To be published in: LISP AND SYMBOLIC COMPUTATION: An International Journal, 4, 3, 1991

Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF*

CRAIG CHAMBERS
DAVID UNGAR†
BAY-WEI CHANG
URS HÖLZLE (*self@self.stanford.edu*)

Computer Systems Laboratory, Stanford University, Stanford, California 94305

Abstract. The design of inheritance and encapsulation in SELF, an object-oriented language based on prototypes, results from understanding that inheritance allows parents to be shared parts of their children. The programmer resolves ambiguities arising from multiple inheritance by prioritizing an object's parents. Unifying unordered and ordered multiple inheritance supports differential programming of abstractions and methods, combination of unrelated abstractions, unequal combination of abstractions, and mixins. In SELF, a private slot may be accessed if the sending method is a shared part of the receiver, allowing privileged communication between related objects. Thus, classless SELF enjoys the benefits of class-based encapsulation.

1 Introduction

Inheritance is a basic feature of most object-oriented languages. Many of these languages are based on *classes* and use inheritance to allow a class to obtain methods and instance variables [26]. (Sometimes classes and inheritance are also used to specify type compatibility.) Within the last few years, however, there have been several proposals for languages based on *prototypes* [3, 9, 10, 12, 22, 25]. These languages do not include classes but instead allow individual objects to inherit from (or delegate to) other objects. The issues surrounding inheritance and encapsulation need to be revisited when designing such a language.

This paper describes the inheritance and encapsulation mechanisms we designed and implemented in one prototype-based language, SELF [4, 5, 11, 25]. Our design

*This work has been generously supported by National Science Foundation Presidential Young Investigator Grant #CCR-8657631, and by Sun Microsystems, IBM, Apple Computer, Cray Laboratories, Tandem Computers, NCR, Texas Instruments, and DEC.

†Author's present address: Sun Microsystems, 2500 Garcia Avenue, Mountain View, CA 94043.

is based on the philosophy that an object's parents should be treated as shared parts of the object, and that inheritance should be a simple, declarative way to maximize the possibilities for sharing. This paper describes two heretofore unpublished innovations: a *prioritized multiple inheritance* scheme that unifies unordered and ordered multiple inheritance, and an *object encapsulation* model that provides many of the benefits of class-based encapsulation in a language without classes. In addition, our inheritance system supports *directed* and *undirected resends* to forward messages to an object's ancestors, a unique *sender path tiebreaker rule* that resolves many ambiguities between unrelated unordered parents, and *dynamic inheritance,* which allows an object to change its parents at run-time to effect significant behavioral changes due to changes in its state.

1.1 The Benefits of Inheritance

We feel that there are two reasons to include inheritance in a dynamically-typed language like SELF: *malleability* and *reusability.*[1] Inheritance allows the behavior common to a set of objects to be factored out into a single shared object, such as a superclass. As a result, the behavior of every object in the set may be changed with only a single change to the shared object. This malleability facilitates program construction, maintenance, and extension.[2]

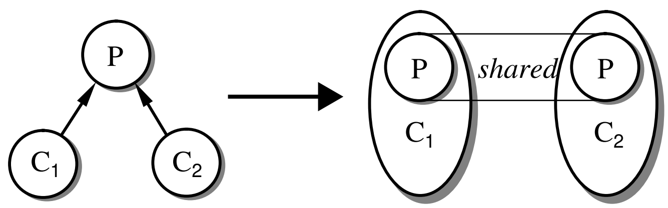
Inheritance encourages the reuse of code and data by allowing the programmer to write new abstractions in terms of existing abstractions. By separating out potentially reusable pieces of code or protocol when implementing an abstraction, other abstractions may be able to share the code and avoid duplicating programming effort. This style of programming is called *differential programming* and is one of the most powerful features of object-oriented systems. Programmers need only write the differences from existing code when defining new abstractions; the rest of the code may be shared among the old abstractions and the new abstraction. Improvements to one abstraction automatically propagate to every abstraction that shares the behavior, further amplifying the programmer's power.

1.2 Guiding Principles for the Design of Inheritance in SELF

The common theme in malleability and reusability is *sharing*: one object is shared by other objects, promoting malleability and providing for reusability. Inheritance is just a declarative way of specifying which objects are shared by which other objects. One guiding principle of our design for inheritance in SELF, then, is that *an object's parents are treated as shared subparts of the object.*

[1] In languages with static type systems, there may be other reasons to include inheritance, such as to declare type hierarchies for type checking. We concern ourselves here only with inheritance as a code sharing mechanism.

[2] An alternate approach would be to provide a mechanism to visit every object in some set, and automatically perform the change to each object in the set. This approach has been explored by other researchers [6, 19].



Parents are Shared Parts of their Children

This view of inheritance provides a natural explanation for many aspects of an object-oriented language. The rules for message lookup can be easily derived from the (simpler) rules for message lookup in the absence of inheritance by treating the message receiver's parents as part of the receiver. This view even explains one of the most basic ideas of object-oriented programming: the meaning of *self* within a method. If there were no inheritance, *self* would refer to the receiver object holding the method. With inheritance, since inherited methods are considered to be shared parts of the receiver, the method being invoked is part of the same receiver object. Therefore, *self* always refers to the receiver of the message, even for inherited methods.

Another guiding principle of our inheritance system is to support as much sharing of objects as possible in order to maximize the malleability and reusability of SELF programs. Unfortunately, powerful inheritance schemes have been notoriously difficult to use effectively. This is especially true for those that automatically resolve ambiguities between multiply-inherited conflicting behavior, often confusing users with unexpected but "correct" behavior. Thus, the power of inheritance must be balanced against its complexity and its potential to confound intuition.

1.3 Basic SELF Object Model

An object in SELF contains a set of named slots, each slot containing a reference to another SELF object. A method in SELF is an object that additionally contains code to execute when invoked; a method object is viewed as a prototypical activation record, and its slots are its arguments and local variables. An object may allow assignments to a data slot by associating an *assignment slot* with the data slot. New objects are created by *cloning* (shallow-copying) pre-existing objects. For a more complete description of SELF's syntax and object model, see [5, 11, 25].

In the absence of inheritance, message lookup in SELF is handled by *searching* the receiver for a slot that matches the message name and *evaluating* the contents of the matching slot (or generating a `messageNotUnderstood` error if there is no matching slot). Evaluating a method executes its associated code. Evaluating a simple data object just returns the data object itself. Since there is no way in SELF to access state other than by sending messages, state access is unified with method invocation; this guarantees that SELF code is always representation-independent.

2 Prioritized Multiple Inheritance

SELF's inheritance system supports multiple inheritance by allowing an object to have more than one parent. Most modern object-oriented systems support some form of multiple inheritance, since it offers significantly more possibilities for sharing than simple single inheritance. But multiple inheritance introduces a new complexity: two or more parents may define slots with the same name.

2.1 Ordered vs. Unordered Multiple Inheritance

Previous languages belong to one of two camps in handling this name clash problem. Some languages, like New Flavors [16], CommonLoops [2], and CLOS [1], rank an object's parents, and automatically resolve the ambiguity in favor of the slot inherited from the highest-ranked parent. We call this approach *ordered inheritance*. Ordering an object's parents works best when the object is more like one parent than the others, or when programming using shared *mixins* which are designed to override behavior "from the side."

In addition to ordering, these languages *linearize* the inheritance graph, constructing a total ordering of all classes that is consistent with each class' local total ordering, defined as the class followed by its direct superclasses in order. An error results if there is no global total ordering that is consistent with each class' local total ordering. Linearization has two drawbacks: it masks ambiguities between otherwise unordered ancestors, and it fails with inheritance graphs that it deems inconsistent.

The opposite approach to resolving ambiguities among an object's parents is to leave it up to the programmer. Languages like Trellis/Owl [17, 18], Eiffel [14, 15], C++ version 2.0 [23, 24], and CommonObjects [20, 21] treat an object's parents as equals without a relative ordering and treat ambiguities as programming errors that need to be explicitly resolved by the programmer.[3] We call this approach *unordered inheritance*. It works best when combining relatively equal parents or unrelated parents, since any ambiguities are likely to need explicit resolution by the programmer. Artificially ordering equal parents would mask these ambiguities, introducing subtle and obscure errors in programs.

[3] C++ orders superclasses to invoke constructors and destructors but not to resolve virtual function calls.

2.2 Prioritized Parents

Both approaches have their advantages and disadvantages. We have developed a simple new approach that combines ordered and unordered multiple inheritance, even within the same object. Each of an object's parents has an associated *priority*. Parents at different priority levels are *ordered*, with the higher-priority parents' slots taking precedence over the lower-priority parents' slots if any names clash. Parents at the same priority level are *unordered* with respect to each other, and accesses to any clashing slot definitions will generate an ambiguous message error. Priorities allow the programmer to make the best choice for each situation.

To support prioritized multiple inheritance, the receiver's parents are treated as shared subparts; the parent objects are themselves extended with their parents' slots, and so on. If an object and one of its ancestors define slots with the same name, then the object's slot takes precedence over the ancestor's slot; this implements the standard rule that an object may override its ancestors' slots. If two of an object's parents define (or inherit) slots with the same name, then the slot from the higher-priority parent takes precedence; if both parents are of the same priority (i.e., unordered), then the system generates a `messageAmbiguous` error on access to the slot. An ancestor's slots are only included once, no matter how many paths lead from the receiver to the ancestor, so an object won't generate ambiguities with its own slots if it is inherited along several paths. This rule also handles cycles in the inheritance graph, since inheriting an ancestor repeatedly in a cycle has the effect of including it just once.

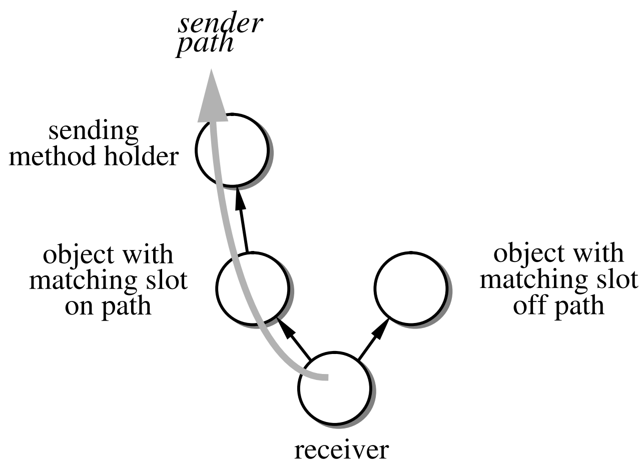
An object's parents are found in its *parent slots*, which are normal data slots that have been marked with a parent priority.[4] If a parent slot is assignable, an object may change that parent by assigning to its slot. This feature is called *dynamic inheritance*.

2.3 Sender Path Tiebreaker Rule

SELF incorporates a unique feature that frequently resolves ambiguities when inheriting equally from two unrelated abstractions. If a method in an object sends a message to *self*, it expects to find the matching slot in one of the object's descendants or one of its ancestors. This corresponds to the method invoking one of the methods defined in a more general abstraction (an ancestor) or in a refining abstraction (a descendant). However, if a descendant of the object containing the method uses unordered multiple inheritance to combine the object with an otherwise unrelated object that happened to define a matching slot, then the message would become ambiguous. To resolve such ambiguities, SELF's inheritance rules additionally specify that if two slots with the same name are defined in equal-priority

[4]Priorities are syntactically indicated by adding one or more asterisks (`*`) after the slot name, with more asterisks indicating lower priority (the asterisks are more like footnote-style asterisks than movie ratings).

parents of the receiver, but only one of the parents is an ancestor or descendant of the object containing the method that is sending the message (the *sending method holder*), then that parent's slot takes precedence over the other parent's slot.



Sender Path Tiebreaker Rule

This *sender path tiebreaker rule* automatically resolves ambiguities between unrelated abstractions if the sender of the message is part of one abstraction but not the other. An ambiguity may be resolved in favor of one parent for one message send, and in favor of another parent for a different message send, depending on the location of the sending method holder. This dynamic behavior would be difficult to program explicitly without changing the slot names of one of the inherited abstractions.

The sender path tiebreaker rule is unique to SELF. Trellis/Owl, C++, and CommonObjects can obtain similar results in some cases by statically binding calls to private members. This approach is inferior to a general sender path tiebreaker rule because it fails to disambiguate references to public and subclass-visible (protected) members.

2.4 Resends and Directed Resends

To support differential programming at the method level, many object-oriented languages include a *resend* mechanism that allows a method to be written as a variation of the method it overrides, invoking the overridden method as part of the overriding method. In SELF, a method may "continue the lookup" to find the next matching slot that the resending method is overriding by prefixing the name of the message with the reserved word `resend` followed by a period; for instance, `resend.clone` resends the `clone` message, finding whatever `clone` slot this method overrides. These resends may be chained, with one method doing a resend to call an overridden method, which in turn does another resend, and so on. A single method may do any number of resends, and although the receiver of a resend must be *self*, each resend may have a different message name and different arguments from the resending method.

SELF also includes a variant of the resend mechanism that directs the message lookup only to one of an object's parents instead of to all of them. This *directed resend* is normally used to explicitly resolve ambiguities among an object's parents. Syntactically, directed resends are specified by prefixing the name of the message with the name of one of the sending method holder's parent slots followed by a period, analogous to normal resends specified using the `resend` reserved word (since `resend` is reserved, it can't be mistaken for the name of a parent slot). For example, `clonableTraits.clone` resends the `clone` message to the parent object referenced by the `clonableTraits` parent slot of the sending method holder. Directed resends are not a general delegation mechanism because they can only be directed to parents of the sending method holder. General delegation (i.e., starting the message lookup with an arbitrary object) is provided in SELF using special primitives.

Flavors, CommonLoops, and CLOS use *method combination rules* to handle the case where multiple classes define methods with the same name. The standard method combination rules in CLOS support `call-next-method`, which is similar to SELF's undirected resend, except that `call-next-method` does not allow the programmer to change the name of the message. The standard method combination rules include combinations not provided by SELF, for example, `:before` and `:after` dæmons and `:around` methods. Users may even write their own method combination rules. This approach is more powerful and flexible than our inheritance rules. However, we want SELF's inheritance rules to be simple; we feel that the added expressiveness of user-definable method combination would be outweighed by the extra complexity in the language.

In most languages with unordered inheritance, ambiguities must be resolved statically. Eiffel forces programmers to `rename` inherited features to either disambiguate or resend to them. Trellis/Owl allows programmers to select one of the inherited definitions using the `inherit` clause. Trellis/Owl, C++, and CommonObjects also support mechanisms similar to SELF's directed resends. None of these languages support undirected resends and so must cope with three problems:

- Directed resends may mask ambiguities that would be caught using undirected resends.
- Directed resends may need to be changed if the names of an object's parents are changed or if the inheritance hierarchy is changed.
- Directed resends may not be used in mixins, where the method invoked by the resend differs depending on what objects are mixed in at a lower priority.

In BETA [8, 13], virtual functions are invoked from *least specific* to *most specific*, with the keyword `inner` being used to invoke the next more specific method. This mechanism is a product of the philosophy in BETA that subclasses should be behavioral extensions to their superclasses and therefore specialize the behavior of their superclasses at well-defined points (i.e. at calls to `inner`). It is much more restrictive than SELF's resend mechanism.

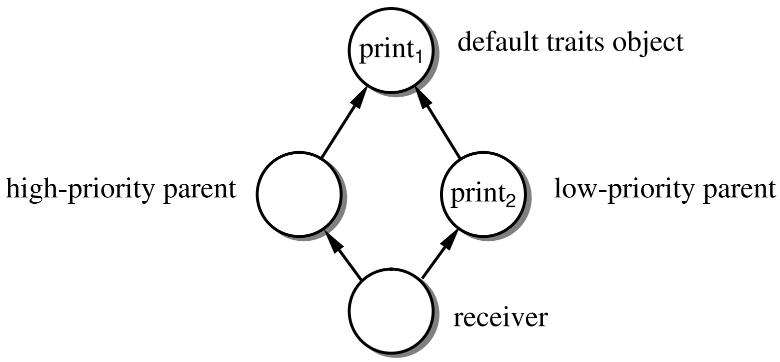
2.5 Complexities with Ordered Multiple Inheritance

Ordered multiple inheritance is quite powerful, supporting both unequal multiple inheritance and mixins. These programming techniques are very useful, and we make significant use of them in our own SELF code. However, the power of ordered multiple inheritance has a few hidden drawbacks, and the effects of ordering parents can occasionally surprise novice and expert SELF programmers alike.

2.5.1 Ordered Multiple Inheritance, Mixins, and Resends

One consequence of using higher-priority parents to implement mixins in SELF is that these mixin objects should almost always be defined with *no* parents. Otherwise, the slots of the mixin's ancestors, no matter how general, would override any slots in lower-priority parents, no matter how specific. For example, if a mixin inherited $\texttt{print}_1$ from some default behavior traits object, then this default behavior would override any specific behavior of the lower-priority parent such as $\texttt{print}_2$. This is almost never what the programmer intends. Once bitten, however, the programmer learns to leave mixins parentless to prevent them from mixing in more behavior than expected.

Another potentially surprising effect of ordering an object's parents is that a chain of resends may eventually "backtrack" and call a method defined in a lower-priority parent of a *descendant* of the sending method holder, if no more ancestors of the descendant's higher priority parent contain matching slots. Continuing the example, a resend from $\texttt{print}_1$ would invoke $\texttt{print}_2$. This is a desirable feature of ordered multiple inheritance and resends, since it allows mixins to invoke the methods that they override, which are defined in lower-priority "cousins." However, backtracking to lower-priority branches may surprise the novice programmer in other situations, especially if the lower-priority branch is a child of the resending method.



Problems with SELF's Priorities and Mixins

We are considering altering SELF's lookup rules to always search children before their ancestors, even if the children are on lower-priority paths than the ancestors; Flavors and CLOS use a similar rule when linearizing classes. This would have the effect of using the declared priorities to break ties between parents left unordered by the inheritance graph. Both of the problems described above would be remedied by this change. In the example above, $\texttt{print}_2$ would be found before $\texttt{print}_1$, and resends in $\texttt{print}_1$ would never invoke $\texttt{print}_2$.

2.5.2 Ordered Multiple Inheritance, Resends, and Dynamic Inheritance

Ordered multiple inheritance, resends, and dynamic inheritance have complex interactions. Dynamic inheritance does not normally affect message lookup, since assignable parents cannot change while a message send is being handled. However, they *may* change between resends within a chain of resends. This would not be a problem in a system with single inheritance or unordered multiple inheritance, since the message lookup could always begin with the resending method holder's parents and proceed upwards. But with the introduction of ordered multiple inheritance, a resend might have to backtrack to a lower-priority parent of a descendant of the resending method holder to find the next matching slot. If a parent between the receiver and the resending method holder were changed between resends, it would be difficult to determine what the next matching slot should be, especially if the resending method holder were no longer an ancestor of the receiver at all.

SELF's current rules for resends in the face of dynamic inheritance are complex to explain and to implement. It has become the "tar baby" of SELF's inheritance system, illustrating the potential dangers of combining seemingly well understood, innocuous language features. We are actively debating possible solutions.

2.6 Inheritance Rules as a Partial Order on Ancestors

The lookup rules for inheritance define a *partial order* on the receiver and its ancestors, derived from the inheritance graph, the receiver, and the sending method holder. This order is defined according to the following rules:

- An object's higher priority parents (and their ancestors) are ordered before the object's lower priority parents [*priorities*].
- One object is ordered before another if they are left unordered by the previous rule and all paths in the inheritance graph from the receiver to the second object pass through the first object [*children before ancestors*].
- One object is ordered before another if they are left unordered by the previous two rules and the first object lies on the sender path while the second does not [*sender path tiebreaker*]. An object lies on the sender path if the sending method holder is an ancestor of the receiver, and if the object either lies on a path in the inheritance graph from the receiver to the sending method holder or is an ancestor of the sending method holder.

This partial order is used to determine the results of message lookups. Normal lookups search the nodes in the partial order for ancestors that contain slots that match the name of the message. If no matching slot is found, then the message is not understood. If there is no single greatest ancestor containing a matching slot (ancestors earlier in the partial order are considered greater than those later in the partial order), then the message is ambiguous. Otherwise the greatest ancestor containing a matching slot is the result of the lookup, and the contents of the matching slot is evaluated as the result of the message. Resends are similar to normal lookups, but they only search ancestors in the partial order that are *less* than the resending method holder; if the resending method holder is not in the partial order (because it has been spliced out by dynamic inheritance), then the message is not understood. Directed resends are like resends, except that all parent slots of the resending method holder except the one being directed through are ignored when constructing the partial order.

The rules described above fail to cope with cycles in the inheritance graph. While it appears possible to accurately model SELF's treatment of cycles, doing so would complicate the rules so much that they would no longer be useful in understanding the normal acyclic situation. This difficulty indicates a need to revisit our treatment of cycles, perhaps by making all ancestors in a cycle mutually incomparable (or equivalently, by treating the inheritance graph as a preorder, in which all ancestors in a cycle are equal).

This partial ordering is different from the linearization used in Flavors and CLOS. Their linearizations are *total* orders performed once for the entire system. Our ordering is partial (so that real ambiguities are not resolved arbitrarily by the system) and is (conceptually) constructed dynamically for each message send; different receivers may have different "views" of the ordering of objects in the inheritance hierarchy, and different message send sites may have different constructed partial orders because of the sender path tiebreaker rule.

3 Encapsulation

Inheritance in SELF as defined so far does not support data abstraction and information hiding very well. Any object may send any message to any other object; as long as a matching slot is found, its referent is evaluated. Such unrestricted message passing prevents an object from maintaining local invariants about its data, since any other object can invoke its assignment slots freely. In addition, unrestricted access to slots camouflages an object's external interface. Lacking the means to distinguish between an object's internal implementation and its external interface, an object's clients may invoke an operation that was intended to be for internal use only, creating an unwanted coupling between the clients and the object's implementation. These problems may not be serious for an individual's exploratory programming—in fact unrestricted access may be desirable—but they hinder the construction of more permanent, reusable abstractions.

3.1 Visibility Declarations

SELF allows individual slots to be declared either *public* or *private*; a slot with no such declaration is said to be *unspecified*. Public slots, prefixed with a circumflex (^), are part of the external interface of an abstraction and may be invoked by messages from any object. Private slots, prefixed with an underscore (_), are invisible to other objects.

Since assignment slots are declared simultaneously with their corresponding data slots, a ^ or _ declaration applies to both the data slot and the assignment slot. Frequently the programmer may want the data slot to be publicly accessible while protecting the assignment slot. To specify mixed-mode declarations, the SELF programmer may prefix a data slot/assignment slot declaration pair with ^_ (meaning public data slot and private assignment slot) or _^ (meaning private data slot and public assignment slot). Trellis/Owl and CommonObjects also support different visibility declarations for data access and assignment. This construct makes it easy for the programmer to limit either access or modification of data as appropriate.

Unspecified slots, with no prefix, act like public slots, but the connotations to SELF programmers are different. During rapid development of code, it may be convenient to ignore encapsulation issues until the abstractions become more mature; visibility declarations can just be left off, making all slots accessible. Then, as code solidifies, slots may be annotated with public and private declarations to better define the external interfaces.

We have deliberately chosen a terse, per-slot syntax to make it as easy as possible for the programmer to add and modify visibility declarations. Some slots may remain unspecified, either because the implementor has not yet decided whether the slot should be public or not, or because access is required for some cooperating abstraction, but the slot should not be generally considered part of the external interface to the abstraction. This distinction between unspecified scope and public scope is a unique feature of SELF designed to support both exploratory and production programming.

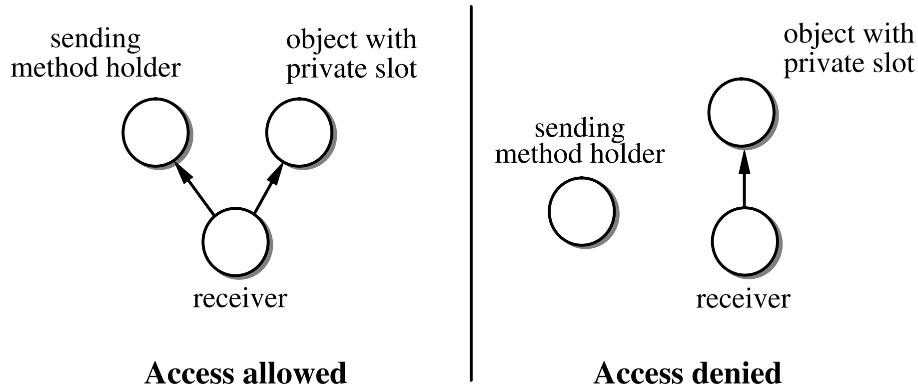
3.2 The Meaning of Privacy

Existing encapsulation systems are either object-based or module-based (class-based). For example, the Smalltalk-80[5] language [7] provides object-based encapsulation of instance variables, meaning that a method may only access the instance variables of *self*. Trellis/Owl and Eiffel use object-based encapsulation for both instance variables and methods. Unfortunately, object-based encapsulation alone is too restrictive. Consider a method to add two points, which must create a new point and set its `x` and `y` slots. Since the new point is not the same as *self*, there must be a public method to set `x` and `y`. Module-based encapsulation, on the other hand, allows code defined within a module or class to access the private methods and data of *any* object of the module, enabling many more internal operations to be declared private. C++ is an example of a language that uses module-based encapsulation. Module-based encapsulation is a minimum level of encapsulation needed for realistic programs to protect their private data and operations from outsiders without unduly restricting internal access.

How can SELF, a language without explicit modules or classes, support module-based encapsulation? The guiding principle that parents are shared parts leads to a surprising but logical solution: since a method is shared by all objects that inherit the method, the method should have access to the private slots of every object that shares (inherits) it. The shared ancestor object defining the method thus forms its own "module" of sorts, since its methods may access the private slots of any of its descendants. Consequently, the scope of a private slot is large enough to allow most slots that shouldn't be part of the public interface to be declared private.

The complete rule for private slot accesses is that a private slot is accessible if both the sending method and private slot are in objects that are the same as or ancestors of the receiver, that is, if the objects holding the sending method and the private slot are shared subparts of the receiver. In the case of adding two points, the addition method is in a parent of—in other words, is a part of—the new point, and the private slots `x:` and `y:` are in the new point, so the access would be permitted. Encapsulation in SELF provides the benefits of class-based or module-based encapsulation without requiring the existence of explicit classes or modules. These

[5] Smalltalk-80 is a trademark of ParcPlace Systems, Inc.



advantages are a direct consequence of the consistent treatment of parents as shared parts of objects.

One property of these rules is that an object may gain access to any private slot by becoming a child of the object with the private slot. In the presence of dynamic inheritance, it becomes difficult to statically determine all objects that may be children of an object with a private slot, preventing air-tight proofs that the invariants of an implementation are maintained. In practice, this has not been a problem, but we continue to investigate ways to support static reasoning about encapsulation in the presence of dynamic inheritance.

In SELF, private slots are visible to any children, since parents are shared parts. Other languages provide the programmer with more choices. Trellis/Owl, C++, and CommonObjects also enable the programmer to hide members from subclasses. C++ supports *friend* classes and methods that have access to a class' private members, useful for cooperating abstractions. Similarly, Eiffel allows members to be selectively exported to a list of cooperating abstractions; this feature may be used to overcome Eiffel's object-based encapsulation model, at some cost in verbosity.

4 Conclusion

Inheritance confers its benefits of malleability and reusability by supporting sharing while minimizing unexpected surprises. Accordingly, SELF's design is based on interpreting an object's parents as shared parts of the object. SELF's inheritance system introduces *prioritized multiple inheritance*, a simple way to describe both unordered multiple inheritance and ordered multiple inheritance, and provides three ways to resolve ambiguities. The *sender path tiebreaker rule* automatically resolves ambiguities that are almost certainly caused by accidental naming conflicts between unrelated ancestors of an object. *Directed resends* resolve ambiguities among an object's parents for a *particular* message by resending the message to one of the object's parents. *Parent priorities* resolve ambiguities among an object's parents for *all* messages by ranking the object's parents. For the most part, these mechanisms are simple and intuitive; unfortunately, the combination of ordered inheritance and resends sometimes leads to surprising behavior. Unlike previous multiple inheritance schemes, prioritized multiple inheritance supports a wide range of uses: differential programming of abstractions and methods, combination of unrelated abstractions, unequal combination of abstractions, and mixins. No other language supports all of these programming techniques as well.

SELF incorporates encapsulation to assist programmers in identifying an object's public interface and in reasoning about the behavior of a module without examining all of its clients. The syntax has been carefully chosen to encourage programmers to use it: it is concise, it treats each slot independently, it separates access from assignment, and it allows a slot's visibility to remain unspecified. Unspecified slots free the programmer from making public declarations while engaged in exploratory programming. As an object's interface becomes more defined, it can be annotated gradually with visibility declarations. This smooth transition is unique to SELF. Considering parents as shared parts has led to powerful encapsulation semantics. A method may access a private slot if both the method and the private slot can be considered to be part of the receiver. This definition allows most of the slots that should be private to be declared private. SELF is the first prototype-based language to provide the benefits of module-based encapsulation.

There are three lessons from this work: inheritance allows an object to be in two places at the same time, neither ordered nor unordered multiple inheritance is sufficient unto itself, and a language need not include classes to gain the benefits of class-based encapsulation.

5 Acknowledgments

Randy Smith co-designed the original SELF language with the second author (before prioritized multiple inheritance and encapsulation), and continues to be a useful sounding board for new ideas. Peter Deutsch provided much inspiration for the pursuit of SELF. Elgin Lee helped in the early design and implementation of prioritized multiple inheritance. Martin Rinard provided valuable suggestions for the section on inheritance as a partial order on ancestors.

References

1. Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. Common Lisp Object System Specification. Published as *SIGPLAN Notices*, 23, 9 (1988).

2. Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. CommonLoops: Merging Lisp and Object-Oriented Programming. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 17-29.

3. Borning, A. H. Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference* (1986) 36-40.

4. Chambers, C., and Ungar, D. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*. Published as *SIGPLAN Notices*, 24, 7 (1989) 146-160.

5. Chambers, C., Ungar, D., and Lee, E. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*. Published as *SIGPLAN Notices*, 24, 10 (1989) 49-70. Also to be published in *Lisp and Symbolic Computation*, 4, 3 (1991).

6. Cunningham, W. Objects without inheritance. Personal communication (1989).

7. Goldberg, A., and Robson, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA (1983).

8. Kristensen, B. B., Madsen, O. L., Møller-Pedersen, and Nygaard, K. The BETA Programming Language. In Shriver, B., and Wegner, P., editors, *Research Directions in Object-Oriented Programming*, The MIT Press, Cambridge, MA (1987).

9. LaLonde, W. R. Designing Families of Data Types Using Exemplars. In *ACM Transactions on Programming Languages and Systems*, 11, 2 (1989) 212-248.

10. LaLonde, W. R., Thomas, D. A., and Pugh, J. R. An Exemplar Based Smalltalk. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 322-330.

11. Lee, E. *Object Storage and Inheritance for SELF, a Prototype-Based Object-Oriented Programming Language*. Engineer's thesis, Stanford University (1988).

12. Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 214-223.

13. Madsen, O. L., and Møller-Pedersen, B. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *OOPSLA '89 Conference Proceedings*. Published as *SIGPLAN Notices*, 24, 10 (1989) 397-406.

14. Meyer, B. Genericity versus Inheritance. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 391-405.

15. Meyer, B. *Eiffel: An Introduction, Version 2.1*. TR-EI-3/GI, Interactive Software Engineering, Inc., Goleta, CA (1988).

16. Moon, D. A. Object-Oriented Programming with Flavors. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 1-16.

17. Schaffert, C., Cooper, T., and Wilpolt, C. *Trellis Object-Based Environment: Language Reference Manual, Version 1.1*. DEC-TR-372, Digital Equipment Corp., Hudson, MA (1985).

18. Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. An Introduction to Trellis/Owl. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 9-16.

19. Smith, R. B. Objects without inheritance. Personal communication (1990).

20. Snyder, A. *CommonObjects: An Overview.* STL-86-13, Hewlett-Packard Laboratories, Palo Alto, CA (1986).

21. Snyder, A. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 17-29.

22. Stein, L. A. Delegation Is Inheritance. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 22, 12 (1987) 138-146.

23. Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading, MA (1986).

24. Stroustrup, B. The Evolution of C++: 1985 to 1987. In *USENIX C++ Workshop Proceedings* (1987) 1-21.

25. Ungar, D., and Smith, R. B. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 22, 12 (1987) 227-241. Also to be published in *Lisp and Symbolic Computation*, 4, 3 (1991).

26. Wegner, P. Dimensions of Object-Based Language Design. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 22, 12 (1987) 168-182.