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

UVA Technical Report CS-98-03

John Viega, Bill Tutt, Reimer Behrends
March 3, 1998
Last revised: March 12, 1998

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. This simple feature 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.

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

John Viega
Dept. Of Computer Science
University Of Virginia
Charlottesville, VA 22903
viega@list.org

Bill Tutt
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
billtut@microsoft.com

Reimer Behrends
Dept. 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. This simple feature 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.

Automated delegation is a *subclassing* mechanism; i.e., a way by which one object derives methods and possibly variables from another class. In particular, it is a *multiple subclassing* mechanism, since it allows a class to derive methods from more than one class. A complement to multiple subclassing is *multiple subtyping*, which is the ability for a type to be an immediate refinement on multiple types. Both are often seen coupled together as a single feature, which is generally called multiple inheritance. However, this definition of multiple inheritance is not universally accepted. For example, multiple inheritance could be used to refer to only one of multiple subtyping or multiple subclassing, since the former is the ability to inherit multiple types, and the later is the ability to inherit multiple sets of methods and data. However, the term multiple inheritance is generally not used to refer to one without the other. For example, Java interfaces are a multiple subtyping mechanism, yet Java is generally considered to be lacking multiple inheritance. In the course of this discussion, we will refer to multiple inheritance as a mechanism that provides both multiple subclassing and multiple subtyping, coupled with the explicit notion of inheritance. In contrast, automated delegation is a subclassing mechanism that is separate from the inheritance mechanism of a language. Thus, while a complementary subtyping mechanism may exist, as with Java, we will not refer to the combination as multiple inheritance, since the mechanisms do not embody the language's notion of inheritance.

In section 2, we examine the arguments against multiple inheritance as traditionally found in class based programming languages, in order to understand why it was deliberately omitted from programming languages such as Java. In section 3 we discuss the advantages of multiple inheritance, which an ideal replacement would also provide. In section 4, we discuss Java's interface construct, showing why by itself it is not a wholly adequate replacement for multiple inheritance. Section 5 presents Jamie, which is a language extension to Java that demonstrates our ideas. Section 6 discusses the design decisions that we

made in adding this feature, and section 7 analyzes the advantages and shortcomings of automated delegation as a language feature.

2 Drawbacks of multiple inheritance

While multiple inheritance is a feature commonly found in many modern class-based object-oriented programming languages such as C++ and Eiffel, its appeal is not universal, as demonstrated by its exclusion from other prominent languages such as Modula-3, Objective C and Java.

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. 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.

- *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 functions 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 at 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.

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, as in Python? 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].

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 for containment, even though they are generally taught to

only use it 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.

Another significant problem with most implementations of multiple inheritance is the potential for obscure code. Take, for example, a class `A` that inherits from classes `B` and `C`, which both define a method `foo()`. When running code in `C`, 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 or obfuscate.

3 Advantages of multiple inheritance

While many people use any given subset of the arguments presented in the previous section to condemn multiple inheritance, it is still generally considered to be a useful feature in languages that provide it, and is a frequently requested feature in languages that do not. Programmers often resort to devising their own mechanisms to do what they otherwise would have used multiple inheritance to do. One of the advantages of multiple inheritance is that it obviates such ad hoc solutions that would otherwise be fairly common.

The most common strategy for simulating multiple inheritance in such a language is writing code to delegate messages to an instantiated object. Take, for example, a military simulation in Java, where we are writing a class `Tank`, which needs to inherit from class `Vehicle`. We would like to implement the `Armored` interface in class `Tank` using an instance of the `TankArmor` class, which provides implementations of all methods in the `Armored` interface, although it may or may not explicitly be declared with an implements clause. To do these things, we might write the following code:

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

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

  // Implement the Armored interface by forwarding
  // all methods in the interface 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. With the exception of resolving ambiguities (in languages that require you to do so), such work would be automated by multiple inheritance, saving the user from a repetitive chore, where he could easily make a mistake. Such mistakes become more common as the size of an interface grows.

A strategy that would not require the user to write short wrappers for every visible method in `myArmor` would be to allow clients access to `myArmor`, either through a method call, or by making `myArmor` public. However, this strategy not only breaks encapsulation, it also fails to provide the `Tank` class with the methods it needs to implement the `Armored` interface, and makes it impossible for `Tank` to selectively override any of `myArmor`'s behavior.

Another potential workaround is the *"copy and modify"* scheme [Tai96], which has nothing but downsides in that it is highly error prone, breaks encapsulation, and defeats the notion of reuse encouraged by object oriented design methodologies. Yet another alternative is to modify the original base class, which is also undesirable. First, the source code must be available. Second, editing the code can break something that works, which could even have an effect on other users and programs. Both "copy and modify" and editing the original class force the programmer to deal with the lower level implementation, and thus loses the benefit of abstraction [Tai96].

Delegation can be viewed as more useful than multiple inheritance, in that it is capable of dynamic changes that multiple inheritance generally does not support well. For example, users may dynamically change the implementation by reassigning to the

delegate. However, delegation can also be viewed as an unsuitable coping strategy, since the user must tediously write one-line methods to dispatch to a delegate. In general, languages should automate such repetitive and error-prone tasks, if it is simple to do so [Mac87].

There is no universal consensus as to what are acceptable and desirable uses for multiple inheritance. However, researchers tend to enumerate them in similar-looking but not quite identical lists (e.g., [Str94], [Mey88] 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 class inherits implementations of multiple (usually orthogonal) *mixin* classes, that may or may not be implementations of interfaces. Mixin classes are usually small units that encapsulate a general attribute or modifications that may be of interest to many other classes. Such 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. The main advantage of the mixin style of programming is that it encourages programmers to provide modular highly reusable components.
3. *Multiple subtyping*: One of the advantages 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.
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.

4 Why interfaces are not sufficient

The designers of Java chose to exclude multiple inheritance mainly due to the difficulties of implementing repeated inheritance [AG96]. However, they recognized the utility of the feature, and offered language support for interfaces, claiming that they offer all of the desirable features of multiple inheritance, without the drawbacks [GM95].

Java interfaces certainly avoid most problems generally associated with multiple inheritance, since those problems are, for the most part, the direct results of multiple subclassing, and not multiple subtyping.

However, Java interfaces only completely replace one of the four desirable uses of multiple inheritance we have enumerated, namely multiple subtyping. In addition, while Java can pair multiple interfaces to a single implementation, it may not map multiple implementations to interfaces in the same object without subclassing. The language automates neither multiple specialization nor mixin inheritance. Generally, programmers compensate by writing code to delegate to an explicitly instantiated object.

The separation of interface and implementation into two distinct mechanisms would aid in software reuse by encouraging programmers to notice and take advantage of the distinction. Inheriting a type, and no more, does not bind the programmer to any implementation, as long as his implementation meets the interface (or *signature*) specified by the type, which he may do in any way he chooses. In contrast, languages where subtyping and subclassing are part of the same mechanism usually allow but do not encourage the separation. For example, subtyping without subclassing can be achieved by inheriting classes that provide no implementation whatsoever, and subclassing without subtyping can be achieved through manual delegation. Such languages also force the inheritance of an implementation that may not be wanted in order to inherit the associated type (although the entire implementation may certainly be overridden).

For these reasons, we believe a multiple subtyping mechanism that is separate from the traditional inheritance mechanism is desirable, even if such mechanisms are not orthogonal. For example, in Java, they do not appear to be orthogonal mechanisms, since inheritance both subclasses and subtypes. However, this apparent lack of orthogonality does not interfere with the programmer's mental model, since extending a class with the `extends` keyword, and showing that you implement an interface with the `implements` keyword are both distinct and clear. In fact, the conceptual specialization relationship "IS-A" is

desirable, but separate from the notion of subclassing as a way of sharing code and data [LP91].

However, as we have previously discussed, a multiple subtyping mechanism in itself is not sufficient to replace multiple inheritance. We believe that, in order to bridge this gap, a single inheritance based object oriented language that adequately obviates multiple inheritance would provide, in addition to a multiple subtyping mechanism such as interfaces, a second subclassing mechanism, one which does not imply subtyping. We believe that such a mechanism should have the same conceptual clarity and independence as do the inheritance and interface features in Java.

Noting that without multiple inheritance people tend to use delegation when they need subclassing (which does not imply subtyping at all) we suggest that a language should automate delegation. Such a feature in conjunction with a mechanism like Java's interfaces would provide language support for all four desirable uses of multiple inheritance. In the next section, we shall present an implementation, and then in subsequent sections discuss the merits and drawbacks of our implementation, and the mechanism in general.

5 Automated delegation for Java

Jamie is a preprocessor that adds direct support for delegation to Java 1.1, which is freely available from *http://www.list.org/jamie*. 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. Jamie is implemented as a preprocessor, using the Antlr LL(k) parser generator [PQ95] and its supplied Java 1.1 grammar, and Java's reflection mechanism.

To use Jamie, the programmer writes his code in files with a ".jam" extension, which, when Jamie is run, get processed, and converted into a file with a ".java" extension. When the user runs his program, a small library supplied with the Jamie release must be visible to the Java run time system.

The mental model Jamie promotes is that it is simply automating the delegation code that a programmer would ordinarily write himself. 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 3, 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 does not have to be. For each method in the `Armored` interface, a method will be generated in the `Tank` class with the same modifiers as the `Armored` method[†]. The generated method will dispatch to the variable `myArmor` at runtime, and will closely resemble the forwarding function in the code example from section 3. In general, forwarding functions will be generated only for methods that are visible to the delegating object. For example, the delegate's protected and private methods would not be visible, but the public and default access methods would be.

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 may be a subclass of the class or interface whose methods are being delegated. For example, `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
…
```

The programmer would need to forward to `TankArmor` instead of the `Armored` interface if class `TankArmor` did not explicitly declare that it implements interface `Armored`, yet provided all of the methods found in that interface. If that were the case, the class `Tank` would still be able to declare that it implements `Armored`, depending on the programmer's preference.

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

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 delegated to 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., ints, 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;
 …
}
```

The `Tank` class can selectively prevent forwarding functions 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 it is useful in resolving name conflicts, which are errors in Jamie. For example, if `myArmor` and `myWeapons` both provided a method `stat()`, the programmer would be able to resolve the conflict in whatever manner he chooses. If a subclass of `Tank` were to provide its own implementation of `stat()`, yet `Tank` did not, a forwarding function 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. This keyword provides the same sort of ambiguity resolution virtual base classes in C++ can provide, without the potential problems associated with subclasses sharing state.

In Java, such ambiguities are actually quite common, since all classes inherit from the `Object` type. However, Jamie assumes that the programmer would prefer to delegate without delegating to methods from `Object`, so the programmer need not do so himself. We discuss the effects of this rule in section 6.

The programmer is responsible for declaring variables for each of his delegates, and assigning instances to those variables. 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. This responsibility also gives the programmer the flexibility to change the implementation of a delegate at run-time. Such an ability supports the encapsulation of logical states. For example, a class `Window` may like 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. With dynamic inheritance, we can provide 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
{
  public void draw()   { /* draw the window */ }
  public void raise()  { /* already raised  */ }
  public void iconify()
  {
    // add code to iconify the window
    ((Window)forwarder).toggleState();
  }
}

class IconifiedDisplayer implements Displayable
{
  public void draw()   { /* do nothing        */ }
  public void iconify(){ /* already iconified */ }
  public void raise()
  {
    // add code to raise the window
    ((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 iconify() { displayer = icon;  }
  public void raise()   { displayer = raised;}
  public void toggleState()
  {
    if(displayer.equals(raised)) displayer = icon;
    else displayer = raised;
  }
}
```

Note that if the programmer wants the `Window` class to implement the `Displayable` interface, then it needs to say so explicitly, even though both of its delegates implement the interface, and it is delegating to a variable of type `Displayable`. That is because delegation is a subclassing mechanism, and not a subtyping mechanism. Also 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`. The `forwarder` keyword, combined with an interface mechanism, obviates supporting uninstantiable mixins, since the delegate does not need to depend on run time information to compile method invocations. 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 issues

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 compelling 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.

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.

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 functions, which posed the problem that if an exception is raised in a delegate, the forwarding function 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, having the trace sometimes show functions 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, the only solution for delegating to static methods that is available at the preprocessor level is Java's reflection mechanism, which is not an efficient solution.

Static methods posted another significant problem to us. In Java, classes are not first class objects, which meant that there could not be regular semantics for the `forwarder` keyword. We considered returning the meta-data of a class, but encountered another limitation of the preprocessor approach. We could find no completely reliable way to find the current root class from a static method, which we would have needed to do in order to set the value of the `forwarder` keyword at delegation time.

Another important question we had to deal with was what to do about declaring visibility modifiers on the forwarding functions; 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 or default access, then copy the access modifiers of the delegated methods for the forwarding functions.

2. Have the protection level of the variable storing the delegate reflect the most lax protection level a forwarding function can achieve. For example, when delegating to a private variable `X`, all of `X`'s public and default methods would cause forwarding functions 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 functions be visible to others. Our choice was to blindly copy the access modifier of the delegate when declaring a forwarding function. 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. This may not be a wholly acceptable solution. If not, we will consider adding extensions that will allow the programmer control over the most lax access to give to a forwarding function, as can be done with inheritance in C++. However, such a mechanism would also cause an irregularity in the language, since the inheritance mechanism in Java has no such feature.

We also considered modifying the language to not show the forwarding function 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. This would work by, in the forwarding function, invoking the method being delegated to, and then forwarding to the object returned. These two ideas are likely to be future work on Jamie.

In Java, 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 the previous section, 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.

Also, this solution is consistent with other aspects of the Java language. For example, the programmer must explicitly declare every type of exception that a method can throw, except for those exceptions that are subclasses of class `RuntimeException`, such as `ClassCastException` and `NullPointerException`. The rationale for this design is that having to declare these exceptions at every method that could possibly throw them would be an extraordinarily inconvenient burden to place on the programmer [GJS96].

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, in Java, a 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.

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`?

We also 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.

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 most multiple inheritance based systems do not, such as programming with logical states. Fourth, the combination lacks most of the disadvantages of multiple inheritance:

1. The problems of repeated inheritance are obviated. 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 not part of a hierarchy, there is no way that a local call will end up calling into another delegate instead. Therefore, when looking at the code of a delegate, unexpected code paths due to inheritance are far less likely.

3. In our implementation, 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.

4. The feature is simple to implement.

Our mechanism does not directly address the problem of name clashes. However, as we mentioned previously, we personally 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.

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, specializing a prototypical instance offers an alternative to the abstract, set-theoretic inheritance that 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].

Also, delegation is a useful abstraction tool under circumstances that are not easily and cleanly handled by other language mechanisms, including inheritance. 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` as an abstract class provides the skeleton that is filled in by the methods from `Array`. As noted in section 2, 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. We observe a tradeoff between the goal of a clean design and that of reliable software. Obviously, this argument does not hold in the presence of automated delegation, where such a tradeoff does not occur. This allows both maintaining the conceptual integrity of the model and avoiding 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;
    …
}
```

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.

There are some potential disadvantages to 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, any reasonable 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 functions 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` keyword as implemented in Jamie has a potential drawback in that there is no way for the compiler to know the type of the forwarder, since arbitrary classes may delegate to arbitrary classes. Therefore, the programmer must pay the price of a run time cast. This problem could be fixed by letting the delegate specify a type that the forwarding object must have in order to use a particular interface for delegation. We plan on implementing such a feature to Jamie for future work.

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.

The first problem we believe is adequately addressed in Jamie, in that delegation is separate from the notion of type inheritance, and that it has a more explicit syntax than did C++. In C++, an object could be coerced down to a delegate through assignment or casting. However, even if a delegate declared a function as virtual that was also redefined in the delegating object, after the coercion the delegate would always be called when the function 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 Jamie, 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.

Jamie 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]. Such languages were shown to be equally as powerful conceptually as inheritance by Lynn Stein [Ste87], although she notes that, in practice, either delegation or inheritance may be more desirable. 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.

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.

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. 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].

Cecil's predicate classes are a variation on dynamic inheritance for a class-based language [Cha93]. Predicate classes offer dynamic subtyping as well as dynamic subclassing, as 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, since the mechanism is no less desirable than the ad hoc mechanisms programmers would otherwise devise themselves. 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 deep appreciation to Tim Hollebeek, Steve MacDonald and John Regehr for their insightful input when discussing this idea. We also are indebted to the people at Reliable Software Technologies, who provided useful feedback early in the life of this project.

We would also like to thank Gabe Ferrer, Leigh Caldwell, Paul Reynolds and Kathy Ryall 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*.

[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.

[GJS96] J. Gosling, B. Joy, G. Steele. *The Java™ Language Specification*. Addison-Wesley, 1996.

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

[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).

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

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

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

[PQ95] T. Parr, R. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience 25*, 7 (July 1995).

[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. In *ACM Transactions on Information Systems 7*, 2 (Apr. 1989).

[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).

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

[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).

Appendix

Changes in the Java grammar

We have made two changes to the Java grammar as found in [GJS96]. That first change is in the class declaration line. Using the conventions from [GJS96], the class declaration line is now:

ClassDeclaration:
  Modifiers$_{Opt}$ `class` *Identifier Super*$_{Opt}$ *Del*$_{Opt}$ *Interfaces*$_{Opt}$ *ClassBody*

We also add the following supporting productions:

Del:
  `forwards` *ForwardingList*

ForwardingList:
  DelegationSpec
  ForwardingList , *DelegationSpec*

DelegationSpec:
  DelegateName WithoutClause$_{Opt}$ `to` *Identifier*

DelegateName:
  Identifier
  PackageName . *Identifier*

WithoutClause:
  `without` *WithoutList*

WithoutList:
  ClassName
  WithoutList , *ClassName*

The second addition Jamie makes to the Java 1.1 grammar is to primary expressions. The `forwarder` keyword is usable wherever the `this` keyword is:

PrimaryNoNewArray:
  Literal
  `this`
  `forwarder`
  …

For the elided part of the above production, and for non-terminals not defined here, refer to [GJS96].