the system development process and on the tools used, namely:

- The work must be divided among different people. One person should not have to know the details of units built or modified by other people, but just how the unit he is responsible for interacts with the rest of the system. A programming language that enforces encapsulation alleviates some of the problems of multi-person development teams.
- The cost of developing a large system is very high. The development cost can be reduced if some of the units that the system is built from could be taken from already existing systems. Similarly, it would be beneficial if some parts could be reused in future projects.
- Due to the high development cost, large software systems are usually long-lived. During its lifetime, the system undergoes considerable modification. This might be as a result of changes to the specifications over a period of time, perhaps due to discovery of weaknesses in the system as it is used. Therefore, the system should be easily modifiable. Encapsulation minimizes the number of units that have to be changed as a result of changing one unit to fix a bug. On the other hand, extensibility makes it feasible to enhance a system without changing any units at all, but instead adding new capabilities via a new unit that extends one or more existing units.

The rest of the paper is organized as follows: In section 2 we introduce the basic concepts of object-oriented programming and define terminology used. Encapsulation is the topic of section 3. Reusability and extensibility as provided by object-oriented programming languages are discussed in section 4. Inheritance, discussed in subsection 4.1, is an important tool for constructing new systems from old parts, but it can also be used to build extensible systems. These are systems that can grow "gracefully" over a period of time. Polymorphism is another mechanism for building extensible systems, and is discussed in subsection 4.2. We conclude by summarizing the contributions of OOPLs to encapsulation, reusability and extensibility, and therefore to the development of large software systems, and list some open problems.

2. Basic Concepts and Terminology

This section introduces the object-oriented paradigm and defines the terms used in this paper. The discussion will use Smalltalk as the prototypical object-oriented language; the terminology is therefore somewhat specific to this language. At the end of the section, we provide the equivalent terms used in other languages.

2.1. Objects

Object-oriented languages combine the descriptions of data and procedures within a single entity called an *object*. An object is a well-defined data structure coupled with a set of operations that describe specifically how that data can be manipulated. The behavior of an object is characterized by the operations defined on it; this means that only these operations can manipulate the object. There are two distinct views of an object: (1) The *external* view of the object, visible to a programmer who wants to use the object, specifies the behavior of the object but does not say how this behavior is realized. (2) The *internal* view of the object, visible to the programmer who is implementing the object, describes how the object's behavior is actually achieved.

A programmer of an object-oriented system *sends a message* to an object, called the *receiver*, to invoke one of the object's operations. The message includes a symbolic name, the *selector*, which describes the desired operation. It may also contain arguments to be passed to the operation. The message, then, describes what the programmer wants to happen, not how it should happen. The message receiver, in turn, has *methods* which describe how the operations are performed. A method is like a procedure in that it is comprised of a sequence of executable statements. However, methods are inseparable from the objects they are defined for; a method can only be invoked when the object receives a message whose selector corresponds to that method.

As an example, suppose we wanted to build a window management system. Windows are opaque, rectangular areas on a display device. Windows can be moved, overlapped with other windows, and deleted. The system would have a set of objects representing windows — one object for each window on the screen. A window object has *private variables* that contain the window's size and location,which might be the *x* and *y* coordinates of the top left corner, and the width and height of the object. The object also contains methods for moving the window, displaying it on the screen, deleting the window, returning the window's size and location, and so on. Each method corresponds to a selector of a message; for example, the selectors corresponding to the above methods would be `move`, `display`, `delete`, `width`, `height`. (In this paper, we use an alternate typeface to refer to elements of example systems.) If `my-window` denotes a window object (created in a fashion described below), then to display the window on the screen, we would send it a message with selector `display`:

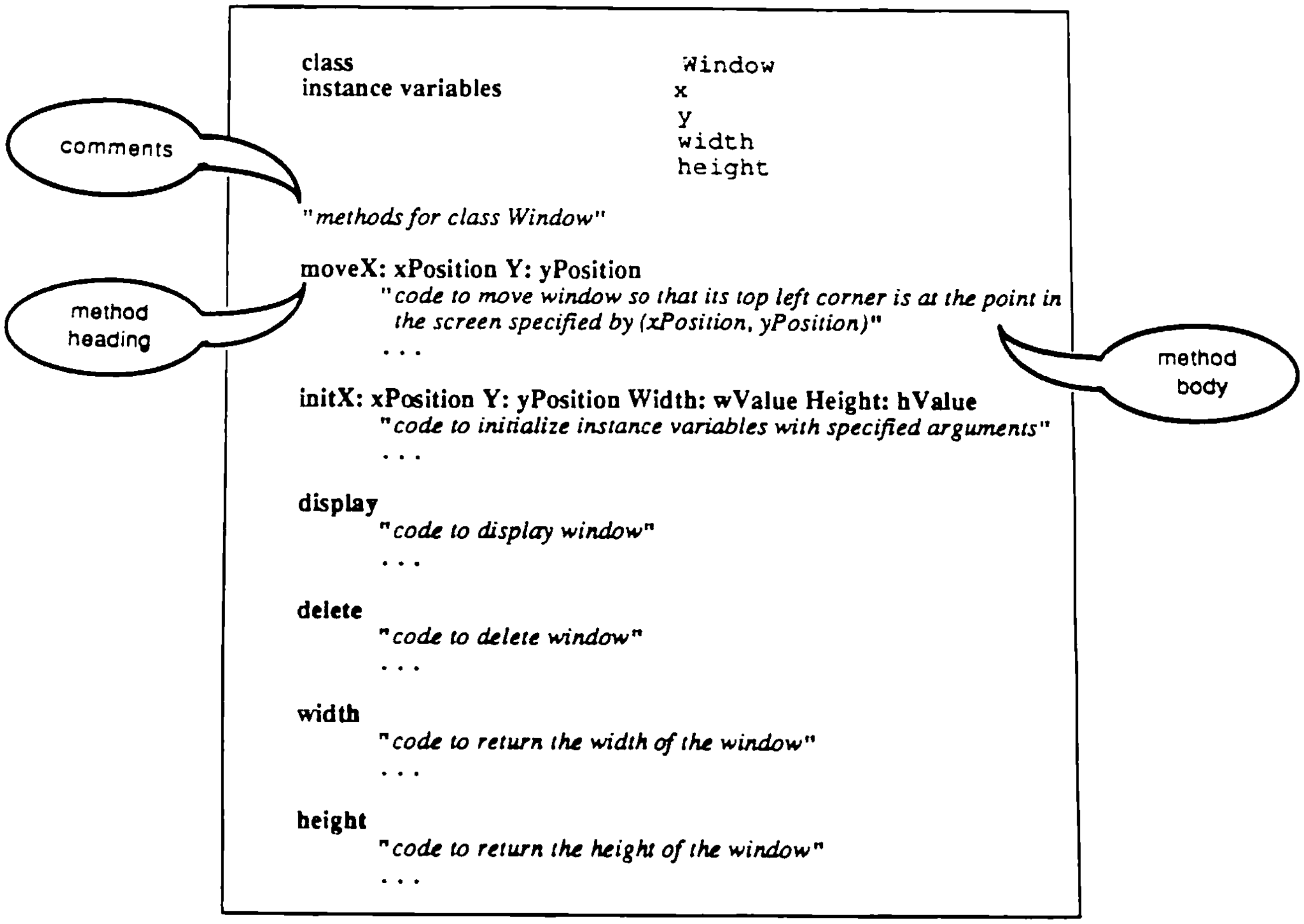
```
    my-window display
```

This is called a *message expression*; it consists of an object which is the receiver of the message (`my-window`), the selector of the message (`display`), and arguments if specified by the selector (in this case no arguments are needed). `My-window` searches its list of methods, executing the one whose name is `display`.

2.2. Classes

A system will often contain many similar objects. For example, the window management system may have several windows which, with the exception of their location and size, exhibit identical behavior. A description of the common features of these objects is provided in a *class* description; individual objects are known as *instances* of the class. This description includes the form of the instances' private memory, called *instance variables*, and the methods that manipulate the instances. Only the methods of the class can access the instance variables directly; other methods must use message sending to gain access to or update the value of the instance variables. All instances of a class respond to the same messages; they can only differ in the value of their instance variables.

In the window management example, the programmer would define a class `Window` containing instance variables `x`, `y`, `width` and `height`, and methods corresponding to the selectors `move`, `init`, `display`, `delete`, `width`, `height` (see figure 2-1). The selector for the `move` method is a *compound selector*; it consists of two symbols, each with a trailing colon. A symbol with a trailing colon is called a *keyword*. When a message with such a selector is sent, arguments are inserted after each colon. To refer to a compound selector, the keywords are concatenated, as in `moveX:Y:`. The `xPosition` and `yPosition` in the method heading are dummy arguments; these are replaced by the actual arguments when the method is invoked. When a message is sent, the receiver's class determines the appropriate method to execute.



```
class                          Window
instance variables            x
                              y
                              width
                              height

"methods for class Window"

moveX: xPosition Y: yPosition
        "code to move window so that its top left corner is at the point in
         the screen specified by (xPosition, yPosition)"
        . . .

initX: xPosition Y: yPosition Width: wValue Height: hValue
        "code to initialize instance variables with specified arguments"
        . . .

display
        "code to display window"
        . . .

delete
        "code to delete window"
        . . .

width
        "code to return the width of the window"
        . . .

height
        "code to return the height of the window"
        . . .
```

Figure 2-1: The Window Class

Classes are objects themselves, and as such they are themselves instances of a class, called the *metaclass*. The metaclass is useful for defining the behavior of the class as a whole; its most common use is to provide methods to create and initialize instances of the class. For example, the creation of a window at a specified location and of a given size can be performed by the method `newX:Y:Width:Height` in the Window metaclass, shown in figure 2-2. The body of this method consists of two steps. First, an uninitialized window is created by sending the message with selector `new` to `self`. *Self* is a pseudo-variable available in every method, which refers to the receiver of the message that invoked that method. Its value can be accessed but not changed. The *new* method is available in all metaclasses, and it creates uninitialized instances.

Secondly, a message with selector `initX:Y:Width:Height` is sent to the window created in the first step. The method for this selector, defined in the class `Window`, initializes the instance variables with the specified arguments. The ^ preceding the expression in the method in figure 2-2 indicates that the return value of the method is the value of the expression; in this case the return value is the new, initialized window.

```
metaclass              Window

"instance creation method"

newX: xValue Y: yValue Width: wValue Height: hValue
     ^ ((self new) initX: xValue Y: yValue Width: wValue Height: hValue)
```

Figure 2-2: The Window Metaclass

To create a window and give it a name, we send a message with selector `newX:Y:Width:Height` to the class `Window`, and assign the result (that is, the newly created window with the specified size and position) to a variable, as in the following example:

```
    my-window <- Window newX:50 Y:100 Height:200 Width:300
```

2.3. Inheritance and Subclasses

Inheritance is a technique that allows new classes to be built on top of older, less specialized classes rather than written from scratch. The new class is the *subclass*; the old one is the *superclass*. The subclass *inherits* the instance variables and methods of the superclass. The subclass can add new instance variables and methods of its own. It can also define a method with the same selector as one of the superclass's methods; this is know as *overriding* a method.

As an example, the window management system might contain windows that have a maximum height. These would be instances of a subclass of the ordinary class of windows, `Window`, that adds an instance variable to represent the maximum height and provides a new method for the message that changes a window's size which only increases the height of the window up to the maximum height.

Every class inherits from the class *Object*. This is a class provided by the language which describes rudimentary behavior common to all objects in the system. The Object class has methods for testing for class membership, object comparison, printing of objects, and copying objects. These messages can be extended and modified in the object's class to provide more specific behavior. The subclass/superclass relation structures the classes of the system into an *inheritance tree* rooted at Object. Some languages allow classes to inherit from more than one superclass; this is called *multiple inheritance.* The inheritance relation in a multiple-inheritance system is in the form of a directed acyclic graph.

2.4. Synonyms for Terminology in Other Languages

Message sending: Instead of message sending, many OOPLs use a variant of function call to invoke an operation on an object. These calls are different from conventional function calls in that the first argument is distinguished from the rest. The first argument corresponds to the receiver, and the actual function that is invoked depends on the kind of object denoted by the first argument. The other arguments in the call are ordinary arguments. The function name is equivalent to the message selector. For example, the function call `display(my-window)` would invoke the `display` operation defined for the object `my-window`.

Method: Also known as operation, procedure or function.

Class: Some languages use the word *type* instead of class. Subtype and supertype correspond to subclass and superclass respectively.

Metaclass: Some languages do not view classes as objects, and so they need an alternate mechanism for creating initialized instances. Some Lisp-based OOPLs, for example, provide a special function, **make-instance**, which can take a variable number of arguments to create initialized instances of any class.

Self: Also known as *this* or *me.*

3. Encapsulation

Encapsulation is a technique for minimizing interdependencies among separately-written modules by defining strict external interfaces [Snyder 86a]. An encapsulated module can only be accessed by clients (that is, other modules that make use of this module) via this interface. Implementation details are "hidden" within the module. The primary reason for requiring encapsulation is to make it possible to change (improve) the implementation of a module without having to change (and/or recompile) the module's clients. This effective decoupling of modules is indispensable for the development of large programs, in particular, where modules are developed by different people. Without decoupling, any change or correction to a module that is used by many other modules in the system would become virtually impossible. A secondary, but not less important motivation for encapsulation is that if the internals of a module are protected from outside access, we can guarantee their correct functioning, and thereby can limit the area to search for an error in the case of a malfunctioning program.

In object-oriented programming languages, the unit of modularity is the object. However objects are more than just modules; they are also implementations of abstract data types. An abstract data type is a description of the behavior of an object without any reference to the actual representation of the object or how the operations defining its behavior are implemented [Shaw 84]. This is exactly the definition of an object in OOPLs: To its users, it is an abstract data object — it can only be manipulated via its external interface. Internally, it chooses a particular data structure to represent its data, and has concrete implementations for the operations specified in its interface. The encapsulation mechanism provided by abstract data types improves the localization of modifications. A change to the data structure is likely to require a change to the concrete implementations of the object's operations, but the effect of these changes is confined within the boundaries of the object. Similarly, a change to the program using the object has no effect on the correctness of the code within the object. An OOPL supports encapsulation if it allows users of objects to access them only via their external interfaces.

A class has two categories of clients: *instantiating* clients, which create instances of the class and perform operations on them, and *inheriting* clients, which inherit from the class. The second category of clients are the descendents of the class in the inheritance graph. There is a spectrum of possible interfaces that can be presented by an object to its descendents, as illustrated in figure 3-1. At one extreme end of the spectrum, an object can present the same external interface to both kinds of clients. This might be too restrictive for descendents: when inheritance is used for code sharing among objects,[1] an object might need to refer to the data or some private operation of an ancestor to be able to take full advantage of the shared code. At the other extreme, a descendent object can be given full access to the data and operations defined in an ancestor object. This would remove the benefits of encapsulation to the descendent objects — a change to an object is visible to all its descendents. Some languages provide separate interfaces to inheriting and instantiating clients; descendents still can only access the object through their interface, but they have access to operations not available to instantiating clients. Such languages provide an interface to inheriting clients that lies somewhere in the middle of the spectrum.

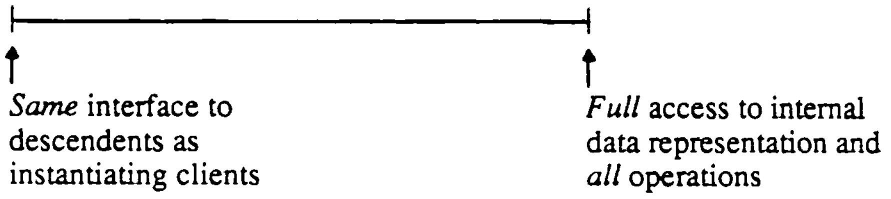

Same interface to descendents as instantiating clients
Full access to internal data representation and *all* operations

Figure 3-1: Spectrum of Possible Interfaces to Inheriting Clients

When inheritance is used as a way of reusing code, the position of an object in the inheritance hierarchy should be invisible to users of the object. The structure of this hierarchy is an implementation decision that should be allowed to change without affecting users of the objects in the hierarchy (as long as this does not change the external behavior of the object), in much the same way that compatible changes to the representation of an object should be invisible. For example, for efficiency reasons we might want to change the implementation of a class S that inherited from class C to a new implementation without inheritance. If the use of inheritance in

[1]An alternative interpretation of inheritance is found in knowledge representation systems [Brachman 85]. In these systems, the inheritance hierarchy is assumed to be a public declaration that a descendent object is a *specialization* of the ancestor. For the purposes of this section, we will only consider inheritance as an implementation mechanism.

S was visible to its clients, this modification would necessitate changes to the clients of S. Being able to safely make changes to the inheritance graph is important for the support of large, long-lived systems: It is often the case that new classes are implemented as subclasses of existing classes for quick prototyping of the system, even if this means a less efficient implementation than starting from scratch. We would like to be able to define the class again more efficiently in a later version and still be able to reuse (most of) the code from the prototype.

In the rest of this section we will compare how several OOPLs support the three levels of encapsulation outlined above, namely (1) the interface provided by an object to instantiating clients, (2) the interface provided to inheriting clients, and (3) the visibility of inheritance. The languages under consideration are Smalltalk [Goldberg 83], Flavors [Moon 86, Keene 85, Weinreb 80], CommonObjects [Snyder 86b], Simula [Dahl 70], Trellis/Owl [Schaffert 86, Schaffert 85, O'Brien 85], and C++ [Stroustrup 86a]. The first three languages are dynamically typed whereas the last three are statically typed. Since this subset of OOPLs is not the only one we could have chosen, we have to justify our choice of languages for comparison. All OOPLs known to the author were divided into disjoint sets where all the languages in any one set provided identical support for encapsulation. One representative language was then taken from each set, with preference given to more popular languages than to lesser known ones.

3.1. Interface to Instantiating Clients

The purpose of this section is to compare how the various languages enforce that access to an abstract data object is done only through the object's external interface, and how they allow minimal external interfaces to be defined. We will do this by means of an example: implementation of the abstract type Complex. The external interface of this abstract data type contains the following operations:

- ***create*** — allocates a new complex number object
- ***real:imag:*** — assigns the arguments to the real and imaginary components of a complex number
- ***real*** — returns the real part of the complex number
- ***imag*** — returns the imaginary part of the complex number
- ***printString*** — prints a string representing the complex number in the form *(x + iy)* where x and y are the real and imaginary parts respectively.

- + — returns the sum of two complex numbers
- * — returns the product of two complex numbers

We will consider two implementations of Complex:

1. a complex number represented by its Cartesian coordinates, and
2. a complex number represented by its Polar coordinates.

The second implementation might be more efficient if the ratio of * to + operations is high; the first implementation would be preferred if the opposite was true. In both implementations, the external interface is the same as above. That is, programs that use complex numbers view them as points in a cartesian coordinate system. Figures 3-2 and 3-3 show how the two implementations would be written in Smalltalk.

In order to determine how well a language supports encapsulation, we will study what the effects of changing from either implementation to the other has on programs using complex numbers. In our example, the interfaces of the two implementations are identical, and therefore in a maximally encapsulated language, changing from either implementation to the other one would have no effect on any other code. If the interfaces of the two implementations were not identical, but one was a superset of the other (that is, a basic and an enhanced implementation), then going from the basic implementation to the more enhanced version should not affect any existing code but the reverse does not necessarily have to hold. There are three areas where OOPLs can potentially violate encapsulation by allowing representation and/or implementation details of an object to be visible to instantiating clients: (1) creation and initialization of the object, (2) accessing and updating the object's internal state, and (3) distinction between the object's public and private operations. These are discussed in turn in the following three sections.

3.1.1. Object Creation

All OOPLs provide a way of creating new instances of a class. For example, in Smalltalk, the method **new** defined in the class Object can be used to create new instances of any class. A complex number instance can be created and denoted by the variable `c` by this statement:

```
    c <- Complex new
```

To initialize the internal state of the created object, another message must be sent to the object specifying values for the instance variables, for example:

```
classname                            Complex
superclass                           Number
instance variables                   real
                                     imag

"instance methods"

"initialization"
        real: realVal imag: imagVal
            real <- realVal.
            imag <- imagVal

"inquiries"
        real
            ^ real

        imag
            ^ imag

"printing"
        printString
            ^ (real asString),
              ((imag positive)
                  ifTrue: [' + i']
                  if False: [' - i']) asString,
              (imag abs) asString

"arithmetic"
        + aComplex
            ^ Complex new
                real: real + aComplex real
                imag: imag + aComplex imag

        * aComplex
            ^ Complex new
                real: (real * aComplex real) - (imag * aComplex imag)
                imag: (real * aComplex imag) + (imag * aComplex real)
```

Figure 3-2: Implementation 1 in Smalltalk

```
c real:3 imag:2
```

Most languages provide a way for performing these two operations (that is, creating a new instance and initializing it) in one step. In Smalltalk, a method which combines the create and initialize operations can be written as follows:

```
"initialized instance creation method"
newReal: realVal imag: imagVal
        ^((Complex new) real: realVal imag: imagVal)
```

thus allowing the previous two statements to be abbreviated to:

```
c <- newReal:3 imag:2
```

In Smalltalk, this does not create a problem with encapsulation since the initialization method is

```
classname                    Complex
superclass                   Number
instance variables           r
                             theta

"instance methods"

"initialization"
        real: realVal imag: imagVal
            self
                 r: (rFromReal: realVal imag: imagVal)
                 theta: (thetaFromReal: realVal imag: imagVal)

"inquiries"
        real
            ^ r * (theta cos)

        imag
            ^ r * (theta sin)

"printing"
        printString
            ^ ((self real) asString),
              (((self imag) positive)
                   ifTrue: [' + i']
                   ifFalse: [' - i']) asString,
              ((self imag) abs) asString

"arithmetic"
        + aComplex
            | realsum imagsum |
            realsum <- self real + aComplex real.
            imagsum <- self imag + aComplex imag.
            ^ Complex new real: realsum imag: imagsum

        * aComplex
            ^ Complex new
                 r: r * aComplex r
                 theta: theta + aComplex theta

"private"
        rFromReal: realVal imag: imagVal
            ^ ((realVal ^ 2) + (imagVal ^ 2) sqrt

        thetaFromReal: realVal imag: imagVal
            ^ (imagVal / realVal) arctan

        r: rVal theta: thetaVal
            r <- rVal.
            theta <- thetaVal
```

Figure 3-3: Implementation 2 in Smalltalk

not coupled with the class's instance variables. Indeed, we can change the representation to polar coordinates, with the class containing instance variables `r` and `theta` (as in figure 3-3) and still have an instance creation method identical to the one above. However, in this case, the method `real:imag:` has different semantics than that of figure 3-2. It first converts the arguments to polar coordinates and then assigns the result to the instance variables. Similar ways of creating initialized instances are found in the languages CommonObjects, Trellis/Owl and C++. However, in some other OOPLs, for instance, Simula and Flavors, the shortcut provided by the language for creating initialized instances exposes implementation details to programs using these instances.

Simula, Flavors

In Simula, a class has formal parameters. Initialized objects can be created by specifying actual values for each formal parameter. A complex number class using the cartesian representation looks as follows:

```
class complex (real,imag); real real,imag;
begin
     /* local variable declarations */
          ...

     /* definition of local procedures for manipulating object */
          ...

     /* class body — automatically executed by new statement to initialize object */
          ...
end complex;
```

A complex number object is declared and created by the following statements:

```
ref (complex) c;                /* an object can only be referenced via a pointer */
c :- new complex(3.0,2.0);      /* creation and initialization of object */
```

Since the class's formal parameters are used in initializing the object, the number, type and semantics of formal parameters are a part of the object's external interface. If in the first implementation of complex numbers, the class declaration only had the formal parameters `real` and `imag`, and no local variables, then to change the representation to polar coordinates, we could not simply make `r` and `theta` the new class parameters because all existing new statements like the one above would be meaningless.

In Flavors, the instance variables of a class (called a flavor in this language) can be initialized when an instance of the class is created if the option **:initable-instance-variables** is given in the class definition. If the complex number class is defined thus:

```
(defflavor complex
        (real imag)
        (number)                    ; inherits from the flavor number
    :initable-instance-variables
    ; other options
    ... )
```

then we can create and initialize a `complex` number object as follows:[2]

```
(setq c
    (make-instance 'complex
        :real 3
        :imag 2))
```

`:real` and `:imag` are *initialization keywords*, formed by a leading colon and the instance variable name.[3] Initialization keywords are used with **make-instance**, a Flavors built-in function that creates a new instance of a class, to initialize the corresponding instance variables. Since the instance variable name is part of the initialization keyword, redeclaring an initable flavor to have a different set of instance variables will invalidate all existing **make-instance** statements. If we redefine the `complex` flavor with instance variables `r` and `theta`, we cannot simply define methods `:real` and `:imag` to convert from cartesian into polar coordinates (as was done in Smalltalk) because method names do not use keywords.

A way to remove this problem with encapsulation is to go back to the two-step method of creating initialized instances. This can be done in Simula by not declaring any formal parameters in the class heading, but instead declare `real` and `imag` as local variables. In Flavors, this is accomplished by not providing the initable-instance-variables option in the class declaration. A separate procedure is used to initialize the local variables (or instance variables in the case of Flavors) after the object is created using the **new** (or **make-instance**) statement. The initialization procedure for the first implementation would simply assign the arguments to local variables (or instance variables). The second implementation would have `r` and `theta` as local variables (or instance variables), and the initialization procedure would first convert the arguments to polar coordinates, and then assign the result to the local variables (instance variables).

[2]Flavors is implemented on top of Common Lisp [Steele 84], and follows its syntax closely.

[3]*Keywords* are a special category of symbol in Common Lisp written with a leading colon. They are self-evaluating, and therefore keyword operation names need not be quoted in an operation invocation.

CommonObjects, Trellis/Owl, C++

In CommonObjects, each instance variable can have the option **:initable**, with the same semantics as in Flavors. However in this language (also implemented on top of Common Lisp) method names also use keywords. Initialized instances can be created using **make-instance** as we described for Flavors. Changing from the first implementation to the second one will not affect client code if the classes are defined in the following way. In the first class, the instance variables `real` and `imag` are declared initable, thus automatically creating the functions `:real` and `imag:` which are used as initialization parameters. In the second implementation, the instance variables would be `r` and `theta`, and these would not be declared initable. Instead, the implementor of the class would define his own methods `:real` and `:imag` corresponding to the initialization methods for the first implementation, which take cartesian coordinates, translate them to polar, and then assign the results to the instance variables.

The Trellis/Owl method for creating initialized instances is identical to what we described for Smalltalk, except for syntax, and so it will be skipped.

A C++ class has *constructors* for creating initialized objects. A constructor is a function declared within the class which has the same name as the class. For example:

```
class complex {
      double real, imag;
public:
      complex(double,double);    //constructor specification
      ...
}

//definition of complex class constructor
complex::complex(double x, double y)
{
      real = x;
      imag = y;
}
```

The function `complex` is the constructor for the class. It takes two parameters, the real and imaginary parts of the complex number and assigns them to the corresponding fields of the class. The class for the second implementation would have fields `r` and `theta` and a constructor defined as follows:

```
class complex {
      double r, theta;
public:
      complex(double,double);
      ...
}

complex::complex(double x, double y)
{
      r = sqrt ( pow(x,2) + pow(y,2) )
      theta = atan(y/x)
}
```

In C++, Flavors, and CommonObjects, one can specify default values for some or all of the instance variables. These are used when a new instance is created if the values are not initialized any other way. Default arguments are very useful in situations which require instance variables to be added or removed from a class declaration without affecting old code. By specifying a default value for the added/removed instance variable in the constructor, one can ensure that all the initialization code written when the class had a smaller number of instance variables remains correct.

3.1.2. Access to Object's Representation

In a strongly encapsulated language, an object's representation, that is, the data structure that defines the object's internal state, should not be manipulated directly. The designer of the class can choose to provide *accessor* and *update* operations for respectively accessing or changing the values of the instance variables. Some languages automatically provide these functions for (all or some of) the instance variables of a class. If the underlying representation of an object changes, it may be possible to provide the exact same operations but implemented in a different way, thus hiding the change from the clients. This section examines our representative set of OOPLs to see whether direct access to the object's representation by the object's instantiating clients is permitted.

Simula

One can access all attributes of a class, that is the class parameters, local variables, and procedures enclosed within the class, through the use of the dot notation. For example, if c is the complex number object created in the previous section, the conjugate of c[4] can be created in the

[4]Two complex numbers are *conjugate* if they differ only in the sign of the imaginary parts.

following manner:

```
real x,y;
x := c.real;
y := -c.imag;
ref (complex) conjugate;
conjugate :- new complex (x, y);
```

Simula does not support strictly encapsulated data types: a program can directly manipulate a complex number object through its representation, without using the operations that are defined to manipulate the abstract data type.

Smalltalk

An object can have two types of instance variables — *named instance variables* and *indexed instance variables*. In our example, the class Complex has only named instance variables. For named instance variables, the designer of the object decides who has access to the instance variables — he might (or might not) provide messages to initialize and return their values.[5] On the other hand, indexed instance variables are accessed through the messages `at:` and `at:put:` defined in the class Object, and therefore available to all instances, which respectively retrieve and store the element indicated by the first argument. Changing from an implementation with indexed instance variables to one with named instance variables requires the existence of a mapping from indexed to named instance variables and the redefinition of the `at:` and `at:put:` methods to access and update from the corresponding named instance variable. The reverse, that is going from a named instance variable implementation to one with indexed instance variables, is accomplished similarly.

Trellis/Owl, Flavors, CommonObjects

Trellis/Owl types can have *components*. A component can be a *field* or a *computed value*. For a field component, there is an actual "slot" in the physical representation of the object that will store the value for this component. For example, the first implementation of complex numbers would have the following components:

[5]In order to implement system components (language interpreter, debugger, programming environment, etc.), most OOPLs provide an escape from this strict encapsulation. In Smalltalk, the messages `instVarAt:` and `instVarAt:put:` are provided for directly accessing named instance variables.

```
component me.real: Real     /* me denotes the controlling object — like self in Smalltalk */
      is field;

component me.imag: Real
      is field;
```

Accessor and update operations for a field component are automatically generated to fetch from and store to the correct slot in the object. The operations generated for the real component would have the following specification:[6]

```
operation get_real (me)
      returns(Real);

operation put_real (me, value: Real)
      returns(Real);
```

The equivalent of field components in Flavors and CommonObjects are the class's instance variables. Accessor functions for the instance variables can also be generated automatically in these two languages. In Flavors, the option **:readable-instance-variables** specified in the class declaration generates accessor functions for all the variables in the class. The **:gettable** option in CommonObjects achieves the same result, but is specified (or not) for each instance variable, thus providing the implementor of the class with a finer level of granularity. The accessor functions generated for the Complex implementation of figure 3-2, containing two instance variables, `real` and `imag`, would be `complex-real` and `complex-imag` in Flavors and `:<real>` and `:<imag>` in CommonObjects.

If the option **:writable-instance-variables** is specified in a Flavors class declaration, a client of instances of that class can alter the values of the instance variables using **setf** and the accessor functions, as in:

```
(setf (complex-real c) '5)
```

Setf is a Common Lisp macro that examines the specified access form and produces a call to the corresponding update function. In the case that the accessor function simply retrieves the value of a memory location (as in the case of the example above), a corresponding update function that stores the new value in the same memory location is provided automatically.

[6]The use of the **me** keyword as one of the parameters of an operation indicates that this operation is an *instance* operation, that is one that applies to individual instances of a type, as opposed to a *class* operation, which applies to the type as a whole. An example of a class operation is the creation operation.

In CommonObjects, the accessor function of an instance variable cannot be used to change its value; instance variables can only be assigned to within a user-defined method, or at object creation time, as described in the previous section.

Trellis/Owl components can also be implemented as *computed values*. That is, the desired accessor and update functions can be described in terms of other components or operations. For example, the complex number class using the polar coordinate representation could be implemented as follows:

```
component me.rho: Real
      get private      /* not available to clients of instances */
      put private      /*          of the class. */
      is field;

component me.theta: Real
      get private
      put private
      is field;
```

We can then define the components `real` and `imag` in terms of the fields `rho` and `theta`:

```
component me.real: Real
      get is
            begin
                  return get_rho(me) * (cos(get_theta(me)));
            end;
      put is
            begin
                  me.rho := sqrt((value**2) + (get_imag(me)**2));
                  me.theta := arctan(get_imag(me) / value);
            end;

component me.imag: Real
            ...
```

The specifications for the real and imaginary components in the second implementation are exactly the same as in the first implementation, and therefore, clients of the type cannot distinguish between components that are fields and those that are computed values. Computed values provide a convenient mechanism for changing an object's representation without changing the external view and yet allowing accessor functions to be automatically generated when possible.

Computed values in Flavors and CommonObjects can be implemented in a similar fashion. In the complex number class with polar representation, computed values for the real and imaginary parts are achieved as follows. The instance variables `r` and `theta` are not declared readable or writable, and therefore cannot be accessed or updated by the clients. Instead, accessor methods `complex-real` and `complex-imag` are explicitly defined by the implementor of the class to compute and return the real and imaginary components from `r` and `theta`, as was done in Trellis/Owl. These methods are exactly what the old class had for accessing its instance variables, so the client code will still run as before.

In order to "update a computed value", an update function corresponding to the accessor function for that computed value (as required by setf) must be defined. For our example, the functions `complex-real-update` and `complex-imag-update` would be defined that take as argument the new value and compute and store the new values for `r` and `theta`. The correspondence between accessor and update functions is accomplished by the **defsetf** declaration available in Common Lisp:

```
(defsetf complex-real complex-real-update)
(defsetf complex-imag complex-imag-update)
```

C++

A C++ class contains *member* declarations, where a member is either a function declaration or a field of the class containing data (the latter corresponds to instance variables of the class). The *public* label separates the class body into two parts, as illustrated in the class template below. The names in the first, *private* part can be used only by member functions,[7] whereas anything declared in the public section of the class is available to the class's clients.

```
class class-name {
          /* private data and function declarations */
          ...
public
          /* public data and function declarations */
          ...
}
```

Data fields declared in the public part of a class can be accessed directly using dot notation. On the other hand, data fields declared in the private section of the class may only be accessed or updated if the programmer has provided accessor and update functions in the public section. Although C++ does not forbid the programmer from putting data representation information in

[7]A class declaration can also have *friend functions*. These are functions declared using the keyword **friend**. They are the same as ordinary functions except they can use private members of classes that name them as friends. They are useful when we need the same function to be a member of more than one class, which is not allowed in C++.

the public part of the class, the designer of the language states that "it is good policy to keep data private and present the public . . . interfaces as sets of functions" [Stroustrup 86b].

3.1.3. Distinction between Public and Private Operations

A class definition may contain operations that are only for internal use by other class operations; these should not constitute part of the class's external interface. An example of such "helper" operations are the methods for translating between cartesian and polar representation in figure 3-3 of the complex number class. Helper procedures should be private to the class, and therefore only invoked from within the class. In our running example, the external interface of the class should only contain the seven operations specified at the beginning of this section; we call this the *specified* interface of the complex number class. The *implementation* interface provided by an implementation of the class should contain exactly the same operations as in the specified interface. This is not possible in a language that has no mechanism for distinguishing between private operations and operations in the specified interface. Such languages are not strongly encapsulated because they do not allow implementations with minimal interfaces (that is, implementation interfaces equal to specified interfaces) to be defined. If clients of a class may invoke private operations, the implementor of the class is precluded from making changes that affect the semantics of the private operations if the new implementation is required to be compatible with existing code, even though the specified interface of the class remains the same.

Simula

In Simula, everything declared within a class can be accessed outside the class through use of the dot notation. Thus, there is no mechanism for implementing private procedures.

Smalltalk

In a class definition, methods that perform similar operations are grouped together into categories to indicate their common functionality. This categorization is intended to make the class description more readable, but it does not affect the operation of the class. The category *private* refers to messages introduced to support the implementation of other messages, and although programmers might adopt the convention that messages should not be sent to private methods from outside the class definition, this is not enforced by Smalltalk.

Flavors, CommonObjects

Methods cannot be marked as private in Flavors or CommonObjects. However, since both languages are built on top of Common Lisp, its package system can be used to distinguish between public and private interfaces by exporting the names of public methods.

Trellis/Owl

Private operations can be distinguished from public ones by using the keyword **private**.

C++

Functions defined in the public part of the class are available to clients of the class whereas those defined in the private part are only accessible to the members of the class.

3.1.4. Discussion

Table 3-1 summarizes how well the object-oriented programming languages considered in this section support encapsulation. It is clear that in order to do this a language must have a way of separating the public user interface from the private implementation details. An object should be viewed by its clients as a "black box" that can only be manipulated through a specific set of operations. Details about the object's representation, as well as operations defined for internal use by the object, should be hidden from the client. A language that enforces such information hiding minimizes the effect of changes to a class on its users and consequently maximizes its implementor's freedom to make changes.

3.2. Interface to Inheriting Clients

The division of a class into a *visible* public part and a *hidden* private part is desirable for instantiating clients but might be too constraining for inheriting clients. Some object-oriented languages have retained this binary division for instantiating classes but make everything public to the subclasses, thus compromising encapsulation. Other languages have recognized that a third alternative is required — *subtype-visibility*. An attribute of a class declared to be subtype-visible is visible to all its descendents in the inheritance hierarchy but invisible to instantiating clients.

The method by which conflicts of instance variable names due to inheritance are resolved also has an impact on encapsulation. For example, if it is illegal in a language to inherit the same

	Initialized Object Creation	*Visibility of Representation*	*Visibility of Operations*
Simula	number and type of instance variables visible	can be accessed directly	all visible
Smalltalk	representation not visible	can only be accessed through operations	can be marked private, but not enforced by language
Flavors	names of instance variables visible	can only be accessed through operations	all visible, but can use Common Lisp packages to export public ones
CommonObjects	representation not visible	can only be accessed through operations	same as in Flavors
Trellis/Owl	representation not visible	can only be accessed through operations	only public visible
C++	representation not visible	data in public part can be accessed directly	only public visible

Table 3-1: Interface to Instantiating Clients

variable name from more than one ancestor, renaming an instance variable in a class can cause inheriting clients to become illegal, even though the external interface of the class was not changed.

We now discuss these two issues for a number of OOPLs. In our examples throughout this section we will use C to refer to a class and S to refer to a subclass of that class.

3.2.1. Visibility of Superclass's Representation and Operations

Many OOPLs allow instance variables of a superclass to be directly accessed by name in a subclass. If the instance variables are accessible to the subclasses of a class, they are part of "the contract between the designer of the class and the designers of descendent classes" [Snyder 86a], therefore restricting the kind of changes the designer of the superclass can make without affecting descendent classes. Renaming, removing, or reinterpreting an instance variable can adversely affect descendent classes that depend on that instance variable. For example, renaming an instance variable in an ancestor may result in an error if one of the descendent's methods directly named that instance variable. Inherited instance variables should be accessed by operations in descendents just the same as in instantiating clients, since in this way encapsulation is not violated: the instance variables can be safely removed as long as the external operations are implemented by some other means.

The designer of a class might want certain instance variables to be accessible to subclasses but not to instantiating clients. If operations are used to access instance variables by both kinds of clients, the designer of a class must have a way to distinguish between operations that are visible to all clients and those that are only for use in inheriting clients. Subtype-visible operations are provided in some languages for just this purpose.

Besides adding new methods of its own, a subclass can override methods defined in its ancestors by defining a method with the same selector name. Therefore, a subclass S of class C has the same methods available to it as non-descendent clients of class C, with the exception of the methods overridden in S. When a new definition supersedes a definition inherited from a parent, it is useful to be able to invoke the parent's version of the operation. For example, if we have a `bordered-window` class that inherits from the class `window`, the `display` method for a bordered window might first invoke the `display` method in the parent, and then draw a border around it. In Smalltalk this is accomplished by using the pseudo-variable **super**, which causes the method lookup to start in the parent:

```
class       BorderedWindow
superclass  Window

"method to display a bordered window"
display
          super display.           "Use Window's display"
          self display-border      "Display the border"
          ...
```

The ability for a descendent class which has redefined an inherited operation to invoke the ancestor's version of the operation is important, and is supported in most languages.

Simula

S can be declared a subclass of class C by prefixing the class declaration for S with C. This operation is called *concatenation*. Any formal parameters in C are adjoined with any in S, as are the declarations and program statements of C and S. If there is a conflict of local names in A and B, then the compiler will systematically change those conflicting names in B. Instances of S have all the attributes of class C as well as the attributes of class S. As we saw in the previous section, *all* attributes of a class can be directly accessed by instantiating clients of a class. The same is true when the client is a subclass. Simula therefore presents the same interface to both inheriting and instantiating clients, but the interface is not minimal.

Smalltalk

A subclass in Smalltalk must provide a new name for itself, but it inherits both the variable declarations and methods of its superclass, its superclass' superclass, and so on. New instance variables may be added by a subclass, but the names must be different from any that have been inherited. This is because the subclass has direct access (i.e., it can reference by name) all instance variables defined in its ancestors.

Flavors

A flavor, `flavor1`, which inherits from two other flavors, `flavor2` and `flavor3`, is defined as follows:[8]

```
(defflavor flavor1 (<instance variables for flavor1> )
     ( flavor2 flavor3)
     ...  )
```

`Flavor1` is composed of three *components*[9]— itself, and the two flavors it inherits from. It inherits all of `flavor1`'s instance variables and methods as well as `flavor2`'s instance variables and methods. A method can access inherited instance variables by name, with the consequences for encapsulation as described in the beginning of this section.

CommonObjects

The designer of CommonObjects paid particular attention to encapsulation, and "plugged leaks" that existed in other languages. Instance variables may not be directly accessed by a descendent class, but may be accessed only by invoking accessor operations defined in the parent (either automatically generated through the use of the **:gettable** option or by the programmer).

Trellis/Owl

In Trellis/Owl access to instance variables (called components) is done through the operations

[8]Flavors has multiple inheritance.

[9]Unfortunately, the same terminology is used in Trellis/Owl for describing a different concept (see section 3.1.2). When we use the term "component" in this paper, we will ensure that the context makes it clear which meaning is intended.

get and put in both instantiating and inheriting clients. Components and operations can be marked (1) public, (2) private, or (3) subtype-visible. We discussed the difference between public and private attributes in the context of instantiating clients in section 3.1.3. Private operations are not visible to the subtypes, just as they are not visible to instantiating clients.

In Trellis/Owl, the supertype C's operations are textually copied to the subtype S, and names in the copied operations are interpreted in the context of S.[10] This means that if an inherited operation F calls an operation G, it will access the version of G appropriate for S and not C. If an inherited operation F in the subtype calls a non-visible operation, that operation is copied to the subtype but with a different name. The new name is chosen so that it will never cause conflicts with existing or future operations. All uses of the renamed operation are also changed to use the new name. As a consequence, inherited operations that use non-visible operations will continue to work without the programmer of the subtype having to be aware that such operations exist. The renamed operations continue to be copied down the type hierarchy as long as they are needed.

Subtype-visible operations are not as restrictive as private but not as general as public. Operations of this kind are not visible outside the defining type and its subtypes. Subtype-visible operations are inherited and can be redefined in a subtype. A subtype can also reduce the visibility of an inherited operation, for instance, making an inherited public operation into a private one, thus hiding it from all clients of that subtype.

C++

In C++, a subclass S of a class C can be defined as follows:

```
class S : public C {
        // private stuff for S
public:
        // public stuff for S
}
```

[10]Implementing inheritance by textually copying the code of the superclass to the subclass is common in statically-typed languages. However, the same effect of interpreting the inherited operations in the context of the original receiver is accomplished in dynamically-typed languages by binding **self** to the original receiver of the message. If a class S receives a message with selector F, and a method for F is found in the superclass C, then if the method for F contains an expression which sends a message with selector G to self, the search for method G starts in S.

Class C is said to be the *base* class for S, and conversely S is said to be *derived* from C. The word **public** in the class header makes C a *public* base class of S. This means that a public member of C is also a public member of S. Alternatively, one can declare a *private* base class by simply leaving out the word public in the class header. This would mean that public members of C are private to S, and therefore accessible only to S's member functions.

In C++, the corresponding construct to the subtype-visible attribute of Trellis/Owl is the *protected part*. For example, consider the following class definition for some kind of tree node:

```
 class node {
      // private stuff
protected:
      node* left;
      node* right;
      // more protected stuff
public:
      // public stuff }
```

The members defined in the protected part are inaccessible to the general user but available to any member function of a derived class. Public and protected members of the base class can be referred to as if they were members of the derived class. If the public and/or the protected parts of the class contain data field declarations (which is allowed in C++), the representation of the superclass C is visible to a subclass S, and S can directly access these fields using the dot notation.

3.2.2. Conflict Resolution of Inherited Instance Variables

Languages with multiple inheritance have rules to determine what happens if a subclass inherits instance variables with the same name from more than one parent. As we see below, the way this conflict is resolved in most OOPLs leads to a violation of encapsulation. Since this problem only comes up with multiple inheritance we will only discuss the languages Extended Smalltalk (this is an extension to Smalltalk with multiple inheritance [Borning 82]), Flavors, CommonObjects and Trellis/Owl in this section.

Extended Smalltalk

In Extended Smalltalk it is an error if a subclass inherits instance variables with the same name. This compromises encapsulation since a change in a class can cause descendents to become illegal even if the specified interface between the class and the descendent is not changed. For example, if an instance variable in a class C is renamed, it may cause a subclass S of C to become illegal if S also inherits from another class containing an instance variable with the same name as the renamed variable in C.

Flavors

The set of instance variables for the new flavor is the union of all the sets of instance variables in all the component flavors. If `flavor1` is defined as in section 3.2.1, and if both `flavor2` and `flavor3` have instance variables named x, then `flavor1` will have an instance variable named x, and any methods that refer to x will refer to this same instance variable. Thus different components of a flavor can communicate with one another using such a shared instance variable. This sharing creates an interdependency among the components that is not specified in the interfaces of the components. Changing a shared instance variable in a component changes the behavior of all other components that "see" it. Similar objections apply to merging inherited instance variables with instance variables defined locally in the class.

CommonObjects

Unlike Flavors, there is no merging of instance variables when a class inherits from multiple parents. For example, if two parents define an instance variable x, instances of the class will contain two instance variables x, one for each parent. There is no merging even if a class is inherited more than once through multiple paths. For example, if two parents of a class both inherit from a common ancestor A, instances of the class will contain two sets of the instance variables of A, one from each parent. This prevents exposure of instance variables outside the class definition.

Trellis/Owl

Inheriting the same component in Trellis/Owl is legal as long as all the supertypes containing that component agree on the interfaces of the `get` and `put` operations (the number and type of the arguments, and the type of the return value) defining that component. It is not clear (at least from the papers and reference manuals available to the author) whether the subtype would have one slot allocated for field components of the same name inherited from multiple supertypes (as in in Flavors) or one slot for each time the component is inherited (as in CommonObjects). If the supertypes do not agree, the programmer of the subtype must resolve the conflict by specifying which component he wants to inherit. This leads to similar problems with encapsulation as in Smalltalk.

3.2.3. Discussion

The analysis presented in this section is summarized in Table 3-2. In order to support encapsulation to inheriting clients, a language should enforce that data fields in ancestors should only be accessible through operations defined in the ancestor. Since the ancestor might want to hide such operations from casual, non-descendent clients, a language should have a way of making an operation visible only to its descendents. Resolving conflicts of inherited instance variable names should not be done in such a way that the inherited names are made visible to an inheriting client.

	Accessibility of Instance Variables	*Conflicting Instance Variable Names*	*Visibility of Methods*
Simula	by name allowed	not applicable	all visible
Smalltalk	by name allowed	error	all visible
Flavors	by name allowed	merged	all visible
CommonObjects	by accessor function only	keeps multiple sets of instance variables	all visible
Trellis/Owl	by accessor function only	programmer must resolve	only public and subtype-visible
C++	by name allowed	not applicable	only public and protected visible

Table 3-2: Interface to Inheriting Clients

Current OOPLs only support the two kinds of interfaces described in this subsection: one for instantiating clients and one for inheriting clients. It might be useful for a class to be able to present different external interfaces to various classes of clients, not necessarily based on the inheritance relation. Such a scheme could be used to implement protection domains, a mechanism used mainly in operating systems to give different access capabilities to different kinds of users [Levy 84]. An example, taken from [Jones 78] is that of a system providing telephone service. The customers would be one class of objects in the system. Other objects provide various kinds of telephone services to the customer. Examples of telephone services are the operator, telephone service repair, and the billing department. Each of these view the customer from a different perspective, and the operations available to them should reflect that perspective. The operator can look up telephone numbers, the telephone service office can perform the lookup operation as well as the change service operation, while the billing department can perform crediting and debiting operations.

3.3. Visibility of Inheritance

In this subsection we analyze several object-oriented languages to see whether they allow the use of inheritance in the construction of a class to be exposed to the class's clients (both instantiating and inheriting kinds). Snyder [Snyder 86c] identifies four ways in which the use of inheritance in a class is made visible to its clients: (1) inability of a subclass to exclude some of the operations defined in its superclass(es) from being inherited, (2) the rules for determining whether a class is a subtype of another class, (3) allowing a class to invoke operations defined in ancestors of that class, and (4) resolving conflicts when operations with the same name are inherited from more than one parent in languages with multiple inheritance.

We now look at these four issues and their effect on encapsulation in the languages Simula, Smalltalk, Flavors, CommonObjects, Trellis/Owl, and C++. We group the statically-typed languages, Simula, Trellis/Owl, and C++ together since much of the discussion is similar for the three of them.

3.3.1. Excluding operations

In most OOPLs, inheritance is *additive*, that is a subclass can add new attributes but cannot exclude operations defined in a superclass. To see that excluding operations is a useful property to have, consider the following scenario. Suppose we need a class implementing the abstract data type *stack*, supporting the operations *push*, *pop*, and *top*. If in our system there exists a class *deque*, containing the operations *push*, *pop*, *top*, *back-push*, *back-pop*, and *back*, the easiest way to build the class *stack* would be to make it inherit from *deque*, excluding the latter three operations. *Deque* contains operations that should not be part of the external interface of *stack*, but if the language has no means for excluding operations, there would be nothing to prevent the user of a *stack* from using one of the *back* operations. If the *stack* class were reimplemented such that it no longer inherited from *deque* (and therefore only had the stack operations), any client of *stack* that used the "excluded " deque operations would become incorrect.

Simula, Trellis/Owl, C++

Excluded operations are not allowed in Simula. A subclass can "hide" a procedure defined in a superclass by redefining it, giving an error message saying that the operation is not defined. Excluding an operation in this way, however, means that an erroneous use of the excluded operation is only caught at run-time. In a statically-typed language such as Simula, use of an operation that is not part of an object's interface should be caught at compile-time.

Trellis/Owl has an **exclude** statement; the statement "`exclude foo`" in a subtype means do not inherit the definition of `foo` from the supertype. The only operations that can be excluded are subtype-visible operations; public operations cannot be excluded because of the way subtyping relies on inheritance (as described below). Private operations are not visible in a subtype, so naturally they cannot be excluded.

In C++ it is possible to inherit public members of a base class in such a way that they do not become public members of the derived class, thereby excluding them from the interface of the derived class. This can be used to provide restricted interfaces to derived classes. For example, given the class `deque`:

```
class deque {
      // ...
public:
      void push(elem*);
      elem* pop();
      elem* top();
      void back-push(elem*);
      elem* back-pop();
      elem* back-top();
};
```

we can define `stack` as a derived class with only `push`, `pop`, and `top` as follows:

```
class stack : deque {        // note: just ":" not ": public", which means public
                             // members of deque are private members of stack
public:
      deque::push;           // make "push" a public member of stack
      deque::pop;            // make "pop" a public member of stack
      deque::top;            // make "pop" a public member of stack
}
```

Smalltalk

In Smalltalk, operations can be "excluded" in the same way as described for Simula, that is, by defining a method that gives an error message when it is invoked. Since Smalltalk is dynamically typed, this does not have the same disadvantages as in Simula. However, it makes programs less understandable: it is not obvious by looking at the method headings of a class (as allowed by the environment browser) that some of them are "not available".

Flavors

Excluding operations can be done as in Smalltalk and Simula.

CommonObjects

In CommonObjects, a class can specify which operations defined on its parents are or are not included in its own external interfaces. If a class `deque` has the methods: `:size`, `:front-push`, `:front-pop`, `:front-top`, `:back-push`, `:back-pop`, and `:back-top`, then a class `stack` can be defined as follows:

```
(define-type stack
      (:inherit-from deque
            (:methods :size )))

(define-method (stack :push) (element)
      (call-method (deque :front-push) element))

(define-method (stack :pop) ()
      (call-method (deque :front-pop)))

(define-method (stack :top) ()
      (call-method (deque :front-top)))
```

The **:methods** option specifies which operations are inherited, thereby preventing the "front" and "back" operations from being inherited. The front operations are then redefined in stack to give them the usual names: `push`, `pop`, and `top`. To be able to access `deque`'s front operations, which were not inherited, the **call-method** construct is used. **Call-method** takes two arguments: (1) the name of the parent from where the method lookup should start, and (2) the method to be invoked.

3.3.2. Subtyping

Subtyping is a method used in statically typed object-oriented languages to determine when objects of one class can be used in contexts expecting another class. If subtyping rules are based on inheritance, then reimplementing a class such that its position in the inheritance graph is changed can make clients of that class type-incorrect, even if the external interface of the class remains the same. Many languages that perform static type-checking allow a variable of type S to be assigned to a variable of type T if and only if S is a subclass of T, thus equating inheritance with subtyping. In the stack and deque example, if *stack* is implemented as a subclass of *deque*, then an assignment of a variable of type *stack* to a variable of type *deque* is legal. The same assignment would be incorrect if *stack* was reimplemented such that it was not a descendent of *deque*.

Simula, Trellis/Owl, C++

Trellis/Owl's subtyping rules are based only on public operations. The reason is that subtyping should be based on an object's behavior (as specified by the public part of a type), not on its implementation (specified by the private and subtype-visible parts). When a type S is declared to be a subtype of type T, the programmer is asserting that every object of type S is also of type T. Thus it is legal to assign a variable of type S to a variable of type T, but not vice versa. For example, if we have two types, `shape` and `circle`, and `circle` is a subtype of `shape`, then in the following code:

```
var x: shape;
var y: circle;

    ...

x := y;
```

the assignment is legal since a circle is also a shape, and all the operations applicable to a shape are applicable to a circle. However the assignment

```
y := x;
```

would cause a compiler error since shapes are not necessarily circles.

For these subtyping rules to work, Trellis/Owl requires that for every public operation F defined on T there must be a corresponding public operation F defined on S; this is the reason that public operations cannot be excluded in a subtype. Moreover, to allow all the operations applicable to a type to be also applicable to its subtype, there are constraints on how a function F can be redefined in the subtype S. If F is an operation defined in a type T with this specification:

operation F (me, a: Atype_T) returns (Rtype_T)

then F can be redefined in S by:

operation F (me, a: Atype_S) returns (Rtype_S)

where

$\texttt{Atype}_S$ is $\texttt{Atype}_T$ or a *supertype* of $\texttt{Atype}_T$, and
$\texttt{Rtype}_S$ is $\texttt{Rtype}_T$ or a *subtype* of $\texttt{Rtype}_T$.

Note that the number of arguments in both definitions of F must be the same, and either both have a return value or both do not. Since the subtype-of relation defines the inheritance graph in Trellis/Owl, subtyping is closely tied to inheritance.

The subtyping rules of Simula are similarly tied to the inheritance hierarchy.

In C++, if a class D is derived from a public base class B, then a variable of type D can be assigned to a variable of type B. The reverse is not possible. The same example assigning a `circle` variable to a `shape` variable and vice versa illustrates why this is so. If B were a private base class of D, then a variable of type D would not be assignable to one of type B. The reason is that a public member function of B can operate on a variable of type B, but is not accessible, and therefore not applicable, to variables of type D. These subtyping rules of C++, which differentiate between whether a subclass inherits the public operations of its superclass or not to define the legality of assignments, is the reason why public operations can be excluded in C++ without causing type-checking problems. However, subtyping still depends on the inheritance hierarchy — even if the behavior of some class A includes the behavior of class B, A is not a subtype of B unless A is derived from B, or from a subclass of B.

CommonObjects

In CommonObjects, the designer of a class is able to define the classes of which it is a subtype, independent of the inheritance relation. This is done by allowing Common Lisp's **typep** predicate (which takes two arguments, an object and a type specification, and returns true if and only if the object is of the specified type) to be defined by the programmer. If the object given to **typep** is an instance of a class and the type specification is a class name, the **:typep** operation is performed on the object, with the type specification as the argument, to determine the result of **typep**. Thus for the stack and deque example, **:typep** can be defined by the following predicate:

```
(define-method (stack :typep) (the-type)
     (equal the-type 'stack))

(define-method (deque :typep) (the-type)
     (or (equal the-type 'deque)
          (equal the-type 'stack)))
```

If the programmer doesn't define a **:typep** predicate, a default one is provided which returns true only if the object is an instance of the class given as argument.

3.3.3. Reference to ancestors

If a client can directly invoke an operation on an ancestor by naming the ancestor and the operation that is to be invoked, the use of inheritance in implementing the client is exposed. The reason is that the requested operation might not have been "passed down" from the named ancestor to the client — it could have been excluded by an intervening class. The only operations that should be visible to a client are those of its parents; these operations constitute the client's interface to its ancestors in the inheritance graph.

Simula, Trellis/Owl, C++

In all three languages, it is possible to invoke an operation F on a non-immediate ancestor T. This is done using the construct "`F'T`" in Trellis/Owl and the construct "`T :: F`" in C++. In Simula "`this T`" results in a reference to a local object of T, and so "`this T.F`" invokes the desired operation. This results in the function F being included in the interface of the client which contains this call, even if F was excluded in some class between T and the client.

Smalltalk

Invoking an operation on non-immediate ancestors is forbidden in Smalltalk. The pseudo-variable **super** can be used to invoke an operation on the parent: "`super message-selector`" specifies that the search for the method specified by `message-selector` should start in the parent class. In Extended Smalltalk, this mechanism is not sufficient: the name of the parent must also be specified since multiple parents may contain the same operation. A *compound* message selector, which consists of a class name, followed by a period, followed by the actual message selector (e.g. "`classfoo.bar`"), can be used. The compound message selector can only be used to name an operation on a parent; it cannot be used to invoke an operation on other ancestors.

Flavors

It is not possible to qualify an operation by the name of the ancestor where the method lookup should start. However, one can achieve the effect of using **super** in Smalltalk by *method* *combination.* This is a powerful mechanism whereby a method is defined as a combination of other methods. For example, a method can have *daemon* methods defined for it that are executed whenever the method is invoked. (This will be explained in detail in section 4.1, which deals with inheritance.)

CommonObjects

The **call-method** construct is available for invoking operations of the parent that were not inherited or overridden in the subclass. It is not possible to specify the name of a non-immediate ancestor as an argument to **call-method**.

3.3.4. Multiple inheritance

Inheriting operations from more than one parent can result in a conflict when operations with the same name are inherited from multiple parents, or from the same parent but along different paths. The way this conflict is resolved in some languages produces different results if the inheritance graph is changed, even though the external interfaces of the objects remain the same.

Trellis/Owl

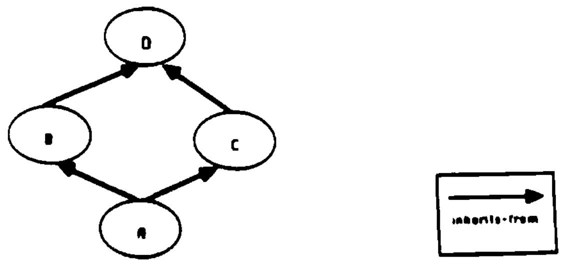
If an operation F is inherited from more than one supertype, the possibility of ambiguity arises. If all of the definitions of F in the supertypes are identical, there is no ambiguity. If the supertypes disagree as to the definition of F, the programmer must specify which definition of F he wants, or write a new implementation for F overriding the inherited F. This method of resolving conflicts does not expose the use of inheritance since the decision of when a multiply inherited operation is erroneous is not tied to the inheritance graph.

Extended Smalltalk

In Extended Smalltalk, it is an error if two or more methods are found corresponding to a received message in a client. But this is only true if the methods are actually different — if it is the same method inherited along multiple paths it is not a problem. In the case of conflicting methods, the programmer is responsible for choosing one of the methods or redefining it in the client. This exposes the use of inheritance as shown by the following example. Suppose we have a program with a class S which inherits operation F from superclasses C1 and C2, which in turn inherit F from A. Since the F inherited through C1 and C2 is the same operation, there is no problem. But if C2 is reimplemented such that it no longer inherits from A, but still provides an operation F with the same specification, the same program is erroneous.

Flavors

When a flavor is defined with one or more components, the Flavors system chooses an *ordering* of its components, including both the direct components (the ones listed in the flavor definition) and all others inherited from the direct components. To compute this ordering of components for some flavor A, a depth-first traversal of the inheritance graph rooted at A is done, with duplicates eliminated. The linearization of the graph transforms the partial ordering of components as defined by the flavors into a total order, with the result that unrelated flavors can be inserted before a class and its parents. To show how this compromises encapsulation, consider the following situation. The four flavors, A, B, C, and D are related by the following inheritance graph:



Suppose flavors B and D both have definitions for method `foo`. If one of flavor A's methods invokes `foo` on parent C, and the ordering chosen by the system is (A, B, C, D), then the method `foo` which is executed is the one in B. This is not what was intended. To correctly use parent C, A also needs to know the internal details of how C uses its parent D, (i.e., that C inherits D's `foo`), a fact that should not be visible to A.

CommonObjects

In CommonObjects, an error is signalled if a class attempts to inherit methods of the same name from multiple parents; the designer must explicitly select one method or redefine the method. The same rule applies even if the conflicting methods originate in the same ancestor, unlike Smalltalk.

3.3.5. Discussion

We summarize the results of this subsection in Table 3-3. When inheritance is used for reusing code, it should be possible for a class to reuse only part of another class by excluding those operations that do not apply to it from being inherited. There are two consequences of allowing operations to be excluded:

1. The external interface of a class is not necessarily equal to the interface of the parents together with the operations defined on the class. Therefore, non-immediate ancestors of a class cannot be accessed directly without violating encapsulation.
2. Subtyping rules cannot use the inheritance hierarchy to determine whether a class is a subtype of another class. If a class X inherited only *some* of the operations of a class Y, the behavior of X does not include that of Y, and so it is not correct to use an instance of X in a place where an instance of Y was expected.

CommonObjects has made a first attempt at separating the *type hierarchy* from the *inheritance hierarchy*. However, having the programmer define the subtyping rules himself means additional work for the programmer, and may be unreliable due to errors in the predicate definition. A formal semantic specification of behavior is needed to be able to correctly do behavioral subtyping; defining such a formal semantics is still an open research problem.

	Excluding Operations	*Subtyping*	*Reference to Ancestors*	*Multiple Inheritance*
Simula	by overriding	based on inheritance	to arbitrary ancestor	not applicable
Smalltalk	by overriding	not done	only to parents	error if duplicate from different source
Flavors	by overriding	not done	only to parents	linearizes graph
CommonObjects	allowed	programmer-defined	only to parents	always error if duplicate
Trellis/Owl	only subtype-visible can be excluded	based on inheritance	to arbitrary ancestor	error if duplicate with different specifications
C++	allowed	based on inheritance	to arbitrary ancestor	not applicable

Table 3-3: Visibility of Inheritance

4. Reusability and Extensibility

The high cost of building large software systems can be attributed to the fact that most software development efforts are done from scratch. Even though there are general facilities that are needed in many systems, such as hash tables, set types, sort functions, and so on, programmers write these over and over again for each application (sometimes more than once in a single application), maybe with some small detail of difference in each. In an ideal software development environment, new systems are built by "ordering components from [catalogs of software modules] and combining them, rather than reinventing the wheel every time" [Meyer 87]. In this section, we show that the object-oriented programming methodology is a step in the right direction in attaining such an ideal software development environment.

Component reuse is not restricted to the initial construction phase of a software system. It continues to play an important part in the maintenance of the system, which is by far the most expensive part of the life cycle of the system. Maintenance is a process of reusing modules across time, rather than across applications [Bassett 87]. Enhancing the system results in modifications to the system components in unforeseen ways, just as during system design reused components need to be tailored to fit the new application. Therefore, reusability and extensibility are interrelated; this is the reason for presenting them together in this section.

The simplest kind of reusability is to use an existing component *as is*. This is possible in most languages through the existence of subroutine libraries — libraries containing primitive routines specific to some domain, such as libraries containing mathematical functions, I/O libraries, and so on. This kind of reusability is limited, though, because the functionality provided by a library subroutine is fixed (with the exceptions of parameters). Therefore it can only be reused in the construction of a new program if there is a need for *exactly* the same behavior as that provided by the subroutine. Although this is useful in certain situations, such as writing mathematical software systems, it does not solve the reusability problem in general. The reason is that subroutine libraries do not allow the routine to be modified in ways unforeseen by its implementor.

This is exactly what defines a truly reusable component: it can be combined, adapted and modified to fit in a new application in ways unforeseen by the implementor of the component.

In addition, it must be clear what the component does in order for a programmer to determine if it is useful for reuse in his application.[11]

As we saw in the previous section, the *class* construct in object-oriented languages provides a data abstraction mechanism with a well-defined interface specifying what the the class does. Thus, the class is the starting point for the creation of reusable components in OOPLs. Classes can be combined and modified to fit a new application by means of inheritance and polymorphism respectively. We discuss these two issues in the next two subsections, and close this section with a comparison of reusability and extensibility in a non-object-oriented language, ADA.

4.1. Inheritance

When an existing class is not *exactly* what is required for a new application, the designer can make use of the existing class by customizing it in some way to fit its new purposes. There are three ways of performing such customization:

1. modify the original class definition,
2. make a copy of the original class definition and modify the copy, or
3. modify the original by augmentation.

The problem with the first approach is that the original abstraction becomes more and more complicated as it is tailored for use in various applications. The modifications usually consist of a case statement which executes different code depending on which application is currently using the class. This creates programs that are difficult to understand and to extend. Modifying a copy has the update problem usual with replicated data — changes to the original for correcting errors or for enhancement are not automatically made to the copy.

The third alternative is achieved in OOPLs by means of inheritance. A new abstraction is defined by specifying in a new class the difference between the new abstraction and a pre-existing class, and appending the new class to the old one by making the former a subclass of the

[11]The organization of the catalog of reusable components also has a large impact on whether the components will actually be used in the construction of new programs or not. A programmer will not reuse a piece of code if it takes him longer to find out (a) that the code exists, and (b) where the code is, than it would take for him to rewrite the code. This problem is addressed in [Prieto-Diaz 87], but is outside the scope of the paper.

latter. An important property of defining a new abstraction using inheritance is that it does not affect existing classes, and no code is replicated. Inheritance is therefore a major tool for reusing software.

Inheritance also promotes an *open software architecture*, that is, it facilitates open-ended sets of extensions to a basic architecture. This is important for long-lived systems because the system's functionality changes over time. The following reasons account for the need for changes: (1) no one has enough insight to build the system right the first time, and (2) the existence of the system and the insight gained from its usage create a demand for new or altered facilities. As Balzer points out in [Balzer 86], such enhancement of the system is the central activity in the lifecycle of a software system.

4.1.1. Single Inheritance

A new class can be built from a more primitive one in different ways, depending on how the new class differs from the old one. Curry and Ayers identify three general ways of how a subclass differs from the base class from which it is derived [Curry 84]:

- *extension* — the subclass *adds* instance variables and methods of its own, thus refining the base class abstraction.
- *variation* — the subclass *redefines* some of the features inherited from the base class.
- *specialization* — this is a combination of extension and variation.

Extension is the simplest kind of inheritance. It is often used in conjunction with a base class that defines primitives common to many other related classes. For example, a class `employee` can be used to hold the data and operations common to all kinds of employees:

```
class employee {
     char* name;
     short age;
     short department;
     int salary;
     ...
}
```

A manager, who is an employee but has some additional properties, can be defined by:

```
class manager : public employee {
     employee* group;          // people he manages
     short level;
     ...
}
```

The manager class inherits from the class employee; it has the features of an employee (name, age, etc.) in addition to the features level and group, which only managers have. This simple kind of inheritance, originated by Simula, is found in all OOPLs.

Sometimes, the base class provides the operations needed by the subclass, but the operations need to be implemented differently in the subclass. Variation is most often accomplished by *overriding* the inherited method, that is, defining the same operation (with the same name) in the subclass. In statically-typed languages there are constraints on how the inherited operations can be redefined in the subclass. The rules used in Trellis/Owl to determine how a superclass's (called supertype in Trellis/Owl) public operations can be redefined in a subclass (subtype) were discussed in section 3.3. These rules are required in order to determine the legality of an operation invocation on an object at compile-time; note, however, that the form of the object at run-time, which determines the actual operation that is invoked, might not be determinable until run-time.

C++ and Simula use *virtual functions* for run-time type resolution. In C++, a function declared as virtual in a base class can be redefined in each derived class.[12] Since a variable of type base class can denote an object of that class or any of its subclasses, type information is stored with base classes containing virtual function declarations so that the correct correspondence between the object and the functions applied to it can be guaranteed. Consider the class employee:

```
class employee {
      employee* next;
      char* name;
      short department;
      ...
public:
      virtual void print();
      ...
};
```

The keyword **virtual** indicates that the function print can have different versions for different classes derived from employee. The compiler and loader guarantee that the appropriate print function is invoked depending on the type of the object it is applied to. The type of the function is declared in the base class and cannot be redeclared in a derived class. A virtual

[12]Simula's virtual functions are similar, so we omit a discussion of them here.

function *must* be defined for the class in which it is first declared. The function to print out the employee's name and department looks as follows[13]:

```
void employee::print()
{
     cout << name << "\t" << department << "\n";
}
```

The virtual function can therefore be used even if no class is derived from its class, and a derived class that does not need a special version of a virtual function does not have to provide one. When deriving a class, one simply provides an appropriate function if it is needed. The `print` function can be redefined in `manager` to print the additional information:

```
void manager::print()
{
     employee::print();              // call print defined in class employee
     cout << "\tlevel " << level << "\n";
}
```

If we have a function `print-employee-list`, which calls the function `print` for each employee in an `employee` list, then the appropriate function is called depending on whether the type of the employee is manager or not. Since the list can be constructed dynamically, the determination of which `print` should be invoked must be resolved at run-time. This requires that type information is stored with each object of the class `employee`. It should be stressed that the function `print-employee-list` could have been written and compiled before the derived class `manager` was even thought of.

C++ is more restrictive than Trellis/Owl in how an inherited operation can be redefined in a subclass:

- The programmer has to "tell" the compiler which operations might be invoked on objects of different type by using the keyword **virtual**. This is done to minimize the space overhead; C++ only needs to keep the extra type information for classes that have virtual functions.
- The redefined functions must have *exactly* the same type as the virtual function in the base class; subtyping rules in C++ are therefore much simpler.

Sometimes neither extension nor variation alone is powerful enough to construct a new class from an old one. The manager class above is an example; it needed to both redefine the print

[13]The operator << writes its second argument into its first, in this case `cout`, which is the standard output stream.

operation and add the other fields, `level` and `group`, and any operations to manipulate those fields. This is an example of specialization.

A subclass can itself be a base class. For example, we can have:

```
class employee { ... };
class secretary : employee { ... };
class manager : employee { ... };
class director : manager { ... };
class vice-president : manager { ... };
class president : vice-president { ... };
```

Such a set of related classes is traditionally called a *class hierarchy*. In some languages (C++, Simula, and Smalltalk-80, for example), a class can inherit from a single base class only, so the hierarchy is a tree. As can be seen from the following example, sometimes a more general graph structure is required:

```
class temporary { ... };
class employee { ... };
class secretary : employee { ... };
class temporary-secretary : temporary : secretary { ... };
class consultant : temporary : employee { ... };
```

Languages with single inheritance can simulate this as shown below, but only at the expense of replicated code and data, the problem that inheritance was supposed to address in the first place:

```
class temporary_secretary: secretary {
      // data fields and operations for temporary class
      ...
}
class consultant: employee {
      // data fields and operations for temporary class
      ...
}
```

Another disadvantage with this solution is that for statically-typed languages, the fact that a `consultant` is a `temporary` is buried in the class definition, and therefore not known to the compiler. This prevents a variable of type `consultant` from being used where one of type `temporary` was needed, even though logically this is correct.

Many object-oriented languages have acknowledged this problem and allow the inheritance relation to form a directed acyclic graph where a class can inherit from more than one superclass. This is called *multiple inheritance*.

4.1.2. Multiple Inheritance

The main difference among OOPLs with multiple inheritance is what they do when an operation is inherited from more than one ancestor. The approaches to this problem fall into three categories:

1. Flavors, Loops [Stefik 86]: The approach taken by these languages is to compute a *class precedence list* by a depth-first traversal of the inheritance subgraph rooted at the class, and choose the method from the class with the highest precedence. The algorithm ensures that (1) a class always precedes its superclasses in the precedence list, and (2) the local ordering of any class's local precedence list is preserved. The *local precedence* of a class is determined by the specified order of the class's immediate superclasses as listed in the class definition.

The traversal algorithm is depth-first up to *joins*. For the inheritance graph of Figure 4-1, which illustrates the inheritance graph of the `temporary-secretary` class given previously, the precedence list would start with the class `temporary-secretary`, then proceed in a depth-first order up to, but not including the join, Object. The precedence order would therefore first have all the classes on the left branch (`temporary`), then the right branch (`secretary`, `employee`), then the join (`Object`), and so on. The reason for the up-to-joins rule is so that Object is put at the end of the precedence list, since the method from a descendent of Object should be preferred to the more general one found in Object.

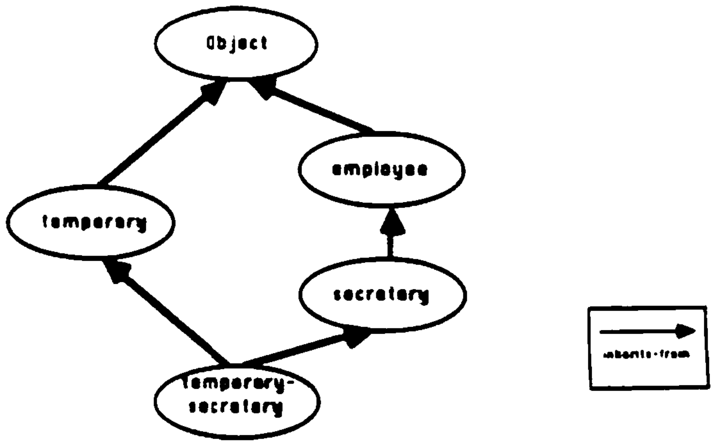


Figure 4-1: Inheritance Graph for temporary-secretary Class

2. CommonLoops [Bobrow 86]: The approach taken in this language is that the precedence list should not be built into the language, but should be under the control of the programmer. The metaclass of a class determines the algorithm for computing the class precedence list from the local precedence of the class being defined. If no algorithm is given in a class's metaclass, then the algorithm of the metaclass of `Class`[14] is used, which is the same algorithm used in Loops.

3. Trellis/Owl, Extended Smalltalk, CommonObjects: These languages take the position that no simple precedence relationship for multiple inheritance will work for all the cases, so none should be assumed at all. Whenever a method is provided by more than one superclass, the programmer of the subclass must explicitly indicate which one should be used.[15]

Although the previous algorithms are needed to determine which method to use when it is provided by more than one class, sometimes a method needs to be defined as a *combination* of other methods. The simplest form of *method combination* is through the use of **super**, or similar constructs. Sending a message to **super** indicates that the search for the method should start in the superclass of the class containing the method in which **super** was used. The `print` operation for `manager` is an example: it is made up of the parent's `print` function, and the additional code to print out the additional information about managers.

Loops extends this notion to allow more than two methods to be combined. The *fringe methods* of a class are the inherited methods that have not been redefined in the class. Using the keyword **SuperFringe** and a message selector, *all* the fringe methods for that selector are invoked in sequence.

Selective combination of methods can be done in Loops by a construct called **DoMethod**. DoMethod allows the invocation of any method from any class on any object. We have seen similar constructs for starting the lookup for an operation in a specified class in the languages Trellis/Owl and C++ (see section 3.3). Programs using this kind of method combination make strong assumptions about the names of other classes and the current configuration of the inheritance lattice. Changes to the inheritance lattice are likely to make such programs stop

[14]Class is the superclass of all the metaclasses. It describes the general nature of classes.

[15]As mentioned in subsection 3.3, Extended Smalltalk allows an operation to be inherited along multiple paths if it originated from the same ancestor, while in Trellis/Owl an operation can be inherited from multiple ancestors if the ancestors agree on the specification of the operation without programmer intervention.

working.

This problem is avoided in the mechanism provided by Flavors for method combination, a powerful feature of this language. A *method-combination type* can be associated with each function name. When a message with some particular selector is sent, the method-combination type sorts the available methods according to the component ordering, thus identifying more specific and less specific methods. The default component ordering is the one computed by the depth-first traversal algorithm described above. The method-combination type can also specify the reverse ordering of components, that is, the most basic flavor comes first. The type then chooses a subset of the methods (possibly all of them). The type specification is transformed into Lisp code that calls the selected methods, and determines what to do with the values returned by the methods. If the programmer does not specify a type for some function, then a default type is used.

Some frequently used built-in types of method combination are listed here:

- Divide the methods into three categories: primary methods, before-daemons, and after-daemons. Call all the before-daemons, then call the most specific primary method, then call all the after-daemons. This is the *daemon* method-combination type, and is the default type provided by the Flavors system.
- Call all the methods, most-specific first or in the reverse order. The combined method returns any values returned by the last of the methods.
- Call all the methods and return a list of their returned values.
- Call all the methods in turn until one returns a non-**nil** value. That value is returned, and none of the rest of the methods are called.

The function **define-method-combination** allows the programmer to define his own customized type of method combination. It consists of three parts: (1) the type name, (2) *method patterns* which select some subset of the available methods, and (3) the *body*, which is a declarative specification of how the selected methods are to be combined. The body is evaluated to produce detailed code to combine the methods. **Define-simple-method-combination** defines a simpler kind of method combination that simply calls all the methods, passing the values they return to a given function. For example, a type of method combination called :sum that uses the function + to add together the values returned by each of the methods, can be defined as follows:

```
(define-simple-method-combination :sum +)
```

It is important to note that defining the individual methods and defining a method-combination type are two different levels of abstraction in designing a system; the former is an example of programming in the small while the latter deals with the interaction of the pieces of the system, so it is an example of programming in the large. Flavors allows the two parts to be specified independently, thus preserving the modularity of the system. When defining a method, the programmer only thinks about what that method must do itself, and not about the details of its interaction with other methods that are not part of a defined interface. When specifying a method-combination type, the programmer only thinks about how the methods will interact, and not about the internal details of each method.

Some languages provide a category of classes that define a particular feature of an object. Examples of such classes include the *mixins* of the Flavors system and the *traits* of the Traits language [Curry 82]. In the following discussion, the word *mixin* is used to refer to this general category of classes, not specifically to the Flavors feature. A mixin cannot be instantiated because it is not a complete description. To use mixins, a new class is constructed from the mixins for the desired characteristics and an appropriate base class, and then instances of that class can be created.

4.1.3. Dynamic Combination of Behavior

One problem with inheritance and method-combination as described previously is that the behavior of an instance is fixed at instance creation time. This is undesirable in long-running systems, where run-time events can cause the behavior of the objects in the system to change over time. As an example, consider an object-oriented system modelling a corporation. An engineer in the company is represented as an instance of the `engineer` class which inherits from the `employee` class and the `engineer-mixin`. If the engineer is promoted to the managerial level, he should be in the the `manager-engineer` class, which inherits from the base class `employee`, and the two features `manager-mixin` and `engineer-mixin`. This is not possible with mixins because this would involve changing the membership of an instance from one class to another, which is not possible in existing systems.

A possible solution is to *coerce* an instance from one class to another. Thus the run-time event "`promote Joe`" would send a `<coerce 'manager-engineer>` message to `Joe` with values for the instance variables needed for the new class. This would change the membership of instance `Joe` from the class `engineer` to the class `manager-engineer`. However, the semantics of promotion depends on what the initial class is: a secretary is promoted to an administrative assistant, a manager to a vice-president, and so on. Writing the promotion function, which sends a coercion message to change the promoted employee's class to the new class, would amount to a bunch of conditionals to distinguish between all the cases of what the initial class is.

In [Hendler 86a], Hendler observes that the root of the problem is due to the fact that mixins are combined with classes to produce other subclasses, thereby forcing an instance to change class membership when it gains new abilities. He proposes that mixins should be used to provide additional functionality to instances, not to classes. Hendler calls this use of mixins *enhancement*. Thus we can create an employee, or any subclass thereof, and enhance him with the capabilities of an engineer. The promotion function no longer needs to be an enormous case statement; to promote `Joe`, the instance representing `Joe` is enhanced with the functionality appropriate for managers.

Hendler's concept of enhancement extends the way instances of a class can vary from each other. Usually, instances of a class can differ only in the values of their instance variables; they have the same number and type of instance variables, and they understand the same set of messages. This is no longer true if we can selectively add functionality to instances; instances of a class can have a different number of instance variables, and can respond to different messages. This approach is similar to that advocated by *prototype* languages, the most prominent of which are the actor languages [Agha 86, Lieberman 81, Lieberman 86].

There is no distinction between classes and instances in actor languages. All objects in the system are called actors. Each actor can respond to a **CREATE** message to make a new actor similar to itself. The parameters to CREATE specify any new information that differentiates the new actor from the creator. The creator actor is called the *proxy* for the created one. When an object receives a request to perform an operation, it first checks its personal behavior to see if it can satisfy the request. If not, it *delegates* the request to its proxy to invoke more general knowledge. Delegation provides the capability for sharing common knowledge among objects in actor languages, similar to inheritance in other OOPLs.

Each actor knows the names of other actors that it can communicate with. These are called the *acquaintances* of the actor. When a new object is created, it gets an initial set of acquaintance names from the creator. However, acquaintance names can be freely communicated in messages, so an actor can acquire new ones during its lifetime. This capability allows actor systems to have a very dynamic communication pattern. The system can reorganize itself as new events happen by changing the set of acquaintances of various actors. Using the same example as before, the `promote` actor can send a message to `Joe`, the engineer, giving it the names of the acquaintances that represent the functionality (data and methods) of a manager. From then on, `Joe` can communicate with these actors to obtain the behavior of a manager.

4.2. Polymorphism

Polymorphism is an important feature of all object-oriented languages that allows the definition of flexible software elements amenable to extension and reuse. A *polymorphic operation* is one that has multiple meanings depending on the type of its arguments [Cardelli 85]. In a language like Smalltalk, a variable or expression representing the receiver of a message may dynamically vary in type. It is up to the receiver to decide how to respond to the message. The method that is executed, as specified by the message selector, is *directly associated* with the type of the receiver. Therefore, different results will be produced depending on the type of the object receiving the message. This is called *simple polymorphism*: the operation invoked is dependent on the type of only one argument, the receiver of the object. *Multiple polymorphism*, on the other hand, is the ability to execute different functions based on the types of more than one of the function's arguments.

4.2.1. Simple Polymorphism

We illustrate polymorphism with an example — a graphics editor. The editor can manipulate different kind of objects: geometric objects such as squares, circles, lines, and so on, or text objects, which are strings of characters. These different classes of objects are arranged in the hierarchy shown in figure 4-2.

An instance of class `Graphics Object` contains instance variables `x-coordinate` and

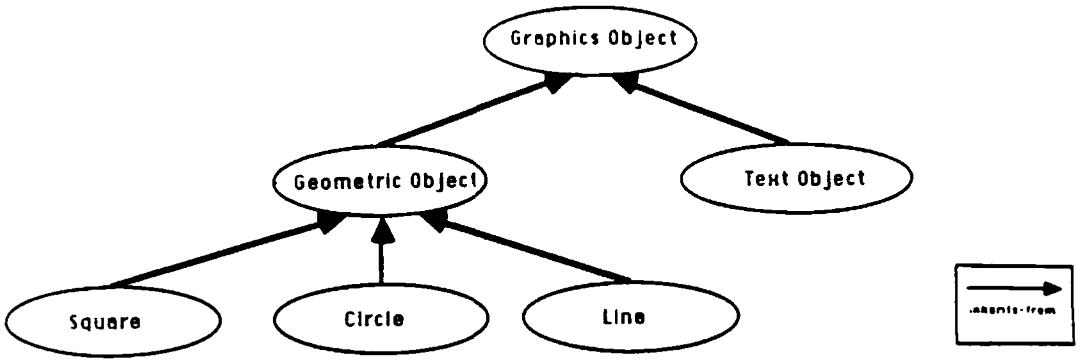


Figure 4-2: Inheritance Graph for Displayable Objects

y-coordinate which indicate the position of the object on the screen. It responds to the message location by returning these coordinates. An instance of Text Object provides a font index and an emphasis (italic, bold, underline). Geometric object instances contain color and pen (dashed, solid) which determine how the outline of the shape is displayed. Text Object and each subclass of Geometric Object also provide the following methods:

- ***draw*** — displays the object on the screen.
- ***reduce*** — reduces the object by a percentage given as an argument.
- ***enlarge*** — enlarges the object as specified by the argument.
- ***move*** — moves the object to the position specified by the argument.

These methods are implemented quite differently in the different classes. For example, the draw method for a square determines the current position of its top left corner (by sending a location message to itself), then uses the color and pen information to draw four lines, assuming it has instance variables containing the length of its sides and its orientation. The draw method for an instance of Text Object is quite different. The current position is found the same way by sending the location message to itself, but this position indicates the bottom left corner of the first character of the text. It uses the font and emphasis information to display the text on the screen. Similar differences exist in the implementations of the other methods for the various objects.

The editor class would have a method, mainLoop, which looks as follows in Smalltalk:

mainLoop
```
    | graphicObject command |
    self Initialize.
    self DrawMenu.
    [true]
        whileTrue:
            [graphicObject <- pickFromCursorPosition.
             command <- pickFromMenu.
             graphicObject command]
```

Each time through this loop, the variable `graphicObject` can represent a different object, depending on where the user has placed the cursor. If the cursor is placed on a text object then `graphicObject` refers to a text object. If the command `reduce` is selected and given an argument of `0.25`, the font variable is reduced by one fourth and the text is drawn on the screen again in the smaller font. If, however, the cursor is placed on a square and the same command is selected, the side-length variable is reduced by a fourth and the smaller square is redisplayed.

Extending the graphics editor to handle a different kind of object, for example, an ellipse, is easily done in a language supporting polymorphism. All that one needs to do is to define a new class `ellipse` which inherits from `Geometric Object` with instance variables holding information about the ellipse (the center and two radii), and implementations of the methods mentioned above for manipulating the ellipse. The graphics editor can now be used on ellipses without any changes to the rest of the code.

In essence, a polymorphic function has an implicit or explicit type parameter which determines which function is to be invoked. In Smalltalk, the language we used to present the graphics editor example, the type parameter is implicitly specified by the class of the receiver of the message; the class of the receiver determines the actual method that is executed. In other OOPLs that use conventional function call instead of message sending, the class (or type) of the first argument serves the same purpose.

4.2.2. Multiple Polymorphism

The simple polymorphism exhibited in the graphical editor example above is not sufficiently powerful to handle all the extensions that may be needed in the evolution of a software system. For instance, extending the functionality of the graphics editor to allow graphical objects to be displayed on various devices, such as an ascii display, a bit-mapped display, or a printer, leads to problems if only simple polymorphism is available. The reason is that we now have two polymorphic variables (that is, variables that may have more than one type), the one holding the graphical object and the one holding the display device. This gives rise to the doubly polymorphic interaction illustrated in figure 4-3.

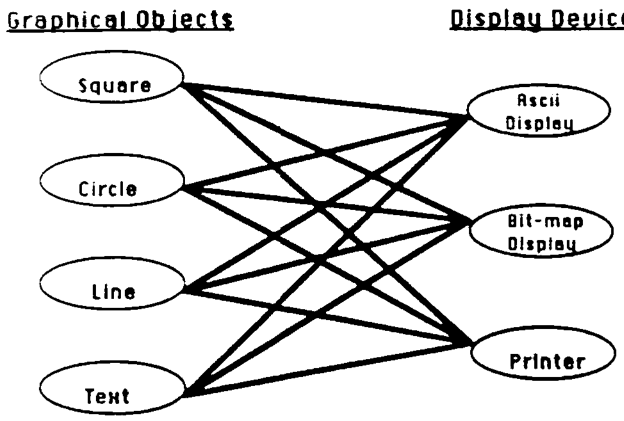


Figure 4-3: Interaction between Two Polymorphic Variables

In languages such as Smalltalk, which only support polymorphism of message receivers, not of arguments, this can be dealt with by putting in code for each graphical object method that knows how to deal with all display devices, as illustrated for `square` below:

```
"draw method for square"
draw: aPort
      aPort isMemberOf: AsciiDisplay
         ifTrue: ["code for drawing square on ascii display"] .
      aPort isMemberOf: BitmapDisplay
         ifTrue: ["code for drawing square on bit-mapped display"] .
      aPort isMemberOf: Printer
         ifTrue: ["code for drawing square on printer"] .
```

This solution results in a system that is not easily extensible: adding another kind of display device entails changing all the `draw` methods of all graphical objects. Languages with only simple polymorphism, thus, facilitate extension in only one dimension. As we will see in section 4.3, the kind of polymorphism that is found in object-oriented languages can be simulated in conventional languages like Ada in a similar fashion, with the same unacceptable results.

CommonLoops [Bobrow 86], unlike most other object-oriented programming languages, handles multiple polymorphism. A method can be specified in terms of the types of any number of arguments. For example, the method `draw` for a square object and a printer display device looks as follows:

```
(defmeth draw ((graphicalObject square) (aPort printer))
      ; code for drawing a square on a printer)
```

This method has two type specifiers: the first is a square and the second a printer. The code for this method is invoked for arguments of type square and printer, or any of their subtypes. Similar methods can be defined for the other combination of graphical objects and display devices. A `draw` operation is called as follows:

```
(draw obj port)
```

This is interpreted as:

```
(funcall
   (method-specified-by 'draw
                          (type-of obj)
                          (type-of port))
    obj port)
```

For any set of arguments in the function call, there may be several methods whose type specification match because of subclassing. The most specific applicable method is called. Method specificity is determined by the specificity of the leftmost type specifiers which differ.

Ingalls proposed a general solution to deal with multiple polymorphism in object-oriented programming languages [Ingalls 86]. He notes that each message transmission reduces a polymorphic variable to a monomorphic one (that is, a variable with exactly one type) by the type dispatch inherent in method lookup. Therefore, for the case of two polymorphic variables as in the graphical objects and display device example, we need to send two messages to reduce the double polymorphism. A *relay method* is defined for each graphical object that effects a further dispatch on the display device type as follows:

```
classname    Square

      "draw method for square"
      draw: aPort
            aPort drawSquare: self

classname    Circle

      "draw method for circle"
      draw: aPort
            aPort drawCircle: self

classname    Text

      "draw method for text"
      draw: aPort
            aPort drawText: self

... similarly for other graphical objects ...
```

We then define methods for each display device class to draw the different graphical objects:

```
classname    AsciiDisplay

      drawSquare: aSquare
            "code to draw a square on ascii display"

classname    AsciiDisplay

      drawText: aText
            "code to draw text on ascii display"

... similar for other graphical objects ...

classname    Printer

      drawSquare: aSquare
            "code to draw a square on printer"

classname    Printer

      drawText: aText
            "code to draw text on printer"

... similar for other graphical objects ...
```

This solution preserves the modularity of object-oriented programming style. If a new graphical object needs to be added to the system, one only needs to define the relay message in the new class for the object, and the appropriate implementation method in each of the display device classes. Adding a new display device is similar. The same technique can be used to reduce higher degrees of polymorphism as well.

4.3. Comparison to Reusability and Extensibility in ADA

In this section we will compare the facilities found in Ada for reusability and extensibility with those of OOPLs described in the previous two sections. The reason for selecting Ada for this comparison is that among languages that are not object-oriented, it is one of the most advanced in its support for reusability and extensibility.

Ada has two constructs for structuring a program: the subprogram and the package. The former, equivalent to the procedure or function in Pascal, provides control abstraction, that is, the ability to treat a sequence of actions as a single one. The package provides data abstraction: type definitions for a user-defined data type can be grouped together with the subprograms that operate on that data type in one package, and the entire structure can then be treated as a built-in type.

Both subprograms and packages can be reused as is; for example, if a new application needs to use a data type that was defined as an abstract data type in a previous program, then the old abstraction can be reused if it is exactly what is needed in the new application. However, as we mentioned in the beginning of this section, this accounts for only a small percentage of the times existing program fragments could be reused, because of the requirement that the old piece and the new use match exactly.

Ada has *generic* subprograms and packages, which relax the "as is" condition on reusability somewhat. Generic units (either subprograms or packages) allow types to be parameterized, thus factoring out the dependency of the units on types. For example, a generic procedure to swap two items can be written as follows in Ada:

```
generic type item is private;[16]
procedure swap(x,y: in out item) is
    temp: item;
begin
    temp := y;
    y := x;
    x := temp;
end
```

[16]Item is declared private; this means that the only operations allowed on items in the unit are assignments and tests for equality and inequality. If other operations were needed, additional generic parameters would need to be supplied so that an actual operator for the type used for item would be given when the procedure is instantiated.

Generic units are templates; the swap procedure given above cannot be invoked as it is. It first must be instantiated and a particular type specified for item. For example, to swap two integers, we could instantiate the generic swap procedure by the following declaration:

```
procedure swapinteger is new swap(integer);
```

This results in a distinct procedure being generated at compile-time with the type item replaced with the type integer. Thus, the generic facility in Ada is similar to a macro expansion feature.

The reusability and extensibility provided by generic units in Ada is not as general as provided by classes and inheritance in OOPLs for two reasons: (1) There are only two levels of reusable units, the generic module which must be instantiated before it is used, and the fully instantiated module which cannot be further refined. (2) Since the generic unit is instantiated at compile-time, it is not possible to specify an operation on a generic object that will execute a different operation depending on the type of the object at run-time. We now expound on the second statement, and describe why the polymorphism found in Ada is not as powerful as that in OOPLs for reusability and extensibility [Hendler 86b, Meyer 86]..

Ada supports two kinds of polymorphism: overloading of subprogram names and generic units. Using overloading, we can define a number of draw procedures that take different types of objects as parameters, for example:

```
procedure draw(graphicObject: in SQUARE);
procedure draw(graphicObject: in CIRCLE);
...
```

Since the subprograms are distinguished by the type of at least one operand, no ambiguity arises. However, this solution falls short of providing true polymorphic entities as in object-oriented programming languages. The reason is that overloading resolution occurs at compile time. There is no way to implement the mainLoop of the graphics editor example in Ada using overloaded procedures since the type of graphicObject is not known at compile time. Note that this is not an inherent restriction of statically-typed languages. In fact, in C++, a statically-typed object-oriented language, this is not a problem because a variable declared of type graphicObject can denote any object of this type or of any of its subclass, thus achieving the same kind of polymorphism discussed for Smalltalk.

Ada's generic procedures do not work for the same reason. Generic instantiation is performed at compile time with actual type values that must be determinable at compile time.

The only feature of Ada which could be used to emulate the polymorphism of the graphics editor example has nothing to do with overloading or genericity; it is the record with variant type.

```
type GRAPHICAL_OBJECT(object: GRAPHICAL_OBJECT_TYPE) is
      record
            ... fields common to all graphical object types ...
            case object is
                  when square => ... fields for square objects ...;
                  when circle => ... fields for circle objects ...;
                  ... other cases ...;
            end case
      end record
```

GRAPHICAL_OBJECT_TYPE is an enumeration type with elements square, circle, line, text, and so on. There would be a single version of each procedure on graphical objects (draw, move, reduce, etc.), each containing a case discrimination of the form:

```
case object is
      when square => ... action for square objects ...;
      when circle => ... action for circle objects ...;
      ... other cases ...;
end case
```

This solution creates "closed" software systems, that is, systems that are hard to extend. Addition of another kind of graphics object to the editor is difficult because it would involve changing the variant type and all discriminations that are scattered throughout the program.

This solution has another weakness, namely that it depends on the programmer manipulating types in a way that cannot be checked by the compiler, a disadvantage for statically-typed languages such as Ada. This can lead to two kinds of errors in large programs. The first is a failure to check what the type of the object is before calling the routines to operate on the object. The second is the failure to account for all the possible types of objects in the case statement. More of these kinds of errors are introduced in large than in small programs, and more in programs being modified by someone besides the original implementor (an inevitable situation in large systems). These kinds of errors are obviated in a language such as C++, where the burden of ensuring the correct correspondence between the operation and the object is left to the compiler (see section 4.1.1 for a discussion of how this is achieved by virtual functions), which is much more proficient at checking these details than the human programmer.

5. Conclusion

In this paper we discussed how encapsulation, reusability, and extensibility are supported in object-oriented programming languages. Encapsulation is the strict enforcement of information-hiding principles to minimize interdependencies among separate parts of a system. At first glance, this does not seem to be an issue for object-oriented systems, which are systems structured around objects that can only be manipulated by operations specified in the object's interfaces. In fact, as we saw in section 3.1, most OOPLs enforce that an object's internal representation is not visible to other objects in the system. A strongly encapsulated object is important for minimizing the effect of changes in one object on other objects in the system. Moreover, the definition of objects as almost independent units that communicate only through well-defined interfaces creates more understandable systems: the objects can be understood separately, and the whole can be understood only through the interface definitions. Unfortunately, in many languages, inheritance negatively affects encapsulation by leaking implementation information outside the entity being defined.

One cause of this problem is that most languages allow inheriting clients to more freely access internal representation and internal operations of an object in order to take full advantage of inherited data and code. This can be solved if the language provides a mechanism for defining a second kind of interface in an object, and enforces that interaction between an object and its descendents occurs only via this interface. This notion can be extended by allowing an object to define different interfaces to different categories of users, thus providing different levels of protection of the data it encapsulates.

The second problem of inheritance with respect to encapsulation arises from the two uses of inheritance in an object-oriented system. A user of an object-oriented system only needs to know the logical structure of the system, that is, how objects in the system relate to one another according to the behavior specified in their interfaces. On the other hand, an implementor of an object-oriented system needs to know the physical structure of the system, that is, how the objects in the system are implemented, in particular, how they relate to one another according to how they are implemented. In this latter view, the use of inheritance in defining a new class that inherits data and code from existing ones is an implementation decision that should be hidden from the user. Most languages confuse these two views, and have only one inheritance hierarchy that is a combination of the two. If the logical hierarchy and the physical hierarchy are forced to be the same, one usually wins out, with the result that either encapsulation is violated and the user sees implementation details, or the implementation is inefficient because it follows the logical hierarchy. The decoupling of the two hierarchies is one of the objectives of Exemplar Based Smalltalk [LaLonde 86]; this work is still at an early stage, and it is not possible to evaluate the results published so far.

Reusability and extensibility are both facilitated by inheritance. Reusing an existing class that is exactly what is needed is not any different from ordinary languages where one can use an existing procedure or module whose specifications meet the requirements. The novel feature of inheritance for reusability in OOPLs is that a programmer can reuse a class for purposes (slightly) different than what it was originally created for by modifying it through subclassing. This allows a class in an object-oriented system to be reused and extended in ways that the original designer of the class need not have anticipated. This is different from reusable units in languages that do not have inheritance, for example generic procedures, where the designer of the generic procedure must specify in advance the kind of types the procedure can be instantiated with.

Besides being able to extend the functionality of a system by subclassing, extensibility of a system is facilitated by the "message-sending" semantics, where the object that receives the message determines the method that is executed. Thus, the same message-sending operation can result in different executions based on the kind of object the receiver is. The extension of a system to handle a new kind of object is easily done by adding a class defining the behavior of the new object, including methods for handling the system's messages, without having to modify the existing code in any way. In statically-typed languages, the message receiver is declared to be of a particular class; at run-time, it can be an instance of that class, or of any of that class's descendents. An interesting issue to be explored in future work is whether this restricts extensibility, and if so what constructs are needed in order to retain the benefits of static type-checking while allowing the development of extensible systems.

The lesson learned from this study is that as a whole, object-oriented programming languages offer good support for encapsulation, reusability and extensibility, which makes them good candidates for use in the development of large software systems. This is substantiated by reported experience with implementing several large software systems in an object-oriented language. These include the Smalltalk language and environment, implemented in Smalltalk itself [Goldberg 84]; Intermedia, an object-oriented hypermedia system and applications framework written in Inheritance C, an object-oriented extension to C (75000 lines of code) [Meyrowitz 86]; and the Application Accelerator Illustration System, an integrated CAD environment that supports the development of application-specific integrated circuits, implemented in Smalltalk-80 (17,000 lines of code) [Miller 86]. The implementors of these systems report positive experiences with using OOPLs as the implementation languages. The systems were more consistent, understandable and modular. An increase in productivity was also observed. This was attributed to the ability to work independently because of class modularity, and to reuse of major portions of the code with minor changes made possible by inheritance.

References

[Agha 86]	Gul Agha. An Overview of Actor Languages. *ACM Sigplan Notices*. 21(10)October, 1986. Special Issue on the Object-Oriented Programming Workshop, June 9-13, 1986.
[Balzer 86]	Robert Balzer. Program Enhancement A Position Paper. *ACM Sigsoft Software Engineering Notes*, 11(4)August, 1986.
[Bassett 87]	Paul G. Bassett. Frame-Based Software Engineering. *IEEE Software*, July, 1987.
[Bobrow 86]	Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik and Frank Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In *Object-Oriented Programming Systems, Languages and Applications 1986 Conference Proceedings*. Association for Computing Machinery, September, 1986.
[Borning 82]	Alan H. Borning and Daniel H. Ingalls. Multiple Inheritance in Smalltalk-80. In *Proceedings of the National Conference on Artificial Intelligence*. 1982.
[Brachman 85]	Ronald J. Brachman. I lied about the Trees -- Or, Defaults and Definitions in Knowledge Representation. *AI Magazine*, , 1985.
[Cardelli 85]	Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, December, 1985.
[Cox 84]	Brad J. Cox. Message/Object Programming: An Evolutionary Change in Programming Technology. *IEEE Software*, January, 1984.
[Curry 82]	Gael Curry, Larry Baer, Daniel Lipkie, and Bruce Lee. Traits: An Approach to Multiple-Inheritance Subclassing. In *Proceedings of the SIGOA Conference on Office Automation Systems*. Association for Computing Machinery, April, 1982.
[Curry 84]	Gael A. Curry and Robert M. Ayers. Experience with Traits in the Xerox Star Workstation . *IEEE Transactions on Software Engineering*, September, 1984.
[Dahl 70]	Ole-Johan Dahl, Bjorn Byhrhaug and Kristen Nygaard. *Common Base Language*. Technical Report Publication No. S-22, Norwegian Computing Center, October, 1970.

[DeRemer 76] Frank DeRemer and Hans H. Kron.
Programming-in-the-Large Versus Programming-in-the-Small.
IEEE Transactions on Software Engineering, June, 1976.

[Doyle 86] K. Doyle, B. Haynes, M. Lentczner, L. Rosenstein.
An Object Oriented Approach to Macintosh Application Development.
BIGRE + GLOBULE, (48 (ISSN 0221-5225))January, 1986.
Proceedings of the 3rd Working Session on Object Oriented Languages sponsored by AFCET and IRCAM.

[Goldberg 83] Adele Goldberg and David Robson.
Smalltalk-80: The Language and its Implementation.
Addison Wesley, 1983.

[Goldberg 84] Adele Goldberg.
Smalltalk-80: The Interactive Programming Environment.
Addison Wesley, 1984.

[Habermann 83] A Nico Habermann and Dewayne E. Perry.
Ada for Experienced Programmers.
Addison Wesley, Reading, Massachusetts, 1983.

[Hendler 86a] James Hendler.
Enhancement for Multiple-Inheritance.
ACM Sigplan Notices, 21(10)October, 1986.
Special Issue on the Object-Oriented Programming Workshop, June 9-13, 1986.

[Hendler 86b] James A. Hendler and Peter Wegner.
Viewing Object-Oriented Programming as an Enhancement of Data Abstraction Methodology.
In *Proceedings of the Nineteenth Annual International Conference on System Sciences.* January", 1986.

[Ingalls 86] Daniel H.H. Ingalls.
A Simple Technique for Handling Multiple Polymorphism.
In *Object-Oriented Programming Systems, Languages and Applications 1986 Conference Proceedings.* Association for Computing Machinery, September, 1986.

[Jones 78] Anita K. Jones.
The Object Model: A Conceptual Tool for Structuring Software.
Operating Systems An Advanced Course.
Springer-Verlag, 1978.

[Keene 85] Sonya E. Keene and David A. Moon.
Flavors: Object-Oriented Programming on Symbolics Computers.
In *Common Lisp Conference Proceedings.* Boston, MA, 1985.

[LaLonde 86] Wilf R. LaLonde, Dave A. Thomas and John R. Pugh.
An Exemplar Based Smalltalk.
In *Object-Oriented Programming Systems, Languages and Applications 1986 Conference Proceedings.* Association for Computing Machinery, September, 1986.

[Levy 84] Henry M. Levy.
Capability-based Computer Systems.
Digital Press, 1984.

[Lieberman 81] Henry Lieberman.
A Preview of Act 1.
Technical Report, Massachusetts Institute of Technology, June, 1981.

[Lieberman 86] Henry Lieberman.
Using Prototypical Objects to Implemenent Shared Behavior in Object Oriented Systems.
In *Object-Oriented Programming Systems, Languages and Applications 1986 Conference Proceedings.* Association for Computing Machinery, September, 1986.

[Liskov 81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler, and A. Snyder.
CLU Reference Manual.
Springer-Verlag, Berlin Heidelberg , 1981.

[Meyer 85] Bertrand Meyer.
Eiffel: A Language for Software Engineering.
Technical Report, University of California, Santa Barbara, Computer Science Department, November, 1985.

[Meyer 86] Bertrand Meyer.
Genericity versus Inheritance.
In *Object-Oriented Programming Systems, Languages and Applications 1986 Conference Proceedings.* Association for Computing Machinery, September, 1986.

[Meyer 87] Bertrand Meyer.
Reusability: The Case for Object-Oriented Design.
IEEE Software, March, 1987.

[Meyrowitz 86] Norman Meyrowitz.
Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia Syste and Applications Framework.
In *Object-Oriented Programming Systems, Languages and Applications 1986 Conference Proceedings.* Association for Computing Machinery, September, 1986.

[Miller 86] Michael S. Miller, Howard Cunningham, Chan Lee, Stven R. Vegdahl,
The Application Acceleratot Illustration System.
In *Object-Oriented Programming Systems, Languages and Applications 1986 Conference Proceedings.* Association for Computing Machinery, September, 1986.

[Moon 86] David A. Moon.
Object-Oriented Programming with Flavors.
In *Object-Oriented Programming Systems, Languages and Applications 1986 Conference Proceedings.* Association for Computing Machinery, September, 1986.

[O'Brien 85] Patrick O'Brien.
Trellis Object-Based Environment Language Tutorial.
Technical Report, Eastern Research Lab, Digital Equipment Corporation, November, 1985.

[Prieto-Diaz 87] R. Prieto-Diaz and P. Freeman.
Classifying Software for Reusability.
IEEE Software, 4(1)January, 1987.

[Schaffert 85] Craig Schaffert, Topher Cooper and Carrie Wilpolt.
Trellis Object-Based Environment Language Reference Manual.
Technical Report, Eastern Research Lab, November, 1985.

[Schaffert 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt.
An Introduction to Trellis/Owl.
In *Object-Oriented Programming Systems, Languages and Applications 1986 Conference Proceedings.* Association for Computing Machinery, September, 1986.

[Schmucker 86] Kurt J. Schmucker.
Macintosh Library: Object-Oriented Programming for the Macintosh.
Hayden Publishing Co., Hasbrouck Heights, NJ, 1986.

[Shaw 84] Mary Shaw.
Abstraction Techniques in Modern Programming Languages.
IEEE Software, 1(4):10-26, October, 1984.

[Snyder 86a] Alan Snyder.
Encapsulation and Inheritance in Object-Oriented Programming.
In *Object-Oriented Programming Systems, Languages and Applications 1986 Conference Proceedings.* Association for Computing Machinery, September, 1986.

[Snyder 86b] Alan Snyder.
CommonObjects: An Overview.
ACM Sigplan Notices, 21(10)October, 1986.
Special Issue on the Object-Oriented Programming Workshop, June 9-13, 1986.

[Snyder 86c] Alan Snyder.
Inheritance and the Development of Encapsulated Software Components.
In *Proceedings of the Nineteenth Annual International Conference on System Sciences.* January", 1986.

[Steele 84] Guy L. Steele, Jr.
Common Lisp The Language.
Digital Press, Burlington, MA, 1984.

[Stefik 86] Mark Stefik and Daniel G. Bobrow.
Object-Oriented Programming: Themes and Variations.
The AI Magazine, 6(4)1986.

[Stroustrup 86a] Bjarne Stroustrup.
The C++ Programming Language.
Addison Wesley, 1986.

[Stroustrup 86b] Bjarne Stroustrup.
An Overview of C++.
ACM Sigplan Notices, 21(10)October, 1986.
Special Issue on the Object-Oriented Programming Workshop, June 9-13, 1986.

[Weinreb 80] Daniel Weinreb and David Moon.
Flavors: Message Passing in the Lisp Machine.
Technical Report, Massachusetts Institute of Technology, November, 1980.