Multiple Inheritance in Smalltalk-80

Alan H. Borning
Computer Science Department, FR-35
University of Washington
Seattle, WA 98195

Daniel H. H. Ingalls
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

Abstract

Smalltalk classes may be arranged in hierarchies, so that a class can inherit the properties of another class. In the standard Smalltalk language, a class may inherit from only one other class. In this paper we describe an implementation of multiple inheritance in Smalltalk.

1. Introduction

Smalltalk is a powerful interactive language based on the idea of objects that communicate by sending and receiving messages [Ingalls 78, LRG 81, Goldberg 82]. Every Smalltalk object is an instance of some class. Classes are organized hierarchically, so that a new class is normally defined as a subclass of an existing class. The subclass inherits the instance storage requirements and message protocol of its superclass. It may add new information of its own, and may override inherited responses to messages.

In standard Smalltalk, a class can be a subclass of only a single superclass. On occasion, this restriction is undesirable and leads to unnatural coding styles. For example, the Smalltalk system includes a class *Transcript* that displays and records notification messages and the like. It is declared to be a subclass of *Window*, but also has the message protocol of a *WriteStream* to which one can append characters. Since it cannot be a subclass of both *Window* and *WriteStream*, the necessary methods for stream behavior must all be duplicated in *Transcript*. Such duplication is unmodular. If some method for streams is added or modified, the class *Transcript* does not automatically feel this change (as it would if it were a subclass of *WriteStream*).

The natural solution is to allow classes to be subclasses of more than one superclass. In this paper we describe an implementation of multiple superclasses, which is now available in the Smalltalk-80 system used within Xerox PARC.

2. Semantics of Multiple Superclasses

A class may have any number of superclasses; however, an instance is always an instance of precisely one class.

2.1. Message Handling

When an instance receives a message, it first checks the method dictionary of its own class for a method for receiving that message. If none is found, it searches the method dictionaries of its immediate superclasses, then *their* superclasses, and so on. If a single method is found, then it is run. If no method or more than one method is found, an error message is issued. The overriding of inherited methods is still allowed: it is an error only if a class with no method of its own inherits different methods from two or more of its superclasses. Further, it is not an error if the same method is inherited via several paths. (This is a simplified explanation; Section 4 describes our actual implementation.)

2.2. Access to Overridden Inherited Methods

In single-superclass Smalltalk, the programmer can access an inherited overridden method using the reserved word *super*. For example, in code defined in a given class *C*, the inherited method for *copy* may be invoked using the expression *super copy*, even if *C* itself has a method for *copy*.

This mechanism may be insufficient in the presence of multiple superclasses -- for example, if *C* inherits two different methods for *copy*, the user needs a way to indicate which is wanted. To allow for this, we extend the syntax of Smalltalk by adding *compound selectors* consisting of a class name, followed by a period, followed by the actual selector, e.g. *Object.copy*. When one of these compound selectors is used in a message, the lookup for the method starts with the class named in the compound selector.

When there is no ambiguity, it is still convenient to be able to say "use the method inherited from my superclass" without naming that superclass. In analogy with the above form of compound selector, this can be accomplished by writing e.g. *self super.copy*.

Finally, there are times when one would like to invoke *all* the inherited methods for a given selector, rather than just one of them: the principal example of this is for the *initialize* method. To accomplish this, the programmer would write *self all.initialize*. It would be straightforward to add other sorts of method combination schemes using this basic mechanism.

3. Examples of Using Multiple Inheritance

In this section we present a number of examples that illustrate the usefulness of multiple inheritance.

3.1. Simula-style Linked Lists

Simula, which has a single-superclass inheritance hierarchy, defines a list-processing package that supports doubly-linked lists [Birtwistle 73]. The class *Link* specifies that each of its instances contain a reference to a successor and to a predecessor object. Subclasses of *Link* may then be defined that inherit this ability to be included in linked lists. An analogous class may be easily defined in Smalltalk. (An advantage of implementing linked lists in this way, rather than having a separate link object that simply points to an object X in the list, is that X can know about the list in which it resides.)

However, there is a problem with the class *Link* in both Simula and single-superclass Smalltalk. Given an arbitrary existing class C, unless C already has *Link* in its superclass hierarchy, a programmer cannot use C in defining a new subclass that also has the properties of a *Link*.

Multiple superclasses provide a natural solution. For example, if the programmer wants to make objects that are like windows and can also be included in doubly-linked lists, he or she can simply define a new class *QueueableWindow* that is a subclass of both *Window* and *Link*. The new class will inherit the instance state requirements and message protocol of both *Window* and *Link*, yielding the desired behavior.

3.2. Other Examples

As mentioned in the introduction, another situation in which multiple inheritance is useful is in defining the class *Transcript* as a subclass of *Window* and of *WriteStream*.

To take another example from the standard Smalltalk-80 system, a number of kinds of streams are defined, including *ReadStream*, *WriteStream*, and *ReadWriteStream*. *ReadWriteStream* is rather arbitrarily declared to be a subclass of *WriteStream*, with the extra methods needed for *ReadStream* behavior copied by the programmer. Using our new system, *ReadWriteStream* is naturally defined as a subclass of both *ReadStream* and *WriteStream*.

3.3. Pool Variables

This last example is of a somewhat different nature. In addition to instance variables, the Smalltalk-80 language allows the programmer to define *class variables* that are shared by all instances of a given class and its subclasses. However, on occasion, the programmer wants variables that are to be shared by a number of non-hierarchical classes, but which aren't properly declared to be global variables. A mechanism for handling this exists already: one may declare a dictionary of *pool variables* that may be shared among several classes. (An example of this is the *FilePool* of constants and variables that are shared by all the classes used in file I/O.)

Multiple superclasses provide a more elegant solution. Rather than using pool variables, one can for example define a class *FileObject* that has class variables corresponding to all the variables that used to be in *FilePool*. Each of the file classes can now be made a subclass of *FileObject* as well as of its old superclass, so that it has access to these shared variables. In this way, the pool mechanism becomes unnecessary and could be eliminated from the language.

4. Implementation

4.1. Finding the Right Method to Receive a Message

Our implementation of multiple inheritance is a compromise between the extremes of strict runtime method lookup and copying down inherited methods from all superclasses.

In the standard Smalltalk-80 system, methods inherited from superclasses are looked up dynamically. This has the advantage that the system is not cluttered with copied methods, and that there are no copies to update when a method is edited. An alternative would be to copy the inherited methods down into each subclass. This would make finding the methods easier at runtime, at the expense of greater code size and updating complexity.

In our implementation of multiple inheritance, the standard dynamic lookup scheme is used for methods on the chain consisting of the first superclass of each class. If a class C has more than one superclass, at the time C is created it checks each message to which it can respond. If the appropriate method would be found by the dynamic lookup, nothing is done. However, if the appropriate method is in some other superclass, then the code for that method is recompiled in C's method dictionary, so that it will be found at run time.

Finally, if there are conflicting inherited methods for a given selector, an error method is compiled in C for that selector and the user is notified. These error methods are put into a special category, making it easy for the user to browse to them and to resolve the conflicts as necessary.

4.2. Implementation of Compound Selectors

As described in Section 2.2, the programmer can access inherited methods using constructs such as *self Object.copy*, *self super.copy*, and *self all.initialize*. To implement these extensions, we changed the Smalltalk parser to treat compound selectors as single symbols, so that the code that is compiled in *C* for e.g. *self B.copy* actually sends the selector *B.copy*. The first time this is executed, no method for *B.copy* will be found. When this occurs, the interpreter invokes *Object messageNotUnderstood*. The usual behavior at this point is to bring up an error window. However, we modified *Object messageNotUnderstood* to first check for compound selectors. If one is found, then the system attempts to compile an appropriate method for that compound selector by first verifying that *B* is a superclass of *C*, and then looking for a *copy* method in *B* or its superclasses. If one is found, that method is recompiled in *C* under the selector *B.copy*. The system then resends the message, whereupon it will find the newly inserted method. The next time *B.copy* is sent, this method will be found, making the operation efficient. Selectors such as *super.copy* and *all.initialize* are handled by the same mechanism.

4.3. Instance State

A subclass inherits all the instance field requirements of its superclasses, and can specify additional fields of its own. There is only one copy of fields inherited from a superclass via two inheritance paths. In our current implementation, it is an error if there are different inherited instance fields with the same name. (One of our previous experimental implementations [Borning 80] included a mechanism similar to the compound selector construct that allowed the programmer to disambiguate conflicting field names. We may re-introduce this mechanism if the present restriction proves too burdensome.)

To access or store into instance fields, the bytecodes produced by the Smalltalk compiler include instructions such as "load instance field 1". It is of course essential that code inherited from superclasses use the correct field positions for the subclass. Our scheme takes care of this in the following manner. The instance fields are arranged so that the fields inherited from the superclasses on the dynamic lookup chain have the same positions as they do in the superclasses. (This is the same situation as in single-superclass Smalltalk.) In general, fields inherited from other superclasses won't be in the same positions, but when the code for methods from these other superclasses is recompiled into the new subclass, the field positions are adjusted appropriately. As an optimization, before recompiling a method from a superclass the system checks if the offsets of all the fields it references are the same in the subclass. If this is the case, then the system simply copies a pointer to the original method, rather than recompiling it.

4.4. Dynamic Updating

In the Smalltalk environment, the user can add, delete, and edit methods incrementally, and then immediately make use of the changed code. In our multiple-inheritance implementation, some updating may be necessary when such changes are made. If a method is edited which has been recompiled or copied into some subclasses, then the newly edited method is recompiled or copied into subclasses as necessary. Similarly, if a method is added or deleted, it may affect which inherited method should be used, and may require changes in the copied inherited methods. Again, the system takes care of this updating automatically.

If methods with compound selectors (e.g. *super.printOn:*) have been automatically compiled into some subclasses, then these methods may be invalid as well. Each such method that may no longer be valid is simply deleted; as described above, it will be recompiled automatically the first time a message is sent that invokes it.

4.5. A Note on the Implementation Process

The changes required to add multiple inheritance to Smalltalk-80 are only a few pages of Smalltalk code. For example, changing the Smalltalk syntax to allow compound selectors of the form *Point.copy* or *Point.+* required a change to only one method. Moreover, no changes to the Smalltalk-80 virtual machine were required. There are few other programming environments in which such a fundamental extension could be made so easily.

5. Relation to other Work

A number of other systems have used multiple inheritance. Among the systems implemented in Smalltalk, the constraint laboratory ThingLab [Borning 81] and the PIE knowledge representation language [Goldstein and Bobrow 80] both supported multiple inheritance. The authors have also implemented some experimental predecessors of the present system [Borning 80].

Some extensions to Lisp allow the use of similar object-oriented programming techniques. The "flavors system" in MIT Lisp Machine Lisp [Cannon 80] allows an object to be defined using several flavors (analogous to multiple superclasses); this system also contains an extensive repertoire of method combination techniques for combining inherited information. Another object-oriented Lisp extension with multiple inheritance is the LOOPS system [Bobrow and Stefik 82], implemented in Interlisp.

The Traits system [Curry 82], imbedded in the Mesa system, is yet another multiple inheritance implementation. It has received extensive use in the coding of the Xerox Star office information system.

References

[Birtwistle 73] Birtwistle, G.M., Dahl, O.-J., Myhrhaug, B., and Nygaard, K.
SIMULA Begin.
Auerbach Press, 1973.

[Bobrow and Stefik 82]
Bobrow, D.G., and Stefik, M.J.
LOOPS: An Object Oriented Programming System for Interlisp.
1982.

[Borning 80] Borning, A.H.
Multiple Inheritance in Smalltalk.
1980.
Unpublished report, Learning Research Group, Xerox PARC.

[Borning 81] Borning, A.H.
The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory.
ACM Transactions on Programming Languages and Systems 3(4):353-387, October, 1981.

[Cannon 80] Cannon, H.I.
Flavors.
Technical Report, MIT Artificial Intelligence Lab, 1980.

[Curry 82] Curry, G., Baer, L., Lipkie, D., and Lee, B.
Traits: An Approach to Multiple Inheritance Subclassing.
In *ACM-SIGOA Conference on Office Automation Systems.* ACM, June, 1982.

[Goldberg 82] Goldberg, A.J., Robson, D., and Ingalls, D.H.H.
Smalltalk-80: The Language and its Implementation.
1982.
Forthcoming book.

[Goldstein and Bobrow 80]
Goldstein, I.P., and Bobrow, D.G.
Extending Object Oriented Programming in Smalltalk.
In *Proceedings of the Lisp Conference.*
Stanford University, 1980.

[Ingalls 78] Ingalls, D.H.H.
The Smalltalk-76 Programming System: Design and Implementation.
In *Proceedings of the Fifth Annual Principles of Programming Languages Symposium,* pages 9-16. ACM, January, 1978.

[LRG 81] The Xerox Learning Research Group.
The Smalltalk-80 System.
Byte 6(8):36-48, August, 1981.