# Automated Delegation is a Viable Alternative to Multiple Inheritance in Class Based Languages

**John Viega**
Reliable Software Technologies
Sterling, VA
viega@list.org

**Paul Reynolds**
Department of Computer Science
University of Virginia
Charlottesville, VA
pfr@cs.virginia.edu

**Reimer Behrends**
Department of Computer Science
University of Kaiserslautern
67653 Kaiserslautern, Germany
behrends@list.org

## Abstract

Multiple inheritance is still a controversial feature in traditional object-oriented languages, as evidenced by its omission from such languages as Modula-3, Objective C and Java™. Nonetheless, users of such languages often complain about having to work around the absence of multiple inheritance. Automating delegation, in combination with a multiple subtyping mechanism, provides many of the same benefits as multiple inheritance, yet sidesteps most of the associated problems. Automated delegation could satisfy both the designers and the users of class based object oriented languages.

In this paper, we discuss why automated delegation is desirable. We also present *Jamie*, a freeware preprocessor-based extension to Java that offers such an alternative.

## 1 Introduction

This paper discusses a language mechanism called *automated delegation*, which we believe can be a desirable alternative to multiple inheritance in class based languages. The purpose of this feature is to automate the practice of forwarding messages to contained classes (commonly called *delegation*). This automation provides an explicit mechanism for abstraction, instead of leaving the user to devise ad hoc solutions. At the same time, automated delegation retains many of the advantages of multiple inheritance, while avoiding its principal drawbacks.

### Terminology

This section defines the terms that will be used throughout this paper. Most of these terms are in common use within the object-oriented community, yet have no universally accepted definition.

#### Subclassing

Subclassing is the derivation of methods and possibly variables from another class. For example, multiple inheritance, as is found in languages like C++ [Str97], is a subclassing mechanism, as it allows the user to use methods from another class.

A subclassing relationship does not imply that type inheritance exists. For example, Sather [Omo93] allows the user to use implementations as a sort of code inclusion; the subclassing mechanism does not perform type inheritance under any circumstances.

The implementation is unimportant, as long as the user may reuse code. For example, delegation is a form of subclassing, even though delegation usually uses a class instance to achieve reuse, as opposed to sharing the blueprint of a class.

Given a class X, a *subclass* of X is any class Y that performs subclassing in order to derive methods or data from class X. In such a case, X is considered to be a *superclass* of Y.

*Multiple subclassing* is directly subclassing from multiple (potentially unrelated) classes at once. A multiple subclassing mechanism should allow the programmer to subclass from an arbitrary number of classes at once.

#### Subtyping

*Subtyping* is the ability for a class to derive its type from another class. An instance of the derived type may be substituted for an instance of the base type, although the reverse may not always be true. If class X is a subtype of class Y, then class X must share class Y's *signature*. In other words, class X's methods and data must be a superset of those provided by class Y. In the Java programming language [AG96], there is a special type of class that is a signature completely devoid of implementation, called an *interface*. A Java interface consists only of method signatures; data may

not be part of an interface[1]. In addition to subtyping through inheritance, a Java class may also subtype by declaring that it *implements* one or more interfaces, meaning that it provides an implementation for each method in the interfaces it declares.

Multiple subtyping is the ability to subtype from multiple classes at the same time. In Java, the programmer may subtype as many times as he wishes, as long as he does so by implementing interfaces.

### Inheritance

There is no universally accepted definition for inheritance. For some people, both subclassing and subtyping are different forms of inheritance. By this definition, the former is implementation inheritance, and the latter is interface inheritance. Also, by this definition, both multiple subclassing and multiple subtyping are to be considered multiple inheritance, and thus, Java is a language with multiple inheritance. However, the common wisdom is that Java does not provide multiple inheritance.

We will use the term *inheritance* to mean subtyping and subclassing coupled together in a single mechanism that is intended for *specialization*. Specialization is the notion that one kind of object is a special kind of another, where all aspects of the extension also are aspects of the original [Tai96]. This relationship is also known as the "IS-A" relationship. However, we will not use the terms inheritance and specialization interchangeably. Inheritance is a mechanism, whereas specialization is a concept. In particular, while inheritance may be intended as a specialization mechanism, it may still be used where the "IS-A" relationship does not apply.

Similarly, we will use the term *multiple inheritance* to mean a feature that provides multiple subclassing as well as multiple subtyping, and is intended for specialization.

### Overview

Section 2 discusses the advantages of multiple inheritance, which an ideal alternative would share. In Section 3, we examine the arguments against multiple inheritance, in order to understand why it is frequently omitted from object oriented programming languages. Section 4 analyzes the most popular strategies for simulating multiple inheritance, including interfaces. Section 5 introduces automated delegation using examples from Jamie, a Java language extension. Section 6 details the major design decisions that must be considered in implementing such a feature, drawing from our experience with Jamie. Section 7 critiques automated delegation, enumerating its advantages and disadvantages. Finally, Section 8 discusses related work, and Section 9 presents our conclusions.

## 2 Advantages of multiple inheritance

While multiple inheritance certainly has its detractors, it is still widely considered a useful feature, as shown by its common use in languages that provide it, such as C++ and Eiffel [Mey87]. It is also a frequently requested feature in languages that do not provide it. Programmers often resort to devising their own mechanisms to accomplish what they otherwise would have used multiple inheritance to accomplish. One of the advantages of multiple inheritance is that it obviates such ad hoc solutions that would otherwise be fairly common. Such workarounds are discussed in Section 4.

There is not a universal consensus as to when multiple inheritance is desirable. However, several researchers have devised similar, though not quite identical lists enumerating beneficial uses of multiple inheritance, including [Str94], [Mey97] and [Sin95]. We consider the following uses of multiple inheritance to be desirable, all of which we have seen promoted in several sources:

1. *Multiple specialization*: An object is conceptually a specialization of two different objects. For example, a class `InputOutputStream` is a specialization of both an `InputStream` and `OutputStream`.

2. *Mixin inheritance*: A *mixin* is a class that encapsulates a general attribute or a set of functionality that may be of interest to many other classes. Mixin classes are generally small, and a class using mixins may often inherit several. The advantage of using mixins is that they encourage programmers to think in terms of modular, highly reusable parts. Mixins are generally not instantiated directly, and may even be independently uninstantiable, as they may depend on the presence of a particular interface in the classes that inherit them. Otherwise, mixins are generally self contained. Mixins may often be a specialization, so there is some overlap between mixins and multiple specialization. For example (from [Mey97]), a class `CompanyPlane` is a reasonable specialization of both `Plane` and `Asset`. However, the class `Asset` represents an intangible property that may apply to a large variety of objects of different types, making it a mixin. However, mixins need not be specializations. For example, simply by renaming the class `Asset` to `ValuableMixin`, the

[1] Although constants may be.

programmer may define a mixin class that is not a specialization of `CompanyPlane`.

3. *Multiple subtyping*: One of the advantageous characteristics of object-oriented languages is *inclusion polymorphism*, i.e., polymorphism through inheritance [CW85]. Multiple subtyping enhances this advantage, as only having a single inherited type is potentially limiting. For example, multiple subtyping is essential for supporting the interface segregation principle [Mar96a], which states that a client should not be required to depend upon an interface it does not use.
4. *Pairing interfaces and implementations*: Keeping interfaces and implementations (subtypes and subclasses) separate helps encourage reuse. However, at some point, interfaces and implementations must be combined to provide a concrete object with an appropriate type. Java supports this type of multiple inheritance, although in a limited manner, since a class may only inherit a single implementation.

## 3 Drawbacks of multiple inheritance

Despite the popularity of multiple inheritance, its appeal is not universal, as demonstrated by its exclusion from other prominent languages such as Modula-3 [Har92], Objective C [Cox84] and Java.

### Name resolution

A common problem that any implementation of multiple inheritance must address is how to handle methods of the same signature inherited from multiple base classes (commonly called a *naming conflict*). Solutions to this problem can be divided into three general categories:

- *Implicit resolution*: The language resolves name conflicts with an arbitrary rule, such as the pre-order traversal of an inheritance tree, as in Python [Lut96]. A common technique that falls under this classification is *linearization* of the inheritance graph, which is essentially reducing the inheritance graph to a flat list. This strategy is common in object-oriented Lisp dialects, including CLOS [Ste90].

- *Explicit resolution:* The programmer must explicitly resolve name conflicts in code. The way programmers must resolve such conflicts varies greatly from language to language. For example, Eiffel requires that the programmer explicitly remove all ambiguity by renaming methods until there are no more clashes. In contrast, C++ allows the ambiguity until used, at which time it requires the caller to explicitly state the base class whose implementation should be used as a part of a method call, unless there is actually only one implementation (i.e., if one virtual base class is inherited multiple times).

- *No resolution*: Naming conflicts are not allowed at all.

Stroustrup notes that his practical experience shows that order dependencies in a language are generally a source of problems. He cites this problem as a reason for requiring explicit resolution [Str94]. Indeed, it is easy to see how it might be problematic if the semantics of a program change based on whether a class inherits `A` before `B`, or `B` before `A`. Linearization has an additional problem in that a class's "real" superclass may not reflect its superclass after the inheritance tree is linearized [Sny86]. This side effect is undesirable because a class may pass messages back to a different base class than the programmer intended. While explicit resolution places the burden of resolving names on the programmer, it does avoid unanticipated, undesirable resolutions.

Another problem with naming conflicts is that two methods with the same signature can be inherited that do not refer to the same conceptual operation, especially when a verb has two different yet common meanings. Eiffel's solution of having the programmer rename methods to resolve name conflicts makes resolving such a problem easier on the programmer. However, a similar feature was considered for inclusion into C++, and later rejected, since such problems do not occur overly often, and such a language feature can lead to following a convoluted trail of chained aliases [Str94]. While most solutions to name conflicts that demand explicit resolution provide a straightforward and clear solution, none of them have completely avoided criticism.

A comprehensive study of issues surrounding naming conflicts is presented in [Knu88], which suggests that attributes should be disambiguated in the class definition, so as to avoid limiting the utility of multiple inheritance. We believe that this solution not only maximizes flexibility but also is simple both for the programmer and the language designer.

### Repeated inheritance

A less straightforward issue that language designers must generally deal with when implementing multiple inheritance is *repeated inheritance*, which is where one class indirectly inherits from another class multiple times. Should inheritance be *virtual*? That is,

should there be only one shared copy of the class inherited multiple times, as in Trellis/Owl [SCB+86]? Should there always be one copy for each time a class is inherited? Or should the programmer have control, as in C++? Virtual inheritance removes multiple copies of instance variables, which saves space and prevents the accidental modification of the wrong set of instance variables. However, unless programmers anticipate potential sharing of instance variables, unexpected side effects may occur. For example, when a class operation performs a depth-first traversal of the inheritance graph, the same class' method may get called twice, unexpectedly [Sny86]. Also, sometimes having a base class explicitly duplicated is the right design decision [Str94].

### Misuse

Yet another problem with multiple inheritance is that it is often overused; i.e., some programmers use it in an unclear or undesirable manner [Boo94]. Programmers often use multiple inheritance to model "HAS-A" relationships, even though they are generally taught to use it only when the "IS-A" relationship is valid (We will discuss what is usually considered acceptable use in more detail in the next section). [Tai96] also notes that multiple inheritance is often used inappropriately, even in the literature. He cites [Mey88], who gave the example of a `Fixed_Stack` class inheriting from classes `Stack` and `Array`. His reasoning is that a fixed-size stack is conceptually a specialization of a stack only, and should therefore use the class `Array` only as a contained component. However, while we agree that multiple inheritance is frequently misused, not everyone considers this example to be such a case. See Section 7 for an in-depth discussion, and its implications for automated delegation.

### Obscurity

Another significant problem with most implementations of multiple inheritance is the potential for obscure code. Take, for example, a class `A` that inherits multiply from classes `B` and `C`, which both define a method `foo()`. When running code in `C` on behalf of `A`, and `C` calls `foo()`, in many languages, `A`'s `foo()` method will be used, which could be the method found in `B`, and not the one found in `C`. When such a thing happens, it is certainly not obvious at all when looking at `C`'s code.

In general, multiple inheritance adds a lot of complexity to an object oriented system, for both the language designer and the end user, and thus is potentially easy to misuse.

## 4 Partial solutions

### Copy and modify

One strategy often used to simulate multiple inheritance is the "copy and modify" strategy. The programmer copies code he otherwise would have inherited, and modifies it if slightly different behavior is required.

This technique suffers from many problems. First, the source code must be available to the reuser. Second, the programmer must deal with an implementation, instead of a class-level interface, thus losing the benefit of abstraction. Third, the burden of maintaining code is left to the person trying to reuse code, instead of the original author [Mar96b]. Fourth, this technique is incapable of simulating subtyping without an additional language level mechanism.

### Base class modification

Another technique used to simulate multiple inheritance is to directly modify the base class in such a way as to obviate the need for multiple subclassing. Such refactoring often involves duplicating code, and thus, this technique tends to suffer from the same problems as does the "copy and modify" scheme. This strategy also has an additional drawback in that the programmer may risk breaking working code.

### Delegation

Delegation, which was introduced in Section 1, is a commonly used technique that can provide many of the same advantages as can multiple inheritance.

As an example of delegation, consider a military simulation where we are writing a class `Tank`, which we would like to have inherit from class `Vehicle`. We would also like class Tank to implement the methods of the `Armored` interface by using an instance of class `TankArmor`, since we may not inherit from both `Vehicle` and `TankArmor`. In Java, we might write the following code:

```
class Tank extends Vehicle implements Armored
{
  private TankArmor myArmor;

  Tank()
  {
    myArmor = new TankArmor();
  }

  // Implement the methods of the Armored
  // interface by forwarding the methods to
  // myArmor.
  boolean protect(Object x) throws InvalidObject
  {
    return myArmor.protect(x);
  }
}
```

This strategy is undesirable, since the user must tediously write a series of small methods that do nothing more than forward responsibility for a method to a delegate (such methods are often called *wrappers*). With the exception of resolving ambiguities, such work would be automated by multiple inheritance, saving the user from a repetitive chore, where he could easily make a mistake. Delegation also is incapable of providing multiple subtyping.

A variation of delegation that allows the programmer to avoid writing wrapper methods is to allow clients access to the delegate. In the previous code example, the programmer could just make the variable `myArmor` public, forcing clients to call the delegate directly when they wish to call the method `protect` in class `Tank`. The drawbacks of this variation are that the `Tank` class no longer implements the `Armored` interface, and that the class may not selectively override any of `myArmor`'s behavior.

### Interfaces

Interfaces, introduced in Section 1, are becoming a popular feature in strongly typed object oriented languages, such as Java and Objective C (where they are called signatures). The Java language designers have claimed that interfaces offer all the desirable features of multiple inheritance, without the drawbacks [GM95]. Interfaces do avoid the principle drawbacks of multiple inheritance, since these drawbacks are, for the most part, the result of multiple subclassing, not multiple subtyping. However, they only completely replace one of the desirable features of multiple inheritance we have enumerated, namely multiple subtyping.

## 5 Automated delegation

Delegation and interfaces coupled go a long way towards being a suitable replacement for multiple inheritance. However, as noted in Section 4, delegation does have drawbacks.

We believe that delegation can be improved upon delegation by directly supporting it in the language. Such support frees the programmer from having to write unnecessary code, providing a structured mechanism to replace an ad hoc strategy.

In this section, we present an example of automated delegation for the Java programming language. However, the concepts should be general enough to map easily to other languages.

### Basic forwarding with Jamie

Jamie is a preprocessor that adds direct support for delegation to Java 1.1[2]. We chose to add our extensions to Java because it already has a multiple subtyping mechanism (interfaces), and because it lacks multiple inheritance, and is unlikely to get it due to the problems with repeated inheritance [AG96].

Jamie automates the process of writing delegation code. Instead of writing short methods that forward appropriate messages to all of the public methods of another object, the programmer just declares that he would like to do so. Returning to our tank example from Section 4, in Jamie we would write the following code:

```
class Tank extends Vehicle
           forwards Armored to myArmor
           implements Armored
{
 TankArmor myArmor;

 Tank()
 {
   myArmor = new TankArmor();
 }
 // rest of class body, if any …
}
```

This code would cause the methods of the `Armored` interface to be delegated to the variable `myArmor`. The variable `myArmor` needs to be a subtype of `Armored`, yet the `Tank` class itself does not have to be, as automated delegation is a subclassing mechanism, and not a subtyping mechanism. For each method in the `Armored` interface, a method will be generated in the `Tank` class with the same modifiers as the `Armored` method[3]. The generated method will dispatch to the variable `myArmor` at runtime, and will closely resemble the forwarding method in the code example from Section 4. In general, forwarding methods will be generated only for methods that are visible to the delegating object. For example, the delegate's private methods would not be visible, the public methods will always be visible, and the protected and default access methods may or may not be visible.

The `forwards` clause shown above has two parts. The first part is the class or interface whose methods are to be delegated. The second is the variable that will handle those methods, which is specified after the `to` keyword. The actual delegate variable must be a subtype of the class or interface whose methods are being delegated. For example,

[2] Jamie is freely available from http://www.list.org/jamie.

[3] Jamie copies all modifiers from the delegate into the forwarder, with the exception of the `final` modifier.

TankArmor is of type Armored, but is not an Armored instance.

In the above code example, if the programmer wished to delegate to all of TankArmor's methods, instead of just the ones in the Armored interface, he could instead write:

```
class Tank extends Vehicle
           forwards TankArmor to myArmor
…
```

When forwarding methods to an interface, the delegate must implement that interface. Otherwise, the programmer must forward to the class of the delegate, or one of its base classes, as in the previous example. The delegating object can implement any interface it fulfills using any combination of its own methods, inherited methods and delegated methods.

Delegation information must appear before an implements clause and after an extends clause, if either exists. The variable specifying the delegate object must refer to a variable defined by the class that is doing the delegating. Such a variable must be visible to the class. In particular, a private variable in the Vehicle class could not be a delegate in this instance, since the Tank class would not have access to that variable. Either a locally declared attribute or an inherited attribute is acceptable, so long as the attribute is an object type (e.g., integers, floats and arrays are not acceptable).

It is possible to forward to multiple delegates. For example, if the Tank should also implement the Armed interface by delegating to an instance of TankWeapons (which itself implements Armed), the programmer may write the following code:

```
class Tank extends Vehicle
           forwards Armored to myArmor,
                    Armed to myWeapons
           implements Armored, Armed
{
  TankArmor   myArmor;
  TankWeapons myWeapons;
 …
}
```

## Exclusion

The Tank class can selectively prevent forwarding methods from being generated by providing its own implementation for any method in the Armored or Armed interfaces. Providing such a method is useful for selectively overriding behavior and is necessary for resolving name conflicts, which are errors in Jamie. For example, if myArmor and myWeapons both provided a method stat(), the programmer would need to resolve the conflict by writing his own stat() method.

If a subclass of Tank were to provide its own implementation of an unambiguous method that Tank delegates, a forwarding method would still get generated inside Tank, although it would be overridden by the definition in the subclass.

If two delegates both inherit from a similar base class or interface, overriding methods can get tedious quickly. Jamie provides a second way to resolve name conflicts that does not involve overriding methods. Take for example, class A, which the programmer would like to use in class T, except it implements S1 and S2, which T already inherits by extending B. If the programmer would like to use the implementations of S1 and S2 from B, he may write the following code:

```
class T extends B
        forwards A without S1, S2 to a
{
  A a;
  …
}
```

The without keyword takes a list of supertypes to exclude from delegation, allowing the programmer to delegate only the methods he needs. Since automatic delegation is a subclassing mechanism only, it is impossible to use the without keyword to make a class that is not substitutable for one of its base types.

In Java, such ambiguities are quite common, since all classes inherit from the Object type. However, Jamie assumes that the programmer would like to refrain from delegating the methods of Object, so the programmer need not do so himself. We discuss the effects of this rule in Section 6.

## Dynamic features

The programmer is responsible for declaring variables for each of his delegates, and assigning instances to those variables, for if the programmer were to leave a delegate uninitialized, and a class then tried to forward a message to it, the runtime system would throw a NullPointerException. This responsibility gives the user the flexibility to instantiate delegates at his convenience, and to initialize delegates with appropriate constructor arguments. It also gives the programmer the flexibility to change the implementation of a delegate at run-time by assigning different objects to the delegation variable. Such an ability supports the encapsulation of logical states [Tai96], also known as the state pattern [GH+95]. For example (adapted from [Tai96]), a programmer may wish for a class Window to inherit

an object that implements the `Displayable` interface with methods such as `draw()`, that will always have to act differently based on whether the window is iconified or visible. By assigning different objects to the delegation variable, he may switch between two different implementations of `Displayable`, one for drawing iconified windows, and one for drawing visible windows. Such a design keeps all mode specific code together, instead of spreading it around via a conditional test in each relevant method, which can have a deleterious effect on code readability [Tai96]. For example, with Jamie, a user may write the following code:

```
interface Displayable
{
  public void draw();
  public void raise();
  public void iconify();
}

interface Window
{
  public void toggleState();
}

class RaisedDisplayer implements Displayable
            forwarder implements Window
{
  public void draw()   { /* draw the window */ }
  public void raise()  { /* already raised  */ }
  public void iconify()
  {
    // add code to iconify the window
    forwarder.toggleState();
  }
}

class IconifiedDisplayer implements Displayable
              forwarder implements Window
{
  public void draw()   { /* do nothing        */ }
  public void iconify(){ /* already iconified */ }
  public void raise()
  {
    // add code to raise the window
    forwarder.toggleState();
  }
}

class MyWindow forwards Displayable to displayer
              implements Displayable, Window
{
  Displayable displayer,icon,raised;
  public MyWindow()
  {
    raised = new RaisedDisplayer();
    icon   = new IconifiedDisplayer();
    displayer = raised;
  }

  public void toggleState()
  {
    if(displayer.equals(raised)) displayer = icon;
    else displayer = raised;
  }
}
```

Note the addition of the keyword `forwarder`, which returns an instance of type `Object`, representing the object that forwarded to the current object. If the current execution is not a result of delegation to the current object, then `forwarder` will be `null`. In order to be able to use the forwarder keyword within a class, the programmer must declare a type to which all forwarding objects must conform.

The `forwarder` keyword, combined with an interface mechanism, obviates supporting uninstantiable mixins, since the delegate has enough information not only to communicate with a delegating object but to be independently instantiable. We feel that the above example is straightforward and powerful, and that it is highly preferable to the code one would write by hand without this extension. For example, to get the effects of the `forwarder` keyword without such an extension, a programmer would likely pass the `this` object (i.e., the current object in Java) as an extra argument to the method in the delegate, or devise some other ad hoc solution.

# 6 Design and implementation

## Syntax

Our original design used a keyword that modified variables that were to be delegated. That design had the advantage that the syntax for delegation did not clutter the inheritance clause. However, it also had some notable drawbacks:

1. It added a large irregularity, in that the language would sometimes use the inheritance clause for subclassing, and would sometimes use a variable modifier.

2. It unnecessarily allowed for the arbitrarily large separation of subclassing information within a single class. We felt that such information should be consistently and conveniently located, if possible.

3. In order to support superclass exclusion using a `without` clause, we would have needed to add significant irregularities to the syntax for variable declarations.

4. The programmer would be unable to delegate to an inherited final variable without aliasing, which would be a minor inconvenience.

5. Consider the following example from Jamie:

```
class FixedStack forwards AbstractStack to container
```

   where `container` is of type `Array`. Our original design had no such facilities, as it would cause unacceptable irregularities in variable declaration syntax. Therefore, to delegate only the methods from class `AbstractStack`, the programmer would have declared the delegation variable to be of type `AbstractStack`. Then, whenever the programmer would want to take advantage of undelegated methods in his implementation, he would either alias a variable of type `Array` (which would increase the object size unnecessarily) or cast the delegation variable.

The mechanism by which a delegate refers back to the forwarder changed significantly several times, as we realized flaws in each of our designs. We wanted delegates to be clearly separated from their clients, and so we chose not to have the `this` keyword point back to the forwarder, as is done with the self-reference operator in many delegation-based languages [Tai96]. Our first mechanism was a `caller` keyword, which returned the object responsible for invoking the current method. We quickly found that such a mechanism did not support procedural decomposition. For example, consider the following code:

```
class Delegate
{
  public void foo()
  {
    System.out.println(caller);
  }
}
```

If an object `X` forwards to an instance of class `Delegate`, when `foo` is called in `X`, `caller` will refer to `X`; however, consider separating the printing code into its own method as follows:

```
class Delegate
{
  public void foo()
  {
    printCaller();
  }
  private void printCaller()
  {
    System.out.println(caller);
  }
}
```

The value of `caller` would always be equal to the `this` reference, since the last call would always be local to the current object, which probably is not what the programmer intended.

We then refined the semantics of the `caller` keyword to return the object that last invoked a method, other than the current object. However, we eventually found that the entire notion of `caller` suffered from two significant problems:

1. While `caller` was an interesting general-purpose mechanism, it usually was not what the programmer expected when the calling object was not the delegating object. Essentially, there was no way for the object to tell if the current call was the result of delegation, or a direct call from a third party.

2. The semantics of `caller` were unclear with respect to forwardings. For example, if an object `O` calls object `X`, which delegates to object `Y`, if object `Y` asked for the value of `caller`, would it get object `X` or object `O`?

Also, we briefly considered replacing the `caller` keyword with an `owner` keyword, since it seemed to be a more accurate representation of the functionality a programmer would generally want when writing delegates. However, the "owner" of a delegate may be ambiguous; one object could easily serve as a delegate to several different clients. For example, multiple objects may wish to delegate to a single cache, which could be stored in a shared class variable.
The `forwarder` keyword, as currently implemented, solves all the problems we found in previous approaches. With it, the delegate can easily distinguish between a method call by delegation and a method call from a third party. Also, the semantics are not ambiguous with respect to which object the caller should be, as the delegating object explicitly distinguishes himself from the caller by use of the `forwards` clause. Finally, there is no sort of ambiguity as there would have been with an `owner` keyword, as the keyword tells the programmer which client is responsible for the most recent delegation.

## Visibility Modifiers

Another important question we had to deal with was what to do about declaring visibility modifiers on the forwarding methods; i.e., should they be based on the visibility modifier given to the variable containing the delegate? For example, we briefly considered the following strategies:

1. Delegate only to objects in variables of public, protected or default access, then copy the access

modifiers of the delegated methods for the forwarding methods.

2. Have the protection level of the variable storing the delegate reflect the most lax protection level a forwarding method can achieve. For example, when delegating to a private variable `X`, all of `X`'s public, protected and default methods would cause forwarding methods to be generated, all of which would be declared to be private, so only the delegating object could use them.

We chose neither of these solutions. They both seemed undesirable, primarily because they are too restrictive: the delegating object may want to control who can assign to the variable by declaring the variable private, yet still have all of the forwarding methods be visible to others. Our choice was to assign the access modifier of the method in the delegate to the forwarding method. This choice has the advantage of not depending on the type of the delegate variable. We believe that this is the desirable choice, because it best supports the usage patterns we are trying to promote with this feature. That is, in our experience, most objects used as delegates were designed to be delegates. If those modifiers are not acceptable to the programmer, he may always subclass off the delegate to change them. Reasonable alternative solutions can be found in other languages, such as public, protected and private inheritance in C++, and the export facility of Eiffel.

## Universal base classes

In Java, as in other languages such as Objective C and Smalltalk [GR89], all classes inherit from the `Object` class, even if indirectly, which causes an unacceptably large number of name conflicts for a multiple subclassing system. As mentioned in Section 5, we chose to avoid such conflicts in this case by explicitly refusing to forward methods originally defined in the `Object` class, despite adding the `without` keyword. While this adds an irregularity to the language, it is an innocuous one. Usually, programmers would prefer to avoid explicitly using the `without` keyword in this instance, unless for some reason they really do need to delegate the methods found in class `Object`. Such a need will certainly be the exceptional case, whereas explicitly specifying `without Object` would get tiresome quickly. Also, Jamie will warn the programmer any time he or she delegates to an object that redefines a method from the `Object` class, reminding the programmer that if that particular method is to be delegated, it must be done explicitly. This strategy prevents the programmer from being surprised by the system not delegating to such a method when he may have expected it to do so.

Another way to solve this problem would be to only allow delegation to variables declared to be interface types. However, we felt this solution would be needlessly restrictive, and would be less useful than our proposed solution in practice. For example, a Java programmer may wish to extend the class `java.util.Vector`, overriding a handful of the methods, but leaving the bulk of them untouched. Without Jamie, this can not be done in any useful manner, since almost all of `Vector`'s methods are `final`, meaning they can not be overridden. Being able to delegate to such a class gives the programmer a reasonable way to extend it.

## Implementation

The decision to implement Jamie as a preprocessor was made primarily so that we could quickly develop a working proof of concept implementation. This decision helped support our notion that delegation is an automation mechanism, since the preprocessor generates code that the user can inspect. Also, being able to inspect the code turned out to be important to us, since we implement delegation by declaring forwarding methods, which posed the problem that if an exception is raised in a delegate, the forwarding method will appear on the stack trace. We found it would be less confusing to the programmer if the stack trace always pointed at the generated Java file, instead of pointing to the Jamie file and having the trace sometimes show methods that could not be found in any code. The preprocessor also has the advantage of keeping our work independent of any particular Java implementation.

However, most aspects of the system could have been implemented far more efficiently if moved from a preprocessor plus library approach to the compiler. For example, our preprocessor keeps track of forwarders by storing them in a stack at delegation time. To ensure that the stack is cleaned up properly, we add the overhead of exception handling to every forwarding method. A language level implementation could avoid such overhead by looking for forwarder information on the call stack. Thus, when an exception gets raised, forwarding information will get cleaned up properly as the call stack is unwound.

## Additional features

Currently, Jamie only forwards methods, although it could forward variables as well. Jamie does not do so for two reasons. First, Jamie's delegation

mechanism is designed to be a complement to Java's interfaces, which can not specify variables. Second, such a feature could not be implemented transparently and efficiently in a preprocessor. This feature may be desirable in other languages.

We also considered modifying the language to not show the forwarding method in a stack trace, but did not do so in the interest of time. For the same reason, we did not implement delegating to a method instead of a variable (e.g., `forwards I to getDelegate()` instead of `forwards I to myDelegate`). These two ideas are likely to be future work on Jamie.

## 7 Analysis of automated delegation

We believe that delegation offers many advantages to a single inheritance class based language, when coupled with multiple subtyping. First, the two together are good at doing the things multiple inheritance does well, such as supporting mixins. Second, they directly support and automate coding techniques that programmers commonly practice in languages without such features. Third, the dynamic nature of delegation supports useful programming techniques that aren't easily achieved in any other manner, such as programming with logical states.

### Comparison to multiple inheritance

Automated delegation is often a useful abstraction tool under circumstances that are not easily and cleanly handled by multiple inheritance, or by any other language feature. Consider Meyer's defense of his use of multiple inheritance to handle a `Fixed_Stack` class which is presented in [Mey97]. In his example, `Fixed_Stack` inherits from both `Stack` and `Array`, where `Stack` is an abstract class provides the skeleton that is filled in by the methods from `Array`. As noted in Section 3, this has been criticized as an inappropriate use of inheritance, as `Fixed_Stack` conceptually is not a specialization of `Array` [Tai96]. Still, if there are a number of similar container classes (e.g., stacks, queues, etc.) in need of an array-based implementation, a well-designed language should facilitate this task by allowing for a clean abstraction. Meyer argues that in this case a class `Fixed_Container`, which implements all the necessary methods as calls to a container attribute of type `Array` would be a both a good abstraction and a class of which `Fixed_Stack` would be a suitable specialization. So creating `Fixed_Stack` as a subclass of both `Stack` and `Fixed_Container` would solve the problem. But then the implementation of that class would simply forward the necessary methods to that of the container object – a tedious and error-prone approach, which is also fragile under change. This drawback can be avoided by inheriting from `Array` directly, rather than seeking the roundabout way via manual delegation. This solution illustrates a tradeoff between the goal of a clean design and that of reliable software.

Obviously, Meyer's argument does not hold in the presence of automated delegation, where such a tradeoff does not occur. Automatic delegation both maintains the conceptual integrity of the model and avoids the artificial introduction of an intermediate class `Fixed_Container`, as in the following example:

```
class Fixed_Stack
      extends Stack
      forwards Array to container
{
   Array container;
   …
}
```

Automated delegation is also less problematic for the programmer, for several reasons:

1. The problems of repeated inheritance are eliminated. Inconsistencies caused by sharing representations in a single class hierarchy should not arise, since the language is only supporting single inheritance and containment.

2. Since delegates are individual objects, and are not part of the delegating object's class hierarchy, local method calls are sure to be handled by the delegate. Therefore, when looking at the code of a delegate, unexpected code paths due to inheritance are far less likely.

3. Delegation is thought of as an operation on a contained object, so the issue of misusing the feature by using it where containment would be a more appropriate mechanism is a moot point. Delegation will only be used when the programmer would like to directly use the methods of a contained object.

Another advantage of our delegation mechanism is that multiple subclassing remains orthogonal to multiple subtyping. This separation strongly encourages programmers to distinguish between interfaces and implementations, which in turn encourages better modularity and code reuse. This distinction also allows the user to subclass, but not subtype when the "IS-A" relationship does not make sense.

Another benefit is that the mechanism only allows for black-box reuse: the delegating object has no special access to the implementation of the delegate. In contrast, multiple inheritance generally (though not always) implies some degree of white-box reuse, which severely weakens encapsulation [Sny86].

Also, the dynamic nature of automated delegation can be an advantage. In addition to supporting logical states, delegation supports subclassing a prototypical instance, which offers an alternative to the abstract, set-theoretic inheritance. This alternative better supports the way people tend to think about real objects [Lie86], and promotes *unanticipated sharing*; i.e., reuse not anticipated by the author of the class [SLU88].

On the other hand, multiple inheritance is still better suited for multiple specialization than is delegation. First, inheritance usually implies an "IS-A" relationship, whereas delegation models an "USES-A" relationship. Second, depending on the language, delegation may not be able to provide *substitutability* (i.e. that the derived class may be used anywhere an instance of the base class is expected). For example, if, in Jamie, the class `S` delegates to instances of class `A`, instances of `S` may not be used wherever an instance of `A` is expected, since there is no way for an `S` instance to be cast to an `A` instance. The best a programmer can do is to have `A` and `S` implement a single interface that should be used in all declaration expecting an object compatible with `A`. Languages that separate implementation inheritance from interface inheritance, such as Sather, would not have this problem.

Also, our mechanism does not solve the problem of name clashes. However, [Knu88] notes that no solution to this problem will always be the best solution. Also, as we mentioned previously, we do not find this complaint to be a significant source of problems in languages with explicit resolution, and believe that any such solution will also be more than acceptable for a delegation based system.

## Drawbacks of automated delegation

There are some potential disadvantages to automated delegation. Its dynamic nature makes it inherently less efficient than static multiple inheritance, as sometimes we can not bind to a single object, since the object we are delegating to may change at runtime. However, even a straightforward implementation would be no less efficient than the code a user would write by hand, and could generally run faster, especially if the implementation avoids invoking forwarding methods when possible, such as by inlining. Also, when a private delegate is only assigned in the constructor (i.e., the compiler can determine that the implementation will not be changed dynamically), the delegate methods could be statically bound, although our implementation makes no such optimizations. And while such a dynamic feature can provide the programmer with much expressive power, if abused, it can lead to code that is harder to read than static software [GH+95].

The `forwarder` mechanism has a potential drawback in that all forwards must conform to a common type. For example, there is no way to specify that interface `I1` requires a forwarder that implements `X`, and that interface `I2` requires a forwarder that implements `Y`. This problem could be fixed with optional additional syntax. One way would be to allow multiple forwarder variables, instead of a `forwarder` keyword. For example, they could be declared in the following manner:

```
forwarder f1 implements X, f2 implements Y
```

However, our implementation does not use such syntax for simplicity's sake. The programmer can enforce his constraints at run-time by specifying the forwarder will be of type `Object`, and then using explicit coercion.

Another potential drawback specific to Jamie and Java that we hope designers of other languages would be able to avoid is the potential for a large separation between the declaration of the delegate variable, and the `forwards` clause. We considered an alternative syntax, where the programmer would actually declare the delegate in the `forwards` clause. For example:

```
class Tank forwards Armored
           to private Armored myArmor
{
 …
}
```

However, this choice had several drawbacks of its own. First, in the previous example, either we would have to allow `myArmor` to be declared multiple times, or we would have to disallow delegating to inherited variables. Second, allowing the declaration of delegate variables outside the class body would add an irregularity to the language. Since Java already has a similar forward referencing problem by allowing methods within the same class to be used before they are declared, allowing the forward referencing is consistent behavior, and avoids adding an irregularity. If designing a language from scratch, however, we would almost certainly devise a suitable syntax to disallow forward referencing.

Bjarne Stroustrup discusses other potential drawbacks to delegation in [Str94]. For a short time, C++ supported a simple delegation mechanism that automated the forwarding of messages to an object. The feature was removed from C++, as it was error prone and confusing [Str94]. Stroustrup believed the two sources of these problems to be:

1. The delegate was an independent object, and thus the delegating object could not override its methods, which could be unexpected if such a method were called directly.
2. There was no straightforward way for the delegate to refer back to the delegating object.

We believe we adequately address the first problem by keeping delegation separate from the notion of type inheritance. In C++, an object could be coerced down to a delegate through assignment or casting. However, even if a delegate declared a method as virtual that was also redefined in the delegating object, after the coercion the delegate would always be called when the method was invoked. The real problem was not the semantics, but that programmers could not remember the semantics; they would assume whatever was most convenient for the code they were writing [Str98]. This particular problem would go away if delegation were separated from type inheritance, as is the case with automated delegation, or if assignment and casting only limited the interface of the delegating object, instead of essentially replacing the object with a delegate. The user would still be able to pass around the delegate as a separate object. However, at that point, the delegate is conceptually an independent object, and that object should indeed be responsible for handling methods explicitly invoked on it, and thus the semantics are clear for the programmer in all cases.

Our mechanism addresses Stroustrup's second problem directly by providing the `forwarder` keyword, allowing the delegate to refer back to the delegating object.

## 8 Related work

Delegation is the foundation of a number of object-oriented languages without classes, such as ThingLab [Bor81], Act-1 [Lie87], a Smalltalk without classes [LTP86], and, perhaps most notably, Self [US87]. Lynn Stein showed such languages to be equally as powerful conceptually as inheritance, although she notes that, in practice, either delegation or inheritance may be more desirable [Ste87]. She even proposes a hybrid model that would allow for both delegation and inheritance in the same language. An excellent analysis of the similarities and differences in both sharing mechanisms is presented in [SLU88]. A hybrid model called *object specialization* is presented in [Sci89]. With object specialization, objects still have a class from which it receives variables and methods, but individual objects determine what they inherit.

As previously mentioned, C++ had a delegation mechanism for a short time before multiple inheritance was added [Str94], but it was error-prone as designed.

### Transframe

The programming language Transframe [Sha97] demonstrates an approach that is superficially similar to ours. Like Java, Transframe allows only single inheritance of implementation. However, multiple inheritance of types is allowed, and it is possible to specify a default implementation for each such type. In such a case, methods of that type are automatically delegated to an anonymous variable referencing the default implementation.

However, this approach has several drawbacks. First, the delegation is merely a mental model for the programmer, and perhaps it does not compare favorably to a fully-featured delegation mechanism. The delegation variable is implicit, and therefore inaccessible; thus, it is not possible for multiple objects to share an instance, or to change the delegate object at runtime.

Also, the actual type of the delegate must be chosen at compile time, resulting in a mechanism that is nearly from traditional multiple inheritance from the programmer's point of view indistinguishable (syntax and terminology aside). One notable exception to this similarity is that with multiple inheritance it is much easier to access the current object than it is to access the forwarding object in Transframe. For the same reason, Transframe style delegation a construct that is strictly weaker than even code inclusion mechanisms, such as the subclassing mechanism of Sather [Omo93].

### Smalltalk-style forwarding methods

Some languages, including Smalltalk and Objective C provide language level support for message forwarding. If a client tries to invoke a method `foo` on object `X`, and `X` does not define `foo`, then the runtime system will call a particular method in `X` to signal that the method was not found, passing in a list of arguments. `X` may ignore the message, causing a runtime error, or forward it on to another object.

This solution suffers from many problems:

1. Forwarding methods have a high built-in overhead. Manually writing forwarding methods

is more efficient. Improving the efficiency of forwarding methods is a difficult job, because the user may write arbitrary code.

2. Since the user may write arbitrary code in a forwarding method, an object may have an inconsistent interface, sometimes responding to a message, and sometimes causing a runtime error.

3. There is no easy way for the programmer to define which subset of a set of messages should be forwarded, and which should result in runtime errors. The only robust solution available to the programmer is to use a construct such as a case statement, and explicitly name each method that should be forwarded. In such a case, the programmer might as well have written forwarding methods. Also, with such a solution, the programmer is responsible for maintaining consistency between messages being forwarded, and messages the delegate handles. That is, when the delegate's interface changes, the user will often need to change the forwarder. Such a dependency should be avoided if possible. A less robust alternative would be to blindly delegate all methods to another object, assuming that if the current class doesn't implement a method, the delegate will not. This method is undesirable because it is important that not only correct code yields correct behavior, but also that incorrect code results in some sort of error. That is, there should be a runtime error if a method is called that the programmer did not intend to forward, but which is present in the delegate object.

4. As with C++, this mechanism generally comes with no way for the delegate to refer back to the forwarder. The programmer must devise his own ad hoc mechanism when such communication is necessary.

5. Forwarding to multiple objects has undesirable semantics. Either the programmer has to name methods explicitly from multiple delegates, or must try delegates until a delegate successfully handles the message.

Automated delegation overcomes all of these problems. First, its delegation mechanism does not allow arbitrary code, which not only affords more efficient implementations but also prevents obscure and ad hoc solutions. Second, it provides forwarding at the class level instead of the method level, meaning that the forwarder's code need not change as the delegates interface changes; the forwarder need only be recompiled. Also, this mechanism allows for concise representation without need for case statements. Third, as previously mentioned, automated delegation provides a way for the delegate to refer to the forwarder, without requiring an ad hoc solution.

## Language support for mixins

Bracha and Cook added direct support for mixins to Modula-3, but they were a static concept, and both a subtyping and subclassing mechanism [BC90].

A method for using parametric polymorphism to support mixins is shown in [AFM97], in the context of a Java language extension. However, this approach to mixins suffers from a few drawbacks. First, the syntax is non-intuitive, in that multiple mixins must be declared as nested template parameters, which forces an ordering, when, conceptually, there should not be one. This also results in a linearized inheritance graph, which, as we noted previously, introduces artificial parents and undesirable order dependencies. Second, to extend a non-parameterized base class such as `Object` and one or more mixins simultaneously, a programmer must either duplicate code, or pass the base class as a parameter to a mixin, both of which are undesirable solutions. For instance, when passing a base class to a mixin, any method defined in both the mixin and the base class will be supplied by the mixin, since the base class will necessarily be a superclass of the mixin. Third, with more complex template specifications, the readability can also degrade significantly. Also, mixin classes must be written so as to take template arguments in order to be composable. For example, the code for class `MyTank`, which inherits from `Tank` as well as mixins `TankArmor`, `TankWeapons` and `Destroyable`, would be written in such a manner:

```
class TankArmor<T>   extends T { … }
class TankWeapons<T> extends T { … }
class Destroyable<T> extends T { … }
class MyTank extends  TankArmor< TankWeapons<
                      Destroyable< Tank > > >
{
 …
}
```

Since, in order to be subclassed in this way, a class must anticipate its use in such a manner, and provide an appropriate template parameter, this solution may not be used for general-purpose multiple subclassing, only as an ad hoc technique. For instance, given two non-parameterized classes `InputStream` and `OutputStream`, a programmer cannot produce `InputOutputStream` as a mixin without changing one of the original classes to have a parameter. This solution will also break all the code relying on a non-parameterized version.

Another drawback of this approach is that the static nature of templates makes it impossible to implement dynamic subclassing in a straightforward manner. We also note that, in our experience, programmers tend not to use templates in this way when using languages with a genericity mechanism as well as a multiple subclassing mechanism, such as C++ and Eiffel.

### Dynamic subclassing

Cecil's predicate classes [Cha93] provide a dynamic subclassing mechanism (i.e., the ability to change a class implementation at run-time). Predicate classes offer the ability to dynamically changing the type of an object as well, all in a single coupled mechanism. For future work, we plan to explore dynamic subtyping issues in Java by extending the interface mechanism, keeping it orthogonal from our delegation mechanism.

An extension to Java based on predicate classes is presented in [VC+97], which is tailored towards the sole goal of specializing classes for efficiency purposes.

## 9 Conclusions

In this paper we have presented automatic delegation for class based languages, which provides a second subclassing mechanism capable of multiple subclassing. We have argued that such a mechanism is preferable to no multiple subclassing mechanism whatsoever. Also, we have compared our delegation mechanism to multiple inheritance, showing that delegation, for the most part, does not suffer from the problems of multiple inheritance. It also enables conceptually clear and desirable abstractions that multiple inheritance does not handle in a satisfactory manner. However, we have also shown that our mechanism has potential drawbacks generally not found in multiple inheritance mechanisms. We hope that language designers will no longer consider the question, "should I allow multiple inheritance?" Instead, we hope they will ask themselves, "Which is the better solution for my language's intended problem domain, multiple inheritance, or multiple delegation?"

## Acknowledgements

We would like to extend our appreciation to Tim Hollebeek, Steve MacDonald, Mike Schatz and John Regehr for their insightful input both in discussions on this work and on drafts of this paper.

We would also like to thank Jonathan Hill, Gabe Ferrer, Kathy Ryall, Zachary Girouard and Leigh Caldwell for reviewing early drafts of this paper.

## References

[AFM97] O. Agesen, S. Freund, J. Mitchell. Adding type parameterization to the Java™ language. In *OOPSLA '97 Conference Proceedings. ACM SIGPLAN Notices 32*, 10 (Oct. 1997).

[AG96] K. Arnold, J. Gosling. *The Java™ Programming Language*. Addison-Wesley, 1996.

[BC90] G. Bracha, W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90 Conference Proceedings. ACM SIGPLAN Notices 25*, 10 (Oct. 1990).

[Boo94] G. Booch. *Object-Oriented Analysis and Design With Applications*, 2nd edition. Addison-Wesley, 1994.

[Bor81] A. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions of Programming Languages And Systems 3*, 4 (Oct. 1981).

[Cha93] C. Chambers. Predicate classes. In *ECOOP '93 Conference Proceedings.*

[Cox84] B. Cox. Message/Object programming: An evolutionary change in programming technology. *IEEE Software 1*, 1 (Jan. 1982).

[CW85] L. Cardelli, P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys 17*, 4 (Dec. 1985).

[GH+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[GM95] J. Gosling, H. McGilton. *The Java Language Environment*. Sun Microsystems, 1995.

[GR89] A. Goldberg, D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[Har92] S. Harbison. *Modula-3*. Prentice Hall, 1992.

[Knu88] J. Knudsen. Name collision in multiple classification hierarchies. In *ECOOP '88 Conference Proceedings.*

[Lie86] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA '86 Conference Proceedings. ACM SIGPLAN Notices 21*, 11 (Nov. 1986).

[Lie87] H. Lieberman. Concurrent Object-oriented programming in Act-1. In A. Yonezawa, M. Tokoro (ed.), *Object-Oriented Concurrent Programming*. MIT Press, 1987.

[LP91] W. LaLonde, J. Pugh. Subclassing ≠ subtyping ≠ is-a. *Journal of Object-Oriented Programming 3*, 5 (Jan. 1991).

[LTP86] W. LaLonde, D. Thomas, J. Pugh. An exemplar based Smalltalk. In *OOPSLA '86 Conference Proceedings. ACM SIGPLAN Notices 21*, 11 (Nov. 1986).

[Lut96] M. Lutz. *Programming Python.* O'Reilly & Associates, 1996.

[Mac87] B. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation*, 2nd Edition. HBJ, 1987.

[Mar96a] R. Martin. The Interface Segregation Principle: One of the many Principles of OOD. *C++ Report 6* (Aug. 1996).

[Mar96b] R. Martin. Granularity. *C++ Report 6* (Nov.-Dec. 1996).

[Mey87] B. Meyer. Eiffel: Programming for reusability and extendability. *SIGPLAN Notices 22*, 2 (Feb. 1987).

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[Mey97] B. Meyer. *Object-Oriented Software Construction*, 2nd edition. Prentice Hall, 1997.

[Omo93] S. Omohundro. The Sather Programming Language. *Dr. Dobb's Journal* (Oct. 1993).

[SCB+86] C. Schaffert, T. Cooper, B. Bullis, M. Killian, C. Wilpot. An introduction to Trellis/Owl. In *OOPSLA '86 Conference Proceedings. ACM SIGPLAN Notices 21*, 11 (Nov. 1986).

[Sci89] E. Sciore. Object Specialization. *ACM Transactions on Information Systems 7*, 2 (Apr. 1989).

[Sha97] D. Shang. *Transframe: the Annotated Reference* (Draft 1.41). 1997. Available from *http://www.transframe.com*.

[Sin95] G. Singh. Single versus multiple inheritance in object oriented programming. In *OOPS Messenger 6*, 1 (Jan. 1995).

[SLU88] L. Stein, H. Lieberman, D. Ungar. A shared view of sharing: the treaty of Orlando. In W. Kim, F. Lochowsky (ed.), *Object-Oriented Concepts, Applications and Databases*. Addison-Wesley, 1988.

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA '86 Conference Proceedings. ACM SIGPLAN Notices 21*, 11 (Nov. 1986).

[Ste87] L. Stein. Delegation Is Inheritance. In *OOPSLA '87 Conference Proceedings. ACM SIGPLAN Notices 22*, 12 (Oct. 1987).

[Ste90] G. Steele. *Common Lisp: The Language*, 2nd Edition. Digital Press, 1990.

[Str94] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Str97] B. Stroustrup. *The C++ Programming Language*, 3rd edition. Addison-Wesley, 1997.

[Str98] B. Stroustrup. Personal communication (Feb. 1998).

[Tai96] A. Taivalsaari. On the notion of inheritance. *In ACM Computing Surveys 28*, 3 (Sept. 1996).

[US87] D. Ungar, R. Smith. Self: the power of simplicity. In *OOPSLA '87 Conference Proceedings. ACM SIGPLAN Notices 22*, 12 (Oct. 1987).

[VC+97] E. Volanschi, C. Consel, G. Muller, C. Cowan. Declarative specialization of object-oriented programs. In *OOPSLA '97 Conference Proceedings. ACM SIGPLAN Notices 32*, 10 (Oct. 1997).