VICTORIA UNIVERSITY OF WELLINGTON

*Te Whare Wananga o te Upoko o te Ika a Maui*

School of Mathematical and Computing Sciences

Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5341, Fax: +64 4 495 5045
Email: Tech.Reports@mcs.vuw.ac.nz
Http://www.mcs.vuw.ac.nz/comp/Publications


# Simulating Multiple Inheritance in Java


Ewan Tempero and Robert Biddle



Technical Report CS-TR-98/1
May 22, 1998



**Abstract**

Multiple Inheritance is a feature whose worth has been vigorously debated, and even now there is no agreement as to whether a language should include it. As an example of this, Java does not have multiple inheritance. Java's designers claim that many of the benefits of multiple inheritance can be gained through the use of a new feature, the `interface`.

In this paper we investigate the claims of the Java designers. We show the relationship between interfaces and multiple inheritance, which aspects of multiple inheritance can be achieved in Java, and which cannot. In particular, we show how the reusability of the Java Core API is adversely affected by the lack of multiple inheritance, and suggest a convention for Java class library designers that will mitigate this problem.

# 1 Introduction

There has been some controversy over multiple inheritance, and whether it should be included in a language. While there appear to be benefits[Wal91], using it correctly is non-trivial[Car91], and so it is not clear that the benefits are worth the difficulties. The Java programming language supports inheritance, but not multiple inheritance. In Java terminology, a (child) class may *extend* only one other (parent) class. However, as well as supporting inheritance, Java introduces the `interface` concept, which represents a set of externally visible features of a class. A Java class may *implement* an `interface`, and it may possibly implement more than one. Arnold and Gosling[AG96] suggest the main reason for not including multiple inheritance was to avoid uncertainties that can arise in its use. They then point out that these uncertainties relate only to inheriting implementation, and do not arise in implementing multiple interfaces via the `interface` concept. They conclude:

> *Interfaces in the class hierarchy, therefore, add multiple inheritance to Java.*

In this paper, we investigate how the `interface` concept can be used to simulate multiple inheritance, and discuss the implications. We use an extended example, based on a real application, to demonstrate the simulation technique. We begin with a summary of the benefits of multiple inheritance and how they work. Section 3 introduces in detail the technique for using the `interface` concept of Java to achieve one of these benefits. We then show that, by following a particular convention, we can achieve the benefit of dealing with multiple implementations, most of the time. In section 4 we evaluate the approach we describe, and discuss some limitations. Finally, we present our conclusions.

# 2 Benefits of Inheritance

There are two important benefits commonly associated with multiple inheritance: *parent class reuse* and *interface conformance*. Parent class reuse means creating the child class by directly using the implementation of the parent class. Interface conformance means creating a class that is similar enough to another class that it can be used in place of the other class. Both of these have important, but different, advantages.

## 2.1 Parent class reuse

The first benefit of inheritance is that part of the implementation of the child class is also the implementation of the parent class. This means that classes, instead of being implemented as monolithic chunks of code, can be built up piecemeal. The advantage of this is that it allows implementors to take advantage of code written in the past, thus reducing the effort required to write new code. This form of parent class reuse is the often cited advantage of using inheritance, but what is often under-emphasised is that inheritance is not needed to reuse code.

To see this, consider the classes shown in figures 1 and 2. They give a skeleton for an implementation of a directed graph. Now suppose we want to give a visual display of a graph. To do this, we would need to add attributes such as position, colour, and size, and behaviour such as painting as appropriate to a visual depiction of the vertices and edges. Figure 3 shows how this might be done for edges without using inheritance. This implementation of `VisualEdge` has the methods expected of an `Edge` (`getFrom` and `getTo`), as well as the methods specific to edges that are to be displayed (`paint`). The `Edge` class is reused by the `VisualEdge` implementation by having an instance of it as a data field of the `VisualEdge`

```
public class Graph {
    private java.util.Vector vertices;
    private java.util.Vector edges;
    public Graph() { ... }

    public void addVertex(Vertex v) {
        vertices.addElement(v);
    }
    public void addEdge(Edge e) {
        Vertex from = e.getFrom();
        Vertex to = e.getTo();
        if (vertices.indexOf(from) < 0 ||
            vertices.indexOf(to) < 0) {
          throw new IllegalArgumentException(
                      "endpoints not already in graph");
        }
        edges.addElement(e);
    }

    public boolean isConnected() { ... }

    public void traverse(Vertex start) { ... }
}
```

Figure 1: *Skeleton of a graph class*

```
public class Vertex {
  private int x;
  private int y;

  public Vertex(int x,
              int y) {
    this.x = x;
    this.y = y;
  }

  public int x() {
    return x;
  }

  public int y() {
    return y;
  }
```

```
public class Edge {
   private Vertex from;
   private Vertex to;

   public Edge(Vertex from,
               Vertex to) {
     this.from = from;
     this.to = to;
   }

 public Vertex getFrom() {
        return from;
   }

 public Vertex getTo() {
       return to;
   }
}
```

Figure 2: *Simple vertex and edge classes*

```
public class VisualEdge {
  // Use delegation to reuse the Edge class
  private Edge edge;
  public Vertex getFrom() {
    return edge.getFrom();
  }
  public Vertex getTo() {
    return edge.getTo();
  }

  // VisualEdge specific code
  private java.awt.Color edgeColour;
  public VisualEdge(Vertex from, Vertex to) {
    edge = new Edge(from, to);
    edgeColour = java.awt.Color.black;
  }
  public void paint(java.awt.Graphics g) {
    g.setColor(edgeColour);
    Vertex from = edge.getFrom();
    Vertex to = edge.getTo();
    g.drawLine(from.x(), from.y(), to.x(), to.y());
  }
}
```

Figure 3: *VisualEdge reuses Edge by delegating to an instance of it, however the Java type checker will not recognise any relationship between the two.*

class, and, any time one of the "edge" methods of `VisualEdge` is called, that call is just forwarded on to, or *delegated* to, the `Edge` instance.

Using delegation to simulate the form of parent class reuse provided by inheritance is not new (see Rumbaugh et al.[RBP$^+$91] for example). In fact, there are languages that provide direct support for delegation[US87, Lie86]. Delegation also appears as an implementation strategy for the *Adaptor* pattern[GHJV95]. There are, however, two problems with using delegation to achieve parent reuse in Java. The first is fairly clear in the example — sometimes the amount of code needed to achieve reuse is as much as that being reused. In general however, the amount of this "wrapper code" will be much less than the amount of code being reused, and so this is not much of a problem.

The second problem is more serious. Consider the `addEdge` method of the `Graph` class in figure 1. This method expects an instance of `Edge` as an argument, but, because there is no explicit type relationship between `Edge` and `VisualEdge`, `addEdge` cannot be used with instances of `VisualEdge`. This means that we cannot reuse the `Graph` class with our new kind of edge, even though `VisualEdge` can do everything that the `Graph` class would expect of `Edge`.

Despite these problems, the delegation technique will prove useful for simulating multiple inheritance in Java, as we will show below.

```
public class VisualEdge extends Edge {
  private java.awt.Color edgeColour;
  public VisualEdge(Vertex from, Vertex to) {
    this.from = from;
    this.to = to;
    edgeColour = java.awt.Color.black;
  }
  public void paint(java.awt.Graphics g) {
    g.setColor(edgeColour);
    g.drawLine(from.x(), from.y(), to.x(), to.y());
  }
}
```

Figure 4: *Because* `VisualEdge` *inherits from* `Edge`*, the Java type checker recognises that* `VisualEdge` *conforms to* `Edge`*.*

### 2.2 Interface conformance

The second benefit that inheritance provides is that the abstract interface[1] of the child, that is, its set of externally visible features, includes the abstract interface of the parent. This means that any instance of the child class can be used where an instance of the parent class is expected, since any feature that that code using the parent class expects will be found in the child class. In such situations, we say the child *conforms* to the parent.

To see the benefits of interface conformance, consider the implementation of `VisualEdge` shown in figure 4. Because `VisualEdge` inherits from `Edge`, its abstract interface includes the methods `getFrom` and `getTo` from the `Edge` class, and so `VisualEdge` conforms to `Edge`. In fact, it is the *abstract interface* of `VisualEdge` that conforms to the *abstract interface* of `Edge`, rather than the classes themselves, but for convenience we will continue to refer to classes conforming to other classes.

The conformance of `VisualEdge` with `Edge` means that an instance of `VisualEdge` can be *safely* passed to `addEdge` in `Graph`, allowing us to reuse the `Graph` code. It is the safety that is the important feature of interface conformance. Both implementations of `VisualEdge` have the methods `getFrom` and `getTo`, but only the implementation in figure 4 *explicitly* states the interface conformance relationship between `VisualEdge` and `Edge`. While the implementation in figure 3 does have the required methods, Java's type system does not recognise that.

Note that interface conformance has no effect on whether or not `Edge` can be reused. What it does affect is the reusability of `addEdge`. That is, interface conformance allows the calling code, or *context* — `addEdge` in this case — to be reused. Thus, because inheritance indicates interface conformance, inheritance supports *context reuse*. As we discuss elsewhere, it is this increase of context reuse that is the most important benefit of inheritance[BT96].

## 3 Achieving multiple inheritance in Java

If we are to achieve the benefits of multiple inheritance in Java as described in section 2, then we must show how to reuse multiple parent classes and conform to multiple interfaces.

[1]We use the phrase "abstract interface" to mean the interface of a class, to distinguish from the Java `interface` mechanism.

### 3.1 Multiple Interface Conformance

Suppose that we are building an application that allows the user to create and simulate a finite state automaton (FSA). It is typical to depict the state transition function of an FSA as a graph, and so we can use the graph implementation discussed in section 2. However, note that a vertex in such a graph also represents a state in the FSA. While these two concepts are generally unrelated, it would be convenient to represent them both with one class. This is the kind of situation for which we need conformance to multiple interfaces, because we want to use our vertex implementation in two different contexts.

Figure 5 outlines how this part of the FSA might be implemented. The figure shows the definition of an `interface` for the concept of state. A Java `interface` provides language support for an abstract interface. A class *implements* an `interface` if its abstract interface contains at least the features described in the `interface`, that is, the class (really its abstract interface) *conforms to* the `interface`.

While Java restricts inheritance to at most one parent, it does allow a class to implement any number of interfaces, as well as extend a class. We can exploit this in our FSA implementation to achieve conformance to multiple interfaces. As the figure shows, `VisualVertex` conforms to both `VisualVertex` and `State`, albeit using two different mechanisms to indicate conformance. Thus instances of `VisualVertex` can be used by code expecting an instance of `Vertex` (such as `Graph`), *and* code expecting an instance of `State` (such as the code implementing the state transition function of the FSA).

### 3.2 Reuse of Multiple Parent Classes

We have shown that we can get the interface conformance benefit of multiple inheritance. Now we examine whether or not we can achieve the reuse of multiple parent classes.

Returning to our graph example, we now consider what needs to be done to complete our implementation of the visual versions of vertices and edges. We need to provide behaviour such as making sure our `paint` methods get called at the right times (such as when windows are uncovered), mouse events get handled properly, and so on. This functionality already exists in the Abstract Window Toolkit (AWT) framework that is part of the Java Core API. To add this functionality to our vertices and edges, their implementations must be extensions of `Component`. This is because the rest of the AWT expects classes that conform to the abstract interface of `Component`. But there is a problem achieving this with `VisualEdge`. We want it to extend `Component`, but it already extends `Edge`. Since Java forbids inheritance from multiple classes, `VisualEdge` cannot extend both.

In fact, we saw how to reuse a parent class without using inheritance in figure 3. We can reuse multiple parent classes using a combination of inheritance and delegation, but this would mean giving up interface conformance. The question then is, how to reuse more than one parent class and yet still conform to the interface of every one of them.

We know how to get conformance to multiple abstract interfaces — we get this by implementing multiple `interfaces`. Figure 6 shows how we can exploit this idea for `VisualEdge`, so that we can reuse the implementations of both `Edge` and `Component`, and yet conform to both of their interfaces, which then allows the AWT and `Graph` contexts to be reused as well. What we do is turn `Edge` into an interface, and move the original `Edge` implementation into a class, `EdgeImpl`, that implements that interface.

Any code that was expecting an instance of the original `Edge` class, `addEdge` in `Graph` for example, can now be safely passed an instance of `EdgeImpl`. The only code that will be affected by this convention of splitting interface and implementation will be code that created instances of `Edge` — now it will have to create instances of `EdgeImpl`. However, had

```java
// A very simple notion of "state"
public interface State {
    public String nameOf();
}

public class VisualVertex extends Vertex
                          implements State {
  private static final int RADIUS = 10;
  private java.awt.Point position;
  private String name;

  public VisualVertex(String name, java.awt.Point position) {
    this.name = name;
    this.position = position;
  }

  public void paint(java.awt.Graphics g) {
    g.setColor(java.awt.Color.green);
    g.fillOval(0, 0, 2 * RADIUS, 2 * RADIUS);
    g.setColor(java.awt.Color.black);
    g.drawOval(0, 0, 2 * RADIUS, 2 * RADIUS);
  }

  public boolean contains(java.awt.Point point) {
    return ((point.x()-RADIUS)*(point.x()-RADIUS) +
            (point.y()-RADIUS)*(point.y()-RADIUS))
           <= (RADIUS*RADIUS);
  }

  // Implement the State interface
  public String nameOf() {
    return name;
  }
}
```

Figure 5: `VisualVertex` *conforms to both* `Vertex` *(via inheritance) and* `State` *(by implementing the* `State` *interface).*

```
interface Edge {
  public Vertex getFrom();
  public Vertex getTo();
}

class EdgeImpl implements Edge {
  // The original implementation of Edge here
}

class VisualEdge extends java.awt.Component
                 implements Edge {
  // Implement the Edge interface using delegation
  private EdgeImpl edge;
  public Vertex getFrom() { return edge.getFrom(); }
  public Vertex getTo() { return edge.getTo(); }

  // Provide VisualEdge specific implementation
  private java.awt.Color edgeColour;
  public VisualEdge(Vertex from, Vertex to) {
    edge = new EdgeImpl(from, to);
    edgeColour = java.awt.Color.black;
  }

  // Now override the version of paint in Component
  public void paint(java.awt.Graphics g) {
    g.setColor(edgeColour);
    Vertex from = edge.getFrom();
    Vertex to = edge.getTo();
    g.drawLine(from.x(), from.y(), to.x(), to.y());
  }
}
```

Figure 6: `VisualEdge` *is reusing both* `Component` *and* `EdgeImpl` *— the former via inheritance and the latter by delegation, and also conforms to their abstract interfaces.*

```
public interface java.util.Observer {
  void update(java.util.Observable o, Object arg);
}

public class java.util.Observable {
  private java.util.Vector observers;

  public synchronized void addObserver(java.util.Observer o) {
    if (!observers.contains(o)) {
      observers.addElement(o);
    }
  }

  public void notifyObservers(Object arg) {
    for (int i = size-1; i >= 0; i--) {
      if (observers[i] != null) {
        observers[i].update(this, arg);
      }
    }
  }
}
```

Figure 7: An extract of the Java Core API for the "observer" pattern.

we followed this convention when we first implemented an edge class, such changes would not have been needed.

Now we can make `VisualEdge` conform to both `Edge` and `Component` just as we did in section 3.1. Furthermore, we can reuse the `EdgeImpl` code using the delegation technique. A similar approach can be used to implement `VisualVertex`. Thus, this "interface-delegation" combination gives the two benefits described in section 2.

It is important to note that if interface conformance is not needed, then there is no need to apply the interface-delegation technique. For example, if `VisualEdge` was only ever being used as a `Component` and never used in conjunction with `Graph`, then the `Edge` implementation could be reused just using delegation. This point is important to understanding some of the limitations of the interface-delegation technique, which we discuss next.

# 4 Limits to Interface-Delegation Technique

As shown in section 3, the interface-delegation technique can provide much of the functionality of multiple inheritance. There are some details we have not addressed that are sometimes needed for practical use of the technique. For example, protected members of the delegation object will not be available, and so a scaffolding class is needed to provide public access to those members. But in general, the interface-delegation technique is of practical benefit. There are, however, some situations for which its simulation of multiple inheritance is not accurate. We discuss these below.

## 4.1 The Java Core API

The interface-delegation approach can only be applied to classes that have an appropriate `interface`. If the programmer has control over the class and its use, then such an `interface` can be created as necessary, just as we did in figure 6. Unfortunately there are a number of classes that the programmer is likely to want to use, but has no control over — those in class libraries such as the Java Core API.

Consider again the example of the visual depiction of a graph. We would like to allow the user to manipulate the graph. In particular we would like to allow the user to move the vertices around, in which case we would expect the edges attached to the vertex to follow the vertex as it is moved. There is a standard design for this situation, the so-called *Observer* pattern[GHJV95]. The edge is registered as an observer of the vertex, and when the vertex changes state (such as its position), it notifies all of its observers, which then have the opportunity to react to the change in state. The Java Core API provides a framework for this pattern, an extract of which is shown in figure 7. In this case, the edge should implement the `Observer` interface, and the vertex must extend the class `Observable`.

But now we have a problem. `VisualVertex` must extend both `Component` *and* `Observable`. We can achieve parent reuse for them both by, for example, using inheritance for `Component`, and delegation for `Observable`, but we cannot produce a class that will conform to both classes. As we observed at the end of section 3, if we don't need conformance, then delegation will be sufficient. But in this case we are dealing with a class and interface that depend on each other — any implementation of `update` provided to implement the `Observer` interface expects an argument that conforms to `Observable`. If we used delegation for `Observable`, then our class would not conform to `Observable`, and so could not be passed to `update`. Exchanging the uses of inheritance and delegation does not help either: if `VisualVertex` inherited from `Observable` and delegated to `Component`, then it would not conform to `Component` and so could not be used in the AWT framework.

Making up our own interfaces for these classes will also not solve our problem. If we did so for `Component` for example, and had `VisualVertex` implement that interface, then `VisualVertex` would conform to that interface, and *not* to `Component`, and so could not reuse any of the AWT classes that expect something conforming to `Component`.

In this particular case, the observer pattern is simple enough that we can fairly easily create our own implementation of it. But this means we are writing code to implement functionality we already have available. Thus, the "reusable classes" of the Java Core API are not as reusable as we would like. Furthermore, as the Java Core API gets even larger, and as other suppliers start providing their own class libraries, it is more likely that we will want to reuse multiple classes from them as parent classes.

Nevertheless, it did not have to be this way. If at least one of the classes `Observable` or `Component` implemented an interface that was used in the Java Core API, then we could have used the interface-delegation approach, as we did in figure 6. Thus, if class library suppliers follow the convention of providing interfaces for the classes they provide, and express any frameworks they create in terms of the interfaces rather than the classes, then this would allow all classes to be available for multiple inheritance using the interface-delegation approach.

## 4.2 Dealing with Ambiguity

Multiple inheritance causes ambiguity because it raises the possibility of different definitions of a name being applicable to a use of the name, since two of the parents may have a method with the same name but different meanings. For example, suppose that our original `Edge` class provided an `isValid` method that returns a boolean value indicating some property about edges. If Java had multiple inheritance and we created a `VisualEdge` class by inheriting

```
public abstract class java.awt.Container
                      extends java.awt.Component {
  // ...
  public java.awt.Component add(java.awt.Component comp) {
      addImpl(comp, null, -1);   // self call
      return comp;
  }
  // ...
  protected void addImpl(java.awt.Component comp,
                         Object constraints,
                         int index) {
    // implementation here
  }
}
```

Figure 8: *Self calls in* `Container`*. Because* `add` *makes a self call to* `addImpl`*, any class that wants to override* `addImpl` *must take care to ensure the appropriate version of* `addImpl` *is called.*

from both `Edge` and `Component` then we would have a conflict, because `Component` also has an `isValid` method. Thus, any calls to `isValid` on instances of `VisualVertex` would be ambiguous.

Ambiguity does not arise with simulating multiple inheritance using the interface-delegation technique, since, as specified by Java, implementation can only be inherited from one place. In the case of `isValid` we will only inherit one of the available versions, in this case the one coming from `Component`. The `Edge` version of `isValid` will only become available if the programmer explicitly provides access to it through a delegation wrapper, in which case the wrapper method will override the version inherited from `Component`. Thus ambiguity is not possible.

While there cannot be ambiguity, it will not be possible to provide both versions of `isValid` to `VisualEdge`. If the application has some contexts that expect the behaviour of one version, and other contexts that expect the behaviour of the other version, then the multiple inheritance provided by the interface-delegation technique will be of no direct use. This is not just a problem with this technique. It also arises with classes that implement multiple interfaces, where the same method name appears in more than one of the interfaces, albeit with different expected semantics.

### 4.3 Self calls

As well as parent class reuse and interface conformance, inheritance can provide quite flexible behaviour through the use of *self calls*, that is, when a method in the class calls another method in same class. For example, consider the code fragment shown in figure 8. It shows part of the implementation of `Container` in which a method, `add`, makes a self call to another method, `addImpl`.

When `add` in `Container` executes, it is not necessarily the case that it is the `addImpl` in `Container` that will be called. For example, consider the classes shown in figure 9. The class `InheritsContainer` uses inheritance to reuse `Container`, and so inherits the method `add` from `Container`. Because `InheritsContainer` overrides `addImpl`, the dynamic nature of method calls means that any calls to the `add` method on instances of `InheritsContainer`

```
// Reuse via inheritance
public class InheritsContainer extends java.awt.Container {
  // ...
  // Override addImpl
  protected void addImpl(java.awt.Component comp,
                         Object constraints,
                         int index) {
    // new implementation here
  }
  // ...
}
```

Figure 9: Using inheritance to achieve parent class reuse of `Container` results in the correct self call behaviour when `addImpl` is overridden.

will result in the `addImpl` method in `InheritsContainer` being called. This is because a self call is really a call to a method in the class of the instance referenced by `this`. The consequence is that when such a self call is made, the method lookup begins with the class of the object, not the class containing the method actually executing. The use of self calls provides a great deal of flexibility in the implementation of classes, because it allows child classes to customise the behaviour of their parents. This behaviour illustrates a distinguishing characteristic of object-oriented programming.

Providing parent class reuse through delegation is complicated in the presence of self calls. To see this, consider the class in figure 10. This class uses delegation to reuse `Container`. Observe that when using the delegation technique, any call to `add` on an instance of `BadSelfContainer` will be explicitly passed to the local `Container` instance. This means that when the `Container` version of `add` is called, `this` will be referencing the `Container` instance, *not* the instance of `BadSelfContainer`. This will result in incorrect behaviour, namely the `Container` version of `addImpl` being called, and not the `BadSelfContainer` version.

A related problem occurs if the class being reused as a parent class passes `this` as a parameter to method calls on another object. If the other object calls a method on the value passed to it, then just as happens in the case of `BadSelfContainer`, the wrong object will receive the call.

The correct behaviour can be achieved, as seen in figure 11. This is done basically by copying the implementation of the method that normally would be delegated to. Since this requires duplicating code, it is not an ideal solution, but at least it can be done. And if the use of self calls is isolated to small methods, then the cost can be kept to a reasonable level.

The behaviour due to self calls becomes more complicated when multiple inheritance is used. Consider figure 12, which assumes that Java does have multiple inheritance and gives an example of so-called "diamond" inheritance, where a class has the same parent through two different routes in the hierarchy. Notice that the version of `addressLabel` belonging to `Employee` makes a self call to `formatName`. Now consider what happens when the following code is executed.

```
Tutor aTutor = new Tutor();
String label = aTutor.addressLabel();
```

There is ambiguity when the method `addressLabel` is called on instances of `Tutor`, since this class inherits `addressLabel` from two places. Languages that provide multiple

```
// This results in incorrect self call behaviour when reusing
// via delegation.
// Assumes java.awt.Container was renamed to ContainerImpl,
// and java.awt.Container becomes an appropriate interface.

public class BadSelfContainer implements java.awt.Container {
  private java.awt.ContainerImpl containerPart;

  // BadSelfContainer version of addImpl
  protected void addImpl(java.awt.Component comp,
                         Object constraints,
                         int index) {
    // new implementation
  }

  // delegate add
  public java.awt.Component add(java.awt.Component comp) {
    return containerPart.add(comp);
  }

  // ...
}
```

Figure 10: The standard application of the interface-delegation technique in the presence of self calls produces the incorrect behaviour.

```
// Correct self call behaviour when reusing via delegation.
// Assumes java.awt.Container was renamed as before.
public class DelegatesContainer implements java.awt.Container {
  private java.awt.Container containerPart;

  // DelegatesContainer version of addImpl
  protected void addImpl(java.awt.Component comp,
                         Object constraints,
                         int index) {
    // new implementation here
  }

  // Recreate add implementation to ensure self call goes
  // to the right place
  public java.awt.Component add(java.awt.Component comp) {
      addImpl(comp, null, -1);   // self call
      return comp;
  }
  // ...
}
```

Figure 11: `DelegatesContainer` shows what we have to do to get the correct self call behaviour.

inheritance take different approaches to dealing with this kind of ambiguity. For example, if there is a name that has different definitions by different paths (`addressLabel` in this case), then the "most specific" definition is taken, where the meaning of "most specific" might be different in different languages. One possibility might be that if one definition overrides another definition along any path, then the first definition is "more specific" than the other. In the example in figure 12, the definition of `addressLabel` in `Employee` overrides that in `Person`, and so it would be the most specific definition. This is the `C++` behaviour.

If we assume the `C++` behaviour, then it is the version of `addressLabel` in `Employee` that is called. When the self call in that version is made to `formatName` is made, the method lookup begins in `Tutor`. The version of `formatName` that would then be called is the most specific one available from `Tutor` — in this case, the version in `Student`!

That a method in one class can call a method in a unrelated class in this manner can at best be described as non-intuitive. In fact, it is this weird "yo-yo" behaviour, and not so much the introduction of ambiguity (which should always be able to be detected by a compiler) that causes some program language researchers to be suspicious of multiple inheritance. The example presented here is a variation of that discussed by Cargill[Car91].

Our interest here is whether that "yo-yo" behaviour can be simulated with our interface-delegation technique. In the case of the particular example shown in figure 12, it can, but only if `Tutor` inherits from `Student`, since this is the only way that the dynamic binding of `this` will allow `formatName` to be called. In order for this to work, `addressLabel`, which must be delegated, must be a copy of the implementation of `Employee.addressLabel`. On the other hand, if `Tutor` delegates to `Student`, then the correct behaviour is not possible. Thus, if the combination of self calls requires that the child class must inherit from more than one class, then it will not be possible to achieve the canonical behaviour.

```
class Person {
  private String name;
  public String nameOf() {
    return name;
  }
  protected String formatName() {
    return nameOf();
  }
  public String addressLabel() {
    String temp = formatName();
    // complete the label (omitted)
  }
}


class Student extends Person {
  protected String formatName() {
    // Modified to add student's ID
  }
}

class Employee extends Person {
  public String addressLabel() {
    // Modified to use the employee's
    // department
    String temp = formatName();
    // complete the label (omitted)
  }
}

class Tutor extends Employee, Student { // Not legal Java
  // implementation omitted
}
```

Figure 12: *The so-called "diamond" multiple inheritance structure: both* `Employee` *and* `Student` *extends* `Person`*, and then* `Tutor` *extends both of them, thus giving it the same parent by two different paths.*

It is unlikely that this kind of behaviour will be required often enough that not being able to provide it will be a serious problem. Nevertheless, it is important to understand the limitations of the interface-delegation technique.

## 5 Conclusions

Java does not support multiple inheritance, but introduces the `interface` feature that does support something similar. It has been suggested that this can be used to accomplish the desirable effects of multiple inheritance. In this paper, we determined which effects are achievable using just interfaces alone. We then introduced the *interface-delegation* technique for simulating multiple inheritance in Java. Essentially, this technique uses the `interface` feature to allow conformance to multiple parent class types, and uses delegation to allow reuse of the parent class implementations.

After presenting the interface-delegation approach, we evaluated our approach. Because we do not require inheritance of implementation, our approach avoids the usual difficulties of multiple inheritance. However, our approach does have limitations: in particular we have discussed how self calls cannot be handled very well when using delegation.

Most importantly, we explained that the interface-delegation approach can indeed be used to gain the main potential code reuse benefits of multiple inheritance, albeit with slightly more work. To facilitate such reuse, however, class developers must follow the convention of providing an `interface` for each class of interest. We believe this is an important consideration for designers of Java class libraries. It is especially important for the development of frameworks. The interdependence of the classes and interfaces that make up a framework mean that interface conformance is necessary, and the only way to ensure that interface conformance is possible for user-defined classes is if all such interdependencies are expressed solely in terms of interfaces. Our main conclusion therefore, is:

> To facilitate code reuse in Java, every class intended for reuse by inheritance should also have a matching `interface` that is used in place of the class wherever possible.

### Acknowledgements

Some of the work presented here was done while Tempero was visiting visiting the Pacific Software Research Center at the Oregon Graduate Institute of Science and Technology. He gratefully acknowledges the support of the Center.

## References

[AG96] Ken Arnold and James Gosling. *The Java Programming Language.* The Java Series. Addison-Wesley, 1996.

[BT96] Robert Biddle and Ewan Tempero. Understanding the impact of language features on reusability. In Murali Sitaraman, editor, *Proceedings of the Fourth International Conference on Software Reuse.* IEEE Computer Society Press, April 1996.

[Car91] Tom A. Cargill. The case against multiple inheritance in C++. *USENIX Computing Systems*, 4(1):69–82, Winter 1991.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[Lie86] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. *SIGPLAN Notices*, 21(11), November 1986.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *1987 Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 227–242, 1987. Published as ACM SIGPLAN Notices 22(12), December 1987.

[Wal91] Jim Waldo. Controversy: The case for multiple inheritance in C++. *USENIX Computing Systems*, 4(2):157–172, Spring 1991.