Encapsulation and Inheritance in Object-Oriented Programming Languages

Alan Snyder

Affiliation: Software Technology Laboratory
Hewlett-Packard Laboratories
P.O. Box 10490,
Palo Alto, CA , 94303-0971
(415) 857-8764

Abstract

Object-oriented programming is a practical and useful programming methodology that encourages modular design and software reuse. Most object-oriented programming languages support *data Abstraction* by preventing an object from being manipulated except via its defined external operations. In most languages, however, the introduction of *inheritance* severely compromises the benefits of this encapsulation. Furthermore, the use of inheritance itself is globally visible in most languages, so that changes to the inheritance hierarchy cannot be made safely. This paper examines the relationship between inheritance and encapsulation and develops requirements for full support of encapsulation with inheritance.

Introduction.

Object-oriented programming is a practical and useful programming methodology that encourages modular design and software reuse. One of its prime features is support for data Abstraction, the ability to define new types of objects whose behavior is defined Abstractly, without reference to implementation details such as the data structure used to represent the objects.

Most object-oriented languages support data Abstraction by preventing an object from being manipulated except via its defined external operations. Encapsulation has many advantages in terms of improving the understandability of programs and facilitating program modification. Unfortunately, in most object-oriented languages, the introduction of *inheritance* severely compromises encapsulation.

This paper examines the issue of encapsulation and its support in object-oriented languages. We begin by reviewing the concepts of encapsulation and data Abstraction, as realized by most object-oriented language. We then review the concept of inheritance and demonstrate how the inheritance models of popular object-oriented languages like Smalltalk [Goldberg83], Flavors [Moon86], and ObjectiveC [Cox84] fall short in their support of encapsulation. We examine the requirements for full support of encapsulation with inheritance.

Object-Oriented Programming

Object-oriented programming is a programming methodology based on the following key characteristics:

- Designers define new classes (or types) of objects.
- Objects have operations defined on them.
- Invocations operate on multiple types of objects (i.e., operations are generic).
- Class definitions share common components using inheritance.

In this paper, we use the following model and terminology: An object-oriented programming language allows the designer to define new *classes* of objects. Each object is an *instance* of one class. An object is represented by a collection of *instance variables,* as defined by the class. Each class defines a set of named *operations* that can be performed on the instances of that class. Operations are implemented by procedures that can access and assign to the instance variables of the target object. Inheritance can be used to define a class in terms of one or more other classes. If a class *c* (directly) inherits from a class *p,* we say that *p* is a *parent* of *c* and that *c* is a *child* of *p.* The terms *ancestor* and *descendant* are used in the obvious way. ** We avoid the traditional terms subclass and superclass because these terms are often used ambiguously to mean both direct and indirect inheritance.*

Encapsulation

Encapsulation is a technique for minimizing interdependencies among separately-written modules by defining strict external interfaces. The external interface of a module serves as a contract between the module and its clients, and thus between the designer of the module and other designers. If clients depend only on the external interface, the module can be reimplemented without affecting any clients, so long as the new implementation supports the same (or an upward compatible) external interface. Thus, the effects of compatible changes can be confined.

A module is *encapsulated* if clients are restricted by the definition of the programming language to access the module only via its defined external interface. Encapsulation thus assures designers that compatible changes can be made safely, which facilitates program evolution and maintenance. These benefits are especially important for large systems and long-lived data.

To maximize the advantages of encapsulation, one should minimize the exposure of implementation details in external interfaces. A programming language supports encapsulation to the degree that it allows *minimal* external interfaces to be defined and enforced. ** One can always improve the encapsulation support provided by a language by extending it with additional declarations (in the form of machine readable comments, say) and writing programa to verify that clients obey these declarations. However, the effective result of this approach is that a new language has been defined (in a way that happens to avoid changing the existing compiler); the original language has not become any less deficient.* This support can be characterized by the kinds of changes that can safely be made to the implementation of a module. For example, one characteristic of an object-oriented language is whether it permits a designer to define a class such that its instance variables can be renamed without affecting clients.

Data Abstraction

Data Abstraction is a useful form of modular programming. The behavior of an Abstract data object is fully defined by a set of Abstract operations defined on the object; the user of an object does not need to understand how these operations are implemented or how the object is represented.

Objects in most object-oriented programming languages are Abstract data objects. The external interface of an object is the set of operations defined upon it. Most object-oriented languages limit external access to an object to invoking the operations defined on the object, and thus support encapsulation. ** Most practical languages provide escapes from strict encapsulation to support debugging and the creation of programming environments. For example, in Smalltalk the operations* `instVarAt:` *and* `instVarAt:put:` *allow access (by numeric offset) to any named instance variable of any object [Goldberg83, p.247] Because these escapes are not normally used in ordinary programming, we ignore them in this analysis.* Changes to the representation of an object or the implementation of its operations can be made without affecting users of the object, so long as the externally- visible behavior of the operations is unchanged.

A class definition is a module with its own external interface. Minimally, this interface describes how instances of the class are created, including any creation parameters. In many languages, a class is itself an object, and its external interface consists of a set of operations, including operations to create instances.

To summarize, objects in most object-oriented programming languages (including class objects) are encapsulated modules whose external interface consists of a set of operations; changes to the implementation of an object that preserve the external interface do not affect code outside the class definition. ** In C++ [Stroustrup86], an operation performed on one object of a class can access the internals of other objects of the class; thus, the set of objects of a class is an encapsulated module rather than each individual object. We ignore this distinction in this paper as it does not affect our analysis.* If it were not for inheritance, the story would end here.

Inheritance

Inheritance complicates the situation by introducing a new category of client for a class. In addition to clients that simply instantiate objects of the class and perform operations on them, there are other clients (class definitions) that inherit from the class. To fully characterize an object-oriented language, we must consider what external interface is provided by a class to its children. This external interface is just as important as the external interface provided to users of the objects, as it serves as a contract between the class and its children, and thus limits the degree to which the designer can safely make changes to the class.

Frequently, a designer will want to define different external interfaces for these two categories of clients. Most object-oriented languages respond to this need by providing a much less restricted external interface to children of a class. By doing so, the advantages of encapsulation that one associates with object-oriented languages are severely weakened, as the designer of a class has less freedom to make compatible changes. This issue would be less important if the use of inheritance were confined to individual designers or small groups of designers who design families of related classes. However, systems designers have found it useful to provide classes designed to be inherited by large numbers of classes defined by independent applications designers (the class *window* in the Lisp Machine window system [Weinreb81] is a good example); such designers need the protection of a well-defined external interface to permit implementation flexibility.

We will begin our examination of inheritance with the issue of access to inherited instance variables.

Inheriting Instance Variables

In most object-oriented languages, the code of a class may directly access all the instance variables of its objects, even those instance variables that were defined by an ancestor class. Thus, the designer of a class is allowed full access to the representation defined by an ancestor class.

This property does not change the external interface of individual objects, as it is still the case that the instance variables of an object are accessible only to operations defined on that object. However, it does change the external interface of the class (as seen by its descendants), which now (implicitly) includes the instance variables.

Permitting access to instance variables defined by ancestor classes compromises the encapsulation characteristics stated above: Because the instance variables are accessible to clients of the class, they are (implicitly) part of the contract between the designer of the class and the designers of descendant classes. Thus, the freedom of the designer to change the implementation of a class is reduced. The designer can no longer safely rename, remove, or reinterpret an instance variable without the risk of adversely affecting descendant classes that depend on that instance variable.

In summary, permitting direct access to inherited instance variables weakens one of the major benefits of object-oriented programming, the freedom of the designer to change the representation of a class without impacting its clients.

Accessing Inherited Variables Safely

To preserve the full benefits of encapsulation, the external interfaces of a class definition should not include instance variables. Instance variables are protected from direct access by users of an object by requiring the use of operations to access instance variables. The same technique can be used to prevent direct access by descendant classes.

Additional language support is required to permit instance variable access operations to be used effectively by descendant classes. Ordinary operation invocation on `self` ** In Smalltalk and many of its derivatives,* `self` *is used within an operation to refer to the object that the operation is, being performed on. Names used in other languages for the same purpose include* `me` *and* `this`. is inadequate, as it may invoke the wrong operation (if the operation is redefined by the class or one of its descendants). Instead, a way is needed to directly invoke (on is inadequate, as it may invoke the wrong operation (if the operation is redefined by the class or one of its descendants). Instead, a way is needed to directly invoke (on **`self`**) an operation as defined by a parent class. Smalltalk provides a mechanism in the context of single inheritance: the pseudo-variable **`super`**. Performing an operation on **`super`** is like performing an operation on **`self`**, except that the search for the operation to invoke starts with the parent of the class in which the invocation appears, instead of with the class of **`self`**. Equivalent features using compound names (parent and operation) to specify the desired operation are provided by CommonObjects [Snyder85a], Trellis/Owl [Schaffert86], extended Smalltalk ** We use the term "extended Smalltalk' to refer to the multiple inheritance extension to Smalltalk defined by Borning and Ingalls.*, extended Smalltalk [Borning82], and C++ [Stroustrup86]. Because the search for the proper operation starts at a statically known class, invoking an operation in this manner can be more efficient than normal operation invocation; in some implementations, no run-time lookup is required.

There are several possible objections to using operations to access inherited instance variables. Most of these objections also apply to the ordinary case of using operations to access the instance variables of an object, and they can be resolved the same way: Syntactic abbreviations allow the inheriting class to use ordinary variable reference syntax to invoke these operations [Snyder85a]. ** This option is a special case of a general construct which we call pseudo variables [Snyder85b]. A pseudo variable looks like an ordinary lexical variable, but the effect of referencing or assigning to the variable is to execute arbitrary (possibly user-specified) code.* Inline substitution avoids the overhead of invoking a procedure (at the cost of requiring recompilation of the client if an ancestor class is incompatibly changed).

The most serious objection to this solution is that it requires the cooperation of the designer of the ancestor class that defines the instance variable (as well as the designers of all intervening ancestors), since no access to an inherited instance variable is possible unless appropriate operations have been provided. We claim this is as it should be. If you (as designer of a class) need access to an inherited instance variable and the appropriate operations are not defined, the correct thing to do is to negotiate with the designer(s) of the ancestor class(es) to provide those operations. ** A software development environment might allow you to circumvent this restriction, say by temporarily defining those operations yourself. However, if you seriously intend to leverage off someone else's code, such negotiation is essential.*

One problem with this scenario is that the instance variable operations defined on a class for the benefit of its descendants may not necessarily be appropriate for users of instances of the class, yet they are publically available. A convenient solution (provided by Trellis/Owl) is to declare that some operations are available only for direct invocation (on **`self`**) by descendant classes, but are not part of the external interface of objects of the class. This notion is a generalization of the "private operations" provided by various languages, including Trellis/Owl and C++. It gives the designer fine-grained control over the external interfaces provided to the two categories of clients of a class.

If instance variables are not part of the external interface of a parent class, then it is not proper to merge inherited instance variables with instance variables defined locally in a class (as is done by Flavors). Clearly, if a local instance variable and an inherited instance variable with the same name wind up as a single instance variable, then changing the name of the instance variable in the parent is likely to change the behavior of the child class. It is also inappropriate to signal an error if a class defines an instance variable with the same name as an inherited instance variable (as is done by

Smalltalk), as changing the name of an instance variable could make a descendant class illegal. Similar objections apply to merging instance variables defined by multiple parents (as is done by Flavors), or signalling an error if multiple parents define instance variables with the same name (as is done by extended Smalltalk).

The Visibility of Inheritance

A deeper issue raised by inheritance is whether or not the use of inheritance itself should be part of the external interface (of the class or the objects). In other words, should clients of a class (necessarily) be able to tell whether or not a class is defined using inheritance?

If the use of inheritance is part of the external interface, then changes to a class definition's use of inheritance may affect client code. For example, consider a class that is defined using inheritance. Suppose the designer of that class decides that the same behavior can be implemented more efficiently by writing a completely new implementation, without using the previously inherited class. If the previous use of inheritance was visible to clients, then this reimplementation may require changes to the clients. The ability to safely make changes to the inheritance hierarchy is essential to support the evolution of large systems and long-lived data.

This issue raises the fundamental question of the purpose of inheritance. One can view inheritance as a private decision of the designer to "reuse" code because it is useful to do so; it should be possible to easily change such a decision. Alternatively, one can view inheritance as making a public declaration that objects of the child class obey the semantics of the parent class, so that the child class is merely *specializing* or *refining* the parent class. This question is addressed in the context of knowledge representation in [Brachman85].

Our position is that being able to use inheritance without making a public commitment to it in the external interface of a class is valuable, and we will analyze how existing object-oriented languages support this option. We will begin by considering only the single inheritance case, i.e., where the inheriting class (the *child*) directly inherits from a single class (the *parent*). Multiple inheritance introduces additional problems and will be discussed below.

Excluding Operations

Most object-oriented languages promote inheritance as a technique for specialization and do not permit a class to "exclude" an inherited operation from its own external interface. However, if inheritance is viewed as an implementation technique, then excluding operations is both reasonable and useful.

For example, consider the Abstractions *stack* and *deque,* where a *stack* is a queue that permits elements to be added or removed from one end, and a *deque* is a queue that permits elements to be added or removed from either end. The external interface of a *deque* is a superset of the external interface of a *stack:* a *deque* has two additional operations for adding and removing elements from the "back end".

The simplest way to implement these two Abstractions (at least for prototyping purposes) is to define the class *stack* to inherit from the class *deque,* and exclude the extra operations. *Stack* inherits the implementation of *deque,* but is not a specialization of *deque,* as it does not provide all the *deque* operations.

The ability for a class to exclude operations defined by its parents is provided in CommonObjects. Of courage, this effect can be achieved in any object-oriented language simply by redefining the operation in the child class to signal an error when invoked. Doing so, however, fails to take full advantage of static type checking (it languages like Trellis/Owl and C++) and can lead to unnecessary name conflicts involving the "excluded" operations.

Even if a child excludes an inherited operation from its external interface, it is useful to be able to invoke the operation (on **`self`**) within the child. Ordinary operation invocation cannot be used, because the operation is not defined on instances of the child. However, as described above in the section on accessing inherited instance variables, several languages (including CommonObjects) provide the required ability to directly invoke an operation defined by a parent class.

Subtyping

One way in which inheritance can appear in the external interface of a class is *subtyping,* the rules by which objects of one type (class) are determined to be acceptable in contexts expecting another type (class). In statically-typed languages like Trellis/Owl, Simula [Dahl66], and C++, subtyping rules are of critical importance because they determine the legality of programs. In dynamically-typed languages like Common Lisp [Steele84], subtyping rules affect the results of type predicates.

Many object-oriented languages relate subtyping and inheritance. For example, in Trellis/Owl, Simula, and C++ a class *x* is a subtype of a class *y* if and only if *x* is a descendant of *y.* If the designer should reimplement class *x* so that it inherits from class *z* instead of *y,* client programs that assume *x* is a subtype of *y* would no longer be legal, even if the *intended*external interface of*x* (the operations) is unchanged. Thus, the use of inheritance is exposed via the subtyping rules ** It should be noted that although equating subtyping of classes with inheritance entails a loss of flexibility, it has significant implementation disadvantages in statically-typed languages like those mentioned, where the cost of performing an operation involves at most an extra level of indirection.*

To avoid this problem, subtyping should not be tied to inheritance. Instead, subtyping should be based on the behavior of objects. If instances of class *x* meet the external interface of class *y,* then *x* should be a subtype of *y.* The example above demonstrates that the implementation hierarchy need not be the same as the type hierarchy (as defined by object behavior) [Canning85]. In that example, *stack* inherits from *deque* but is not a subtype of *deque,* and *deque* is a subtype of *stack* but does not inherit from *stack.*

Behavioral subtyping cannot be deduced without formal semantic specifications of behavior. Lacking such specifications, one can deduce subtyping based solely on syntactic external interfaces (i.e., the names of the operations) [Cardelli84]. Alternatively (or in addition), one can allow the designer of a class to specify which other classes it is a subtype of. The default may be that a class is a subtype of each of its parents. However, as demonstrated by the example, the designer must be able to specify that the class is not a subtype of a parent or that the class is a subtype of an unrelated class (not its parent). The first case arises when the behavior of the objects is incompatible with the interface of parent object. The second case arises when the class is supporting the external interface of another class without sharing its implementation.

CommonObjects, an object-oriented extension to Common Lisp, is an example of a language that allows the designer to specify the type hierarchy independently of the inheritance hierarchy. In Common Lisp, type checking is defined in terms of a predicate **`typep`** that takes two arguments, an object and a type specification, and returns true if and only if the object is a member of the specified type. CommonObjects classes are integrated into the Common Lisp type system such that if the object given to **`typep`** is an instance of a class and the type specification is a class name, then the **`:typep`** operation is performed on the object (with the type specification as the argument) to determine the result of **`typep`**. The designer of a class can write an arbitrary predicate for this operation, although a default is provided. ** Smalltalk defines an operation `isKindOf` on all objects that tests whether the object is an instance of a class or one of its descendants; however, it is not clear that this operation can usefully be redefined.*

The following operations would be used in the above example:

```
(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)
      ))
```

This primitive mechanism is sufficient, but is neither convenient nor reliable. A better solution would explicitly represent subtyping relationships and ensure transitivity.

Attribute Visibility

If the use of inheritance is not part of the external interface of a class, then clients of the class may not directly refer to ancestors of the class. Specifically, a class may refer to its parents, but not its more distant ancestors.

As mentioned above, it is useful for a class to be able to invoke an operation defined by a parent. In most languages that support this feature, the desired operation is specified by a compound name consisting of the name of the parent class and the name of the operation. This solution is sufficient: To access an operation of a more distant ancestor without violating encapsulation, that operation must be passed down via all intervening ancestors including at least one parent. Trellis/Owl and extended Smalltalk allow a class to directly name an operation of a non-immediate ancestor. As a result, in these languages, the names of ancestor classes are unavoidably part of the external interface of a class.

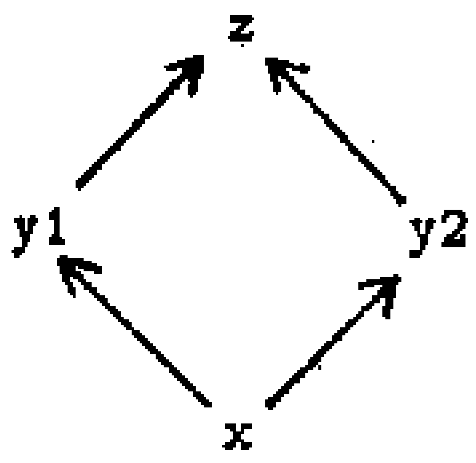

Figure 1: Example of multiple Inheritance.

Multiple Inheritance

Multiple inheritance means that a class can have more than one parent. One can view a class (call it the *root class*) as forming the root of a directed (acyclic) graph, which we will call an *inheritance graph,* where each class is a node and there is an arc from each class to each of its parents. Figure 1 shows an inheritance graph with multiple inheritance. In this example, class *x* is the root class. Class *x* inherits from classes *y1* and *y2,* and classes *y1* and *y2* both inherit from class *z.*

There are two strategies in common use for dealing with multiple inheritance. The first strategy attempts to deal with the inheritance graph directly. The second strategy first flattens the graph into a linear chain, and then deals with that chain using the rules for single inheritance.

Graph-Oriented Solutions

Trellis/Owl and extended Smalltalk are examples of object-oriented languages whose semantics model the inheritance graph directly. In these languages, operations are inherited along the inheritance graph until redefined in a class. If a class inherits operations with the same name from more than one parent, the conflict must be resolved in some way. One way is to redefine the operation in the child class; the new definition can invoke the operations defined by the parent classes. To invoke all definitions of an operation in the inheritance graph (e.g., all **`display`** operations), each class could define an operation that invokes the operation on each of its parents and then performs any local computation, resulting in a depth-first traversal of the inheritance graph. Extended Smalltalk provides a convenient syntax for invoking an operation on each parent of a class.

In these languages, the interesting issues arise when the graph is not a tree, i.e., when a single class is reachable from the root class by multiple paths (as in Figure 1). Trellis/Owl and extended Smalltalk adopt similar solutions for resolving the conflict when a class attempts to inherit an operation from more than one parent: It is an error ** Trellis/Owl signals an error at compilation time; extended Smalltalk creates an operation for the class that signals an error when invoked.* if a class inherits operations with the same name from two or more parents, but *only* if the operations are actually different. In other words, if the *same* operation (from the same class) is inherited by a class via different paths through the inheritance graph, it is not an error.

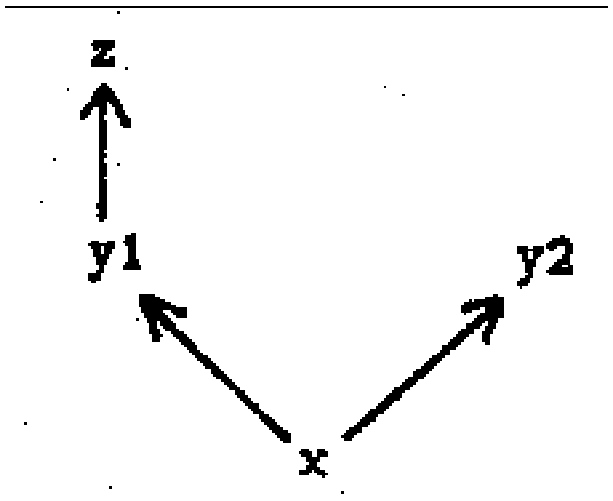

Figure 2: Altering the inheritance structure.

This same-operation exception is motivated by convenience, as operation conflicts are inevitable when there is merging in the inheritance graph. Although this exception may at first glance seem innocuous, it forces the use of inheritance to be part of the external interface.

Consider further the example shown in Figure 1. If an operation *o* is defined (only) by the class *z,* it will be inherited by *y1* and *y2* and then by *x* (with no error). Now suppose the class *y2* is reimplemented so that it no longer inherits from *z* (see Figure 2) but still supports the same behavior. Class *x* will now be in error, because it is inheriting two *different* operations named *o,* one from class *z* via class *y1* and one from class *y2.* (The operations are different yet have equivalent behavior on objects of their respective classes.) Thus, the change to class *y2* to stop inheriting from *z* was visible to class *x,* a client of class *y2.* The external interface of *y2* was therefore changed, even though the external interface of its instances remains unchanged (the instances have the same externally-visible behavior).

To prevent the use of inheritance from necessarily becoming part of the external interface of a class, one must abandon the convenient same-operation exception to the rule regarding operation conflict. An alternative conflict resolution technique that avoids this problem is to select the operation from the "first" parent that defines one (based on the textual order in which the parents are named); this alternative is undesirable because it fails to warn the designer about unintentional conflicts.

Another attribute of the graph-oriented solution is that only one set of instance variables is defined for any ancestor class, regardless of the number of paths by which the class can be reached in the inheritance graph. For example, in Figure 1, an instance of *x* contains one of each instance variable defined by *z,* not two.

While this result is usually desirable, it introduces potential problems. For example, using a depth-first traversal (as described above) on an inheritance graph with merging will result in some operations being invoked more than once on the same set of instance variables. As a result, a designer cannot change the use of inheritance within a class without the danger of breaking some descendant class.

Consider the structure shown in Figure 2. Suppose the operation *o* is defined by classes *z, y2,*and *x,* where the definition of *o* in class *x* invokes operation *o* on both parents. Now suppose the designer of class *y2* decides to reimplement *y2* to inherit from *z* in a manner that preserves the external behavior of objects of class *y2;* assume that class *y2* will either inherit

o from *z* or will revise its definition of *o* to invoke *o* as defined by *z.* The operation *o* on class *x* will now have the effect of invoking the operation *o* on class *z* twice, on the same set of instance variables. If *o* on *z* has side-effects, the result might not be acceptable. Thus, in this example, a change to the use of inheritance by a class (*y2*) has broken one of its clients (*x*), even though its operations have the same external behavior; the use of inheritance is therefore (implicitly) part of the external interface of the class *y2.*

Linear Solutions

The second strategy for dealing with multiple inheritance is to first flatten the inheritance graph to a linear chain, without duplicates, and then treat the result as single inheritance. This strategy is used by Flavors (as recently revised) [Moon86] and CommonLoops [Bobrow86]. These languages use similar algorithms for creating a total ordering that preserves the ordering along each path through the inheritance graph (a class never appears after one of its ancestors). Flavors attempts as well to preserve the relative ordering of the parents of a class (the first parent of a class never appears after the second, etc.); an error is signalled if no total ordering exists that satisfies all such constraints.

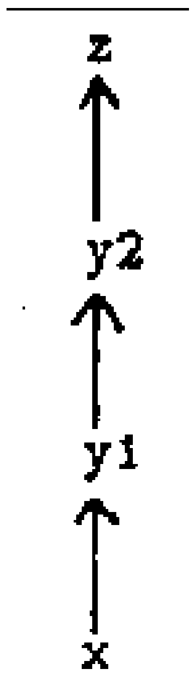

Figure 3: Linearized multiple inheritance.

While the relative ordering of a class and its ancestors is preserved in the total ordering, unrelated classes can be inserted between a class and its parent. For example, the inheritance graph shown in Figure 1 would be transformed into the chain shown in Figure 3. Note that the class *y2* now appears between class *y1* and its parent class *z*. Thus, although these languages treat the computed inheritance chain as single inheritance, the computed inheritance chain has the unusual property that the "effective parent" of a class *c* may be a class of which the designer of class c has absolutely no knowledge.

One problem with this solution is that in the case of operation conflicts (two or more parents defining the same operation), one operation will be selected even if there is no clearly "best" choice. Consider the above example: If operation *o* is defined only by classes *y1* and *y2*, then class *x* will inherit the operation defined by class *y1*. This selection assumes that the ordering of parents in a class definition is significant. Although such conflicts are likely to be unintentional, no warning is given.

Another problem involves the ability of a class to reliably communicate with its "real" parents. CommonLoops allows a class to invoke its "effective parent" (its parent in the computed inheritance chain) using a notation similar to **`super`** in Smalltalk. Flavors provides a declarative mechanism called method combination that is essentially equivalent. These mechanisms avoid the problem of unintentional multiple invocations of operations that can occur using graph-oriented solutions. However, these mechanisms make it difficult for a class to reliably set up private communication with a parent, given the interleaving of classes that can occur when the inheritance chain is computed.

In the above example, suppose that class *y1* wants to communicate privately with its parent *z;* specifically, it wants to invoke (public) operations defined by *z* without being affected by redefinitions of those operations by other classes. This kind of private communication is easily arranged in the graph-oriented languages (Trellis/Owl and extended Smalltalk) by directly naming the operations of *z*. However, using **`super`** or its equivalent, class *y1* can directly invoke only operations of its "effective parent", which in this case is *y2;* if *y2* happened to redefine the operation in question, the "wrong" operation (from *y1*'s point of view) would be invoked. This example demonstrates that to correctly use *y1* as a parent, one must understand details of its internal design regarding its use of its parent *z*.

Unlike the graph-oriented languages, neither CommonLoops nor Flavors allows a class to designate an operation by the name of the defining parent. Adding this feature would allow one to simulate the semantics of the graph-oriented languages. However, its use would be error-prone, as mixed use of the two different styles (linear or graph) would probably produce undesirable results.

Tree Solutions

We are exploring in CommonObjects [Snyder85a] a third strategy for handling multiple inheritance that avoids the problems of the graph-oriented and linear solutions. Like the graph-oriented solutions of Trellis/Owl and extended Smalltalk, the semantics of CommonObjects models the inheritance graph. However, there are two key differences: (1) An attempt to inherit an operation from more than one parent is always an error, regardless of the source(s) of the operation. This change plugs the "leak" described above. (2) The inheritance graph is converted into a tree by duplicating nodes. In other words, each parent of each class defines a completely separate set of inherited instance variables; if a class is reachable by multiple paths through the inheritance graph, a separate set of instance variables will be created for each such path. For example, in Figure 1, an instance of *x* contains two of each instance variable defined by *z*. This change avoids situations where an operation is accidentally invoked multiple times on the same set of instance variables or where two classes conflict in their use of an inherited class.

The tree solution used by CommonObjects avoids exposing the use of inheritance, but radically changes the semantics of inheritance graphs with shared nodes. Applications in other languages that use shared ancestors would almost certainly have to be redesigned in CommonObjects (and vice versa). Shared ancestors typically arise when inheriting multiple "base" classes, where a "base" class is one that defines a "complete" set of operations and is generally designed to be instantiated. An alternative is to inherit a single "base" class and one or more "mixin" classes, where a "mixin" class [Weinreb81] defines a set of operations related to one particular feature and is designed only to be inherited from ("mixed into" a class). Using this strategy, designers will be encouraged to create more, and more general, "mixin" classes. Further experimentation is needed to determine the practicality of this alternative programming style.

Inheritance in ThingLab [Borning81] uses a modified tree solution: it differs from CommonObjects in two major ways: in ThingLab, each parent (called a *part*) is an object in its own right, and the

ThingLab *merge* capability can explicitly induce merging in the inheritance hierarchy.

Summary

We have identified two areas where most object-oriented languages are deficient in their support for encapsulation. One area is the encapsulation of instance variables, a significant feature of most object-oriented languages. Many popular object-oriented languages (e.g., Smalltalk, Flavors, and ObjectiveC) allow free access to inherited instance variables by descendant classes, thus denying the designer the freedom to compatibly change the representation of a class without affecting clients. It is encouraging that several newer languages (Commonobjects, Trellis/Owl, and C++) correct this deficiency by restricting access to inherited instance variables. Where access to inherited instance variables is needed, it should be provided in the form of operations. Language support is needed to permit classes to directly invoke parent operations (on **`self`**) and to permit such operations to be made available in this manner but not via ordinary operation invocation. Furthermore, the implicit by-name merging of instance variables defined in separate classes is inconsistent with the goals of encapsulation.

The other area is the visibility of inheritance itself. Inheritance is a useful mechanism for reusing code; the decision by the designer to use inheritance for this purpose should be private and easily changed. To permit the use of inheritance to be private, a class must be able to exclude operations defined on its parents. A class must not refer to ancestors other than its parents. A language cannot define a subtyping relation based solely on inheritance. Conflicts in operation inheritance between multiple parents must be reported regardless of the source(s) of the operation(s).

The two models of multiple inheritance in common use both require knowledge about the use of inheritance by a class to understand how to correctly inherit that class. CommonObjects provides an alternative form of multiple inheritance that supports encapsulated class definitions whose external interface is fully defined by a set of operations further experimentation is needed to evaluate and refine this programming style.

The challenge for language designers is to provide the means by which the designer of a class can express an interface to inheriting clients that reveals the minimum information needed to use the class correctly.

References

[Bobrow86] Danny Bobrow, et al. *CommonLoops: Merging Common Lisp and Object- Oriented Programming*. Proc. ACM Conference on Object-Oriented Systems, Languages, and Applications. Portland, Oregon, Sept. 1986.

[Borning81] Alan Borning. The Programming Language Aspects of ThingLab, a Constraint- Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems* 3:4 (Oct. 1981), 353-387.

[Borning82] Alan Borning and Daniel Ingalls. Multiple Inheritance in Smalltalk-80. Proc. AAAI 1982, 234-237.

[Brachman85] Ronald J. Brachman. "I Lied about The Trees" — Or Defaults and Definition in Knowledge

Representation. *AI Magazine* 6:3 (Fall 1985), 80-93.

[Canning85] Peter Canning. Personal communication.

[Cardelli84] Luca Cardelli. The Semantics of Multiple Inheritance. *Proceedings of the Conference on the Semantics of Datatypes*. Springer-Verlag Lecture Notes in Computer Science, June 1984, 51-66.

[Cox84] Brad Cox. Message/Object Programming: An Evolutionary Change in Programming Technology. *IEEE Software* 1:1 (Jan. 1984), 50-61.

[Dahl66] O.-J. Dahl and K. Nygaard. Simula — An Algol-based Simulation Language. *Comm. ACM* 9:9 (Sept. 1966), 671-678.

[Goldberg83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[Moon86] David A. Moon. *Object-Oriented Programming with Flavors*. Proc. ACM Conference on Object-Oriented Systems, Languages, and Applications. Portland, Oregon, Sept. 1986.

[Schaffert86] Craig Schaffert, et. al. *An Introduction to Trellis/Owl*. Proc. ACM Conference on Object-Oriented Systems, Languages, and Applications. Portland, Oregon, Sept. 1986.

[Snyder85a] Alan Snyder. *Object-Oriented Programming for Common Lisp*. Report ATC-85-1, Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, California, 1985.

[Snyder85b] Alan Snyder, Michael Creech, and James Kempf. *A Common Lisp Objects Implementation Kernel*. Report STL-85-08, Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, California, 1985.

[Steele84] Guy L. Steele, Jr. *Common Lisp — The Language*. Digital Press, 1984.

[Stroustrup86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1986.

[Weinreb81] Daniel Weinreb and David Moon. *Lisp Machine Manual*. Symbolics, Inc., 1981.