

# Simulating Multiple Inheritance and Generics in Java

Krishnaprasad Thirunarayan  
Dept. of Computer Science and Engr.  
Wright State Univ., Dayton, OH-45435.  
(email: tkprasad@cs.wright.edu)

Günter Kniesel  
Computer Science Dept. III  
University of Bonn, D-53117 Bonn  
(email: gk@cs.uni-bonn.de)

Haripriyan Hampapuram  
Intrinsa Corporation  
Mountain View, CA-94041.  
(email: hhampa@microsoft.com)

## Abstract

This paper presents Java language from an object-oriented software construction perspective. It explains the implications of banning generics and multiple inheritance of classes, and explores the patterns and the idioms used by the Java designers and programmers to redeem their benefits. The paper also discusses an alternative to multiple inheritance, as incorporated in Lava, which extends Java with constructs for type-safe automatic forwarding.

**Keywords:** Java Language, Object-Oriented Programming, Design Patterns, Multiple Inheritance, Generics, Delegation.

## 1 Introduction

Design patterns are a description of communicating objects and classes that are customized to solve a general design problem in a particular context [1, 2, 3]. Design patterns consolidate design experience that can potentially be reused in tackling a variety of problems, and in organizing and analyzing potential solutions to the problem. Many of these patterns improve software reuse and facilitate code evolution.

Idioms, on the other hand, are low-level “patterns” that describe how to solve implementation-specific problems in a programming language. Idioms demonstrate competent use of programming language features [2].

Java does not support generics and multiple inheritance of classes, thereby simplifying the language and its implementation. This paper discusses the implications of Java’s lack of generics and multiple inheritance of classes on software construction, and presents techniques to mitigate its effect.

Section 2 reviews problems associated with support for multiple inheritance in existing object-oriented languages. Section 3 analyzes idiomatic approximations to multiple inheritance in Java using only single inheritance of classes and multiple inheritance of interfaces. It also rationalizes certain aspects of Java in terms of design patterns. Section 4 discusses possible language support for multiple inheritance. It shows that asking for multiple inheritance in Java is not a good idea given that more powerful, dynamic alternatives are available. In particular, support for *delegation* would be preferable to adding multiple inheritance.

Section 5 analyzes idiomatic approximations to generics using reference types and reflection, and discusses their limitations *vis a vis* an explicit support for generics. Section 6 concludes the paper.

## 2 Background : Multiple Inheritance

Object-oriented languages such as Eiffel, C++, etc support multiple inheritance of classes, while Smalltalk, Oberon, Modula-3, Ada-95, etc support only single inheritance of classes [4]. Furthermore, Eiffel, C++, Ada-95, etc support generics/templates to promote reuse. In contrast, Java supports neither multiple inheritance of classes nor generics, explicitly. This simplifies the language and its implementation but “burdens” the programmers and the Java API designers.

There is no consensus among researchers on the semantics of multiple inheritance in the presence of method overriding and potential conflicts due to multiple definitions. (Bertrand Meyer [5] provides a lucid account of the key issues and insights for incorporating multiple inheritance in an object-oriented language. Similar issues have also been studied by Artificial Intelligence and Object-Oriented Databases communities [6].) The programming language designers have taken at least three different approaches, exemplified by the design of C++, Eiffel and Java.

**C++.** C++ views multiple inheritance as *replicated* single inheritance. That is, a child class inherits every field/method of every parent class, unless the field/method is overridden by the child. In case more than one parent defines the same field/method, the child uses the scope-resolution operator to disambiguate the reference. The problem of *repeated inheritance* [5] of a field/method arises when the parents share a common ancestor. In C++, the fields in a child instance due to a common ancestor are either all *shared* (virtual) or all *duplicated*. This approach has two serious drawbacks in practice: (1) the “all or none” granularity in choosing to share or to duplicate fields of a common ancestor, and (2) the place where this choice is made [4, 7]. Another consequence of the C++ decision is that several natural implementations, such as the code for the `Tree` data structure using *leftmost-child-right-sibling* form given below, are deemed illegal.

```
class Cell : public Link { ... };
class Tree : public Cell, public Link { ... };
```

However, the corrected version, given in Chapter 13 of [8], requires unreasonable amount of code duplication that an OOPL is fighting hard to minimize.

```
class Link { ...code... };
class Cell { ...ditto + new code... };
class Tree : public Cell, public Link { ... };
```

To paraphrase S. Tucker Taft’s posting to `comp.lang.ada`: The C++ mechanism for multiple inheritance has the worst of both the worlds: a language made more complex by the addition of linguistic multiple inheritance, and an application made more difficult by the lack of appropriate building blocks for solving the problem the “right” way.

**Eiffel.** In contrast, Eiffel provides a rich set of primitives to program-in an application-specific strategy for multiple inheritance [5]. In particular, it supports:

**redefinition:** to override a (potentially inheritable) field/method definition.

**renaming:** to remove name clashes making available multiple definitions under different names.

**undefinition:** to “join” names and to make unavailable certain definitions.

**selection:** to pick the code that must be run on a subclass instance referenced by an entity of a “repeated” ancestor type using dynamic binding in case of an ambiguity.

Whereas the flexibility and the expressiveness that Eiffel’s design of multiple inheritance endows is widely recognized, it is an open debate whether it commensurates with the additional complexity introduced in the language and its implementation.

Furthermore, Eiffel’s **select** mechanism exhibits a subtle semantic problem of its own. Because it *always* selects the same code, irrespective of the context of the method invocation, it reinstalls the problem that method renaming was intended to solve: the selected method is executed even in contexts where a method with the same signature but different semantics is expected. A detailed discussion is beyond the scope of this paper and can be found in [9].

**Java.** The Java designers have chosen to eliminate multiple inheritance of classes. Instead, the programmer is now expected to realize multiple inheritance by “simulating” it in the language. This decision is motivated by the problems sketched above and the additional desire for binary compatibility: Recall that in Java, the computation of numeric offsets for instance fields in memory is done by the run-time system. This permits updating a class with new instance fields or methods without affecting existing code. In other words, it is possible to run an existing bytecode file for a subclass after updating and recompiling the class, but without recompiling the subclass. This property would be lost if multiple inheritance were supported because multiple inheritance conflicts could suddenly arise during class loading.

Note also that, on older processors, single inheritance of classes had performance advantages over multiple inheritance because the former simplified the layout of instance fields and enabled generation of efficient code for dynamic dispatching of methods [10]. However, modern pipelined processors execute the additional offset adjustment instruction during the delay time required for branching to the method’s code [11]. So, surprisingly, multiple inheritance overheads are negligible even in the presence of polymorphic variables.

We now explore the *extent* to which the reusability benefits of multiple inheritance and generics can be reinstated in Java through specific patterns and idioms.

### 3 Approximating Multiple Inheritance in Java

What does support for multiple inheritance accomplish for the following example, which is illegal in Java?

```
class A { ...
    public String a() { return a1(); }
```

```

    protected String a1() { return "A"; }
}

class B { ...
    public String b() { return b1(); }
    protected String b1() { return "B"; }
}

class C extends A, B { ...
    protected String a1() { return "C"; }
    protected String b1() { return "C"; }
}

```

**Code Reuse:** The fields defined in classes A and B are present in an instance of class C, and a method defined in class A (resp. B) can potentially be run on an instance of class C.

**Polymorphism:** An instance of class C can potentially be used wherever an instance of class A (resp. B) is required.

**Overriding:** Inherited methods can be *adapted* to the needs of class C by *overriding* methods on which they rely. For instance, the definition of method a1() and b1() in C has the indirect effect that the inherited methods a() and b() now return "C".

**Modification:** Any changes to classes A and B are propagated automatically to class C.

Often, multiple inheritance is (mis)used for just one of the above effects, for example, just for code reuse or for achieving multiple subtyping. It is, therefore, important to understand how each effect can be simulated in Java and, even more, to assess which effects are relevant to a particular design.

In the following, we will first present the general scheme for simulating multiple inheritance and identify the ingredients of the simulation that achieve each of the above effects. Then we will present different examples that illustrate how these ingredients can be used in various combinations. The examples will develop from cases that do not really require multiple inheritance (and are indeed better solved without), via cases where multiple inheritance would be appropriate (and can be approximated in Java), to cases where more dynamic techniques than multiple inheritance are required. We show that in the latter cases our simulation is preferable to multiple inheritance but it still cannot provide the full power that can be achieved by suitable language support. We conclude this section by contrasting our simulation with an extension of Java that supports *multiple object-based inheritance*.

### 3.1 General scheme

#### Simple forwarding

In a first attempt, the example above can be approximated in Java as follows (changes are underlined):

```

class A {
    public String a() { return a1(); }
}

```

```

    protected String a1() { return "A"; }
}

interface IB {
    public String b();
}

class B implements IB {
    public String b() { return b1(); }
    protected String b1() { return "B"; }
}

class C extends A implements IB {
    B b; // aggregation
    public String b() { return b.b(); } // forwarding
    protected String b1() { return "C"; }
    protected String a1() { return "C"; }
}

```

Class C inherits from class A and supports the same interface IB as class B. Furthermore, class C forwards every invocation of a method in interface IB to the corresponding field of type B.

The generalization of this translation to “class C extends A, B1, ..., Bn {...}” is straightforward given that Java supports implementation of multiple interfaces. We call the “simulated superclasses” Bi the *parent classes* of C, in contrast to “genuine superclasses” like A. Similarly, C is called *child class* of each Bi.

How well does this translation capture multiple inheritance?

**Code Reuse:** Class C reuses the code for classes A and B but does require glue code to implement forwarding.

**Polymorphism:** An instance of class C can be used for an instance of class A and for an instance<sup>1</sup> of interface IB but *not* for an instance of class B. The latter is because, in general, implementing the common interface IB *does not imply* that class C supports all the methods in class B.

If interface IB contains all the public methods of class B, and IB is used instead of B for type declarations throughout the program, then the ability to substitute C instances for IB instances is sufficient. So, whether the translation supports polymorphism depends on anticipation of future extensions (the interface and the corresponding implements relation in the parent class must be declared), good documentation (of the intention that the interface should be used for typing instead of the parent class) and programmer discipline (you must adhere to the documentation).

**Modification:** Any changes to class A are automatically propagated to class C. Similarly, adding a new field to class B or changing a method body in class B is transparent to class C. However, the addition or deletion of a method from class B or changes to a

---

<sup>1</sup>By an instance of an interface we mean an object that is an instance of a class that implements the interface.

method signature in class B requires changes to interface IB and to class C (to install the glue code for forwarding).

**Overriding:** Overriding and the related adaptation of “inherited” code from aggregated objects is *not* possible. For instance, the definition of method `b1()` in C does *not* influence the result of the method `b()`. An invocation of `a()` on an instance of class C returns "C" but an invocation of `b()` on an instance of class C still returns "B".

Overriding is the core concept of inheritance [12, 13]. Without overriding, existing class hierarchies would dramatically change their semantics and many essential concepts and techniques (for example, protected methods, abstract classes, template methods [1]) would not exist. Because it disables overriding, the above translation cannot be regarded as a satisfactory approximation of (either single or multiple) inheritance.

### Overriding via back-references

Achieving the effect of overriding is also possible on the basis of aggregation and forwarding, provided that the parent classes are available for modification [14, 15]. The essential idea is to give the parent object a *back-reference* to the object that initially forwarded the message. Messages otherwise sent to `this` by the parent are now sent via the back-reference, enabling execution of overriding methods from the child.

The back-reference can either be stored in the parent object or it can be passed dynamically as an additional argument of forwarded messages. The first approach is called the *stored pointer model*, the second one is called the *passed pointer model* [14]. A detailed discussion and usability evaluation of the passed pointer model and the stored pointer model can be found in [15]. Applying the passed pointer model to our example yields:

```
class A { ... }; // like above

interface IB {
    public String b(IB self) ; // modified signature
    public String b1() ; // new public method
}

class B implements IB {
    public String b(IB self) { return self.b1(); }
    public String b1() { return "B"; }
}

class C extends A implements IB {
    B b;
    public String b(IB self) { return b.b(this); }
    public String b1() { return "C"; };
    protected String a1() { return "C"; };
}
```

Now invocations of method `a()` and `b()` on an instance of class C both return the string "C". Note that we had to add the method `b1()` to interface IB and relax the visibility status of method `b1()` from `protected` to `public`, thus generally exposing details that should be known only to subclasses / child classes.

Meta-level techniques can also be used for simulating multiple inheritance and delegation [16]. However, these exhibit shortcomings with respect to type-checking and performance. Therefore we have confined ourselves to non-reflective approaches.

### Conflicts prevent polymorphism

If several (base) classes define methods that share the same signature, the translation sketched in this section permits just one method with that signature in the child class. This is a problem if the different base class methods have different semantics *and* child class instances are substitutable for base class instances: then a wrong method that happens to share the name and signature can be invoked. This problem is illustrated in [17, page 273-275] with the following example (adapted here to pseudo-Java syntax):

```
class Lottery {
    // ...
    int draw() {...};
}

class GraphicalObject {
    // ...
    void draw() {...};
}

class LotterySimulation extends Lottery, GraphicalObject {
    // ... what does draw() mean here? ...
}
```

In such a case we need *two* methods in the `LotterySimulation` class, say `drawLottery()` and `drawGraphical()`. Without language support for multiple inheritance, however, dynamic binding does not know that it should select the `drawLottery()` method for an invocation of `draw()` on a variable of type `Lottery` and the `drawGraphical()` method for an invocation of `draw()` on a variable of type `GraphicalObject`.

Implementation of such context-dependent behavior without language support results in convoluted, impractical designs. This is problematic because it leaves us only with the choice to give up substitutability of a child class instance for a parent class instance (by not declaring the `implements` relation). As a consequence, child classes cannot override methods in the respective parent classes because the simulation of overriding depends on substitutability of child instances for parent class instances:

```
// class Lottery and GraphicalObject like above

class LotterySimulation
    extends Lottery {          // NOT substitutable for GraphicalObject
    GraphicalObject go;
    ...
    public int  draw()          {...}; // overrides draw() from Lottery
    public void drawGraphical() {...}; // does NOT override
                                        // draw() from GraphicalObject
}
```

An example showing that much better results can be achieved with suitable language support is shown in Section 4.2.

### 3.2 Illustrative Examples

To sum up, multiple inheritance can be simulated by joint use of

- *forwarding* as a means to achieve code reuse,
- *interfaces* as a means to achieve polymorphism, and
- *back-references* as a means to approximate overriding.

The applicability of the latter two techniques is inhibited, however, by the need to provide suitable “hooks” in parent classes. Furthermore, conflicting methods in parent classes prevent simulation of polymorphism, even if the parent classes provide the required hooks.

Whereas the above limitations of the available simulation techniques prevent a full simulation of multiple inheritance in cases where it would be useful, there are also other cases where a full simulation is not required and a design without multiple inheritance is preferable. We now present some practical applications of the above translation in Java, which illustrate different facets of both situations.

On the one hand, the examples review some widely-used combinations of the three simulation techniques discussed above, showing that a rigid implementation of the presented translation is often neither appropriate nor possible. The right choice is driven by the needs of a particular application and constrained by the limitations of available *built-in* or *third-party supplied* classes that cannot be modified. On the other hand, they illustrate a number of specialized patterns derived from the above translation. Anticipating the need for customization, vendors can apply such patterns in designing utility classes and frameworks, to facilitate their use in multiple inheritance contexts.

**Code reuse without polymorphism.** One application of multiple inheritance is to provide an implementation of an abstraction, using a concrete class. This has been called “the marriage of convenience” in [5]. In C++, *private inheritance* is used for a similar purpose. In both cases the inheriting class is not intended to be a subtype of the concrete class, it just reuses its code to implement the interface of the abstraction. In Java, this corresponds to a child class that forwards to the concrete class and inherits from an abstract class (or implements an interface) that represents the abstraction. An example would be an abstract class `Stack`<sup>2</sup> that can be implemented using different underlying concrete data types, for example, an `Array` or a `LinkedList`.

**Code reuse and polymorphism.** In Graphical User Interfaces, “containers” such as windows, frames, panels, etc hold primitive “components” such as buttons, check-boxes, text-fields, etc. In many applications, a group of such elements exhibits similar behavior as the individual element or is controlled similarly as the individual element. Java supports such

---

<sup>2</sup>Note that this example does not reflect the current design of the Java APIs, where `Stack` is a *class* with *one* fixed implementation as a subclass of `Vector`. This includes all `Vector` operations in the interface of `Stack`, enabling to bring a `Stack` into an inconsistent state, for example, by directly calling the method `add(atIndex, elem)`.

recursive nestings by letting `java.awt.Container` inherit from `java.awt.Component`. In fact this is an instance of the well-known *composite pattern* [1] that appears in various other contexts such as in the implementation of:

- *Command macros*, where a sequence of commands is treated as a command.
- *Composite figures*, where a set of figures is treated as a figure.
- *Nested menus*, where a menu can be a menu-item.

The essence of the composite pattern is that composite elements behave at the same time like elements *and* like collections of elements. Thus they should be subtypes of both, `Element` and `Collection`. A composite element can ideally be defined using multiple inheritance as follows [5]:

```
class CompositeElement extends Collection, Element { ... }  
  
class CompositeElement extends Collection[Element], Element { ... }
```

In the latter case, `Collection` is a template/generic (see Section 5 for a discussion on generics in Java).

In Java, the composite pattern can be coded using single inheritance of classes by:

1. defining an interface to specify the methods required of each element (whether primitive or composite),
2. defining a class for the composite elements that (a) implements the interface, and (b) extends an existing collection class, and
3. requiring a client to manipulate the elements via the methods listed in the interface.

```
interface Element { ... }  
class BasicElement implements Element { ... }  
class CompositeElement extends <SomeImplementationOfCollection>  
    implements Element { ... }
```

**Code reuse, selective overriding, no polymorphism.** Ideally, the well-known `Clock` applet class ought to multiply inherit from built-in classes `java.applet.Applet` and `java.lang.Thread`. That is,

```
class Clock extends Applet, Thread { ... };
```

This declaration, which is illegal in Java, enables the clock to run in a browser, and update itself periodically on a separate thread. To facilitate implementation of such classes by single inheritance, Java defines an interface

```
interface Runnable { public void run(); }
```

and a `Thread` constructor of the form

```
Thread (Runnable r) { ... }.
```

Now class `Clock` can be coded in Java as:

```

class Clock extends Applet implements Runnable {
    Thread t = new Thread (this);          ...
    public void run { ... };              ...

    // connecting methods of Applet and Thread
    public void start() {
        t.start();                        ...
    }
}

```

The class `Thread` exemplifies an implementation of overriding via stored back-references that are passed to the parent object in its constructor.

This pattern of related interfaces, classes, and constructors enables code reuse and customization. Anticipating the need for customization, vendors can apply it in designing utility classes and frameworks, to facilitate their use in multiple inheritance contexts.

The pattern is appropriate in cases where the ability to customize inherited methods (for example, `start()`) by overriding one or a few methods on which they rely (for example, `run()`) is required, but the ability to substitute child instances for parent instances (for example, `Applets` for `Threads`) is not relevant or impossible to implement.

But why didn't the designers of Java define a more complex interface than `Runnable`? It would have created the opportunity for *conflicts*. For instance, the `start()` method is defined in both classes — `Applet` and `Thread` — and with different semantics, while class `Clock` can support only one definition of `start()`. As explained in Section 3.1 such conflicts prohibit using `Clocks` as `Threads`. So a larger interface than `Runnable` would not have been beneficial anyway but would have added the risk of subtle errors.

**Marker interfaces.** In object-oriented languages that support multiple inheritance, general purpose facilities such as storing and retrieving objects from persistent storage, cloning objects, etc can be defined as library classes and used as ancestors by classes needing such facilities. For instance, in Eiffel, the library class `Storable` defines I/O methods for objects: `store(file:FILE) {...}`, `Storable retrieved(file:FILE) {...}`, etc. In Java, such functionality can be achieved in many cases without taking recourse to multiple inheritance, as discussed below.

A *marker interface* is an interface that declares no methods or fields. It is useful when we need to determine something about the objects without assuming that they are instances of any particular class. (See [18] for examples.)

In Java, certain general purpose methods are housed in the root class `java.lang.Object` and made available only when a subclass implements a suitable marker interface. For instance, the `clone()`-method is defined in class `java.lang.Object` as `protected` and is inherited by all classes. However invoking it on an arbitrary object can cause `CloneNotSupportedException` to be thrown. To run the `clone()`-method on an instance of a class, the class *must* implement the marker interface `Cloneable` and redefine `clone()`.

Java classes such as `java.io.ObjectInputStream` and `java.io.ObjectOutputStream` support `readObject(java.lang.Object)` and `writeObject(java.lang.Object)` respectively to read from / write to a persistent storage. Java also requires a class to implement the marker interface `Serializable` before it can participate in these methods. Observe that the signa-

tures of these methods are “inverted” when compared with the corresponding methods of class `Storable` in Eiffel.

Cloning and storing objects are the only examples that cannot be regarded as instances of our translation. The cloning mechanism is hardwired in a native method of class `Object`; the serialization mechanism makes heavy use of reflection in order to determine whether the implementor of the `Serializable` interface provides its own private `writeObject(java.io.ObjectOutputStream)` method. In fact both make better examples of functionality that cannot be modeled in a way that is *recommendable* as an idiom.

**Static versus dynamic composition.** The lack of multiple inheritance does prohibit certain reasonable static combinations of built-in classes in Java. For instance, one cannot define a class of objects that can be used both as a `java.io.LineNumberReader` and as a `java.io.InputStreamReader`, or as a `java.io.LineNumberReader` and as a `java.io.PushbackReader`.

Instead, Java just provides constructors to turn any `Reader` instance into an instance of `LineNumberReader`, `InputStreamReader` or `PushbackReader` by applying the decorator pattern [19]. Replacing anticipated, static composition of different abstractions via multiple inheritance by demand-driven, dynamic composition via aggregation is a gain of functionality, not a loss.

## 4 Additional Language Support

So far we have seen that simulations of multiple inheritance perform very well on some examples and bad on others. This is not a really satisfactory conclusion. Can we do better? In this section we look at what can be gained by providing language support for the translations discussed above. We analyze two mechanisms: implementation of the implicit interface of a class and built-in delegation.

### 4.1 similar

In order to ease the implementation of scenarios which require substitutability of child instances for parent instances, let us first explore a new construct for class definition, called `similar`, with the following semantics:

`class C similar D {...}` declares that class `C` implements all the public methods in class `D` and that `C` instances may be used for `D` instances.

This construct permits omission of explicit interface declaration for `D` (which was required in the earlier translation). However, code reuse still entails defining a `D`-type field in class `C`, and forwarding some method calls to it while redefining others.

This approach can lead to name conflicts and is therefore subject to the complications discussed earlier. Compare, for instance

```
class Clock extends Applet similar Thread { ... };
```

with the discussion of the `Clock` applet given earlier.

So, to simplify matters, it is tempting to require that the superclass of `C` and the classes `similar` to `C` have disjoint methods. However, the semantics of forwarding captured by these translations may not be appropriate as explained below.

## 4.2 Built-in Delegation

We now discuss an alternative proposal to simulating delegation in Java, as incorporated in Lava [20]. Lava extends Java with constructs for type-safe *automatic forwarding*. In particular, Lava supports two variants of forwarding : *consultation* and *delegation*. The following description of these variants has been adapted from [21].

An object, called the child, may have modifiable references to other objects, called its parents. Method calls for which the receiver has no matching method are *automatically* forwarded to its parents. When a suitable method is found in a parent (the method holder), it is executed after binding its implicit `this` parameter, which refers to the object on whose behalf the method is executed. Automatic forwarding with binding of `this` to the original method call receiver is called *delegation*, while automatic forwarding with binding of `this` to the method holder is called *consultation*. Delegation is object-based inheritance whereas consultation is just an automatic form of method calling.

Lava uses the keywords `delegatee` and `consultee` to mark instance fields of a child class that hold references to parent objects. These keywords specify whether to interpret forwarded method calls via delegation or via consultation respectively. The two approaches differ when a parent method has been redefined in a child. The *delegatee* will invoke the redefined child method, while the *consultee* will invoke the original parent method. Alternatively, one can say that delegation supports overriding whereas consultation blocks it, as illustrated below.

```
class A {
    public    String a()  { return a1(); }
    protected String a1() { return "A"; }
}

class B {
    public    String b()  { return b1(); }
    protected String b1() { return "B"; }
}

class C extends A {
    delegatee B bp = new B(); // C is subtype of B

    protected String a1() { return "C"; }
    protected String b1() { return "C"; }
}

class Test {
    public static void main(String[] args) {
        C c = new C();
        System.out.println( c.a() + c.b() );
    }
}
```

The expected outcome is "CC". If we used consultation the outcome would be "CB".

Note also that an object of `class C` can be treated as an object of both `class A` and `class B`. Delegation and consultation both have the effect of extending the interface of the child class by the `public` and `protected` methods of the declared parent types. It is instructive to compare the above code with the initial multiple inheritance example and the simulation of delegation.

#### 4.2.1 Conflict resolution

In contrast with ordinary class-based languages, there are situations where conflicts can arise only at run-time. This can happen because, with delegation, the parent object may be any instance of the declared parent type *or of one of its subtypes*. In the latter case, it may contain additional methods, which may have the same signature as methods in the child object.

Such conflicts, that cannot be detected statically, are resolved in Lava by a “semantic compatibility” criterion that a method in a child may only override a method of a parent object *if both override the same method* in a common declared supertype of the child’s class and the parent’s class. Consider the following examples from [21].

<pre>// Intended overriding:  class Child {   delegatee   DeclParent p = new Parent();   void bang(){...} }  class DeclParent {   void b();   void bang(){...} }  class Parent extends DeclParent {   void b(){ this.bang(); }   void bang(){...} }</pre>	<pre>// No accidental overriding:  class Child {   delegatee   DeclParent p = new Parent();   void bang(){...} }  class DeclParent {   void b(); }  class Parent extends DeclParent{   void b(){ this.bang(); }   void bang(){...} }</pre>
---	--

Let `c = new Child()`. Then `c.b()` delegates `b()` to `p`, which calls `bang()` on `c` (recall that delegation has the effect of binding `this` to the object that initially received the forwarded message).

In Lava, for the left example, `bang()` in class `Child` will be invoked, thus overriding `Parent`’s `bang()` — this is sensible to do because both methods are derived as specializations of the one in `DeclParent` — hence it can safely be assumed that they have a compatible semantics.

For the right example, the `bang()` method of class `Child` will *not* be selected because there is no evidence that it has the semantics expected in the calling context. Therefore the `bang()` call will be forwarded to the parent object `p` — just like any other call that finds no applicable method in `c` — and the `bang()` method of `p` will be selected. The net effect is that accidental overriding is barred, preventing cumbersome and hard to locate errors.

## 4.2.2 Multiple delegation

In [9], an extension of Lava is discussed in which a class can have multiple forwardees – delegates and consultants – that define methods under the same name. Such ambiguities have to be resolved, by *explicit overriding*. This construct is a simplification of Eiffel’s `redefine-rename-undefine` mechanism, as illustrated below. The notation

```
localName(args) overrides field<-parentName
```

means that the respective local method overrides the method `parentName(args)` in objects referenced by `field`. If the `parentName` is the same as `localName` it may be omitted, like in the following definition of `clone()` in class `Child`:

```
class Lottery {
  void draw() {...} // draw from lottery
  void clone() {...} // deep clone
  void print() {...} // write to output stream
}

class GUIobj {
  void draw() {...} // draw a GUI element on the screen
  void clone() {...} // deep clone
  void write() {...} // write to output stream
}

class Child {
  protected delegatee Lottery l;
  protected delegatee GUIobj g;

  // Conflicting names with different semantics:
  // each overridden by another local method (subsumes "renaming").
  void drawLottery() overrides l<-draw {...}
  void drawScreen() overrides g<-draw {...}

  // Same names with same semantics:
  // overridden by the same local method.
  void clone() overrides l, g {...}

  // Different names with same semantics:
  // overridden by the same local method (subsumes "undefine").
  void write() overrides l<-print, g<-write {...}
}
```

Multiple delegation also enables *repeated delegation* (analogous to repeated inheritance). Therefore it raises the semantic problem pointed out for Eiffel’s `select` statement in Section 3. A detailed analysis of this problem in the context of delegation, and the development of a solution are contained in [9]. A deeper discussion of explicit overriding can also be found there.

### 4.3 Summary

This section complemented the “pure Java” simulation by a discussion of possible language extensions that support forwarding-based designs.

The discussion demonstrated that asking for multiple inheritance in Java is not a good idea given that more powerful, dynamic alternatives are available. If the language should ever be extended, support for (single or multiple) delegation would be preferable to adding multiple inheritance. The primary advantage is the replacement of anticipated, *static* composition of classes via inheritance by unanticipated (mixin-style) *dynamic* composition of objects via delegation. This holds even for static delegation. Dynamic delegation offers the added benefit of being able to model dynamic evolution of object structure and behaviour.

## 5 Approximating Generics in Java

A class is an implementation of a data type. The “container” data types (such as the `Set`, the `List`, the `Tree`, etc) can be abstracted and parameterized with respect to the type of the elements, in order to model *homogeneous* data structures (which contain just one type of elements). Such a parameterized type/module can be instantiated before use by binding concrete types to generic type parameters. For instance, one can instantiate `List` generic type/class to obtain various concrete types/classes such as `List of integers`, `List of strings`, `List of dates`, etc. These examples illustrate *unconstrained genericity*.

In contrast, in certain other situations, a generic type parameter may be partially *constrained*. For instance, a `Sort` routine or a `Dictionary` type may require the elements to be from a `LinearOrder` type. The templates in C++, the generics in Eiffel and Ada-95, the functors in SML [22], etc all support convenient implementation of such types (resp. subprograms) as parameterized classes (resp. routines) where the type of a generic parameter expresses additional requirements. These parameterized modules are then suitably instantiated by binding types that satisfy the necessary constraints, to generic type parameters. This is known as *constrained genericity*.

Java does not support any construct akin to templates or generics. An extension to the Java language and the Java Virtual Machine is proposed in [23] to incorporate templates in their full glory. Pizza [24] and GJ [25] are supersets of Java which support parametric polymorphism that can be translated into pure Java. Here we review and analyze two idioms to approximate generic modules: one that uses the class `Object` and other reference types as the type of the generic element, and another that uses reflection. The first approach is exemplified by various utility classes of the *Java 2 collections framework*, while the second approach is illustrated using an example developed in Section 5.3. Additionally, a marker interface, as discussed in Section 3.2, can be used to express the requirement that the elements of a `Dictionary` must be linearly ordered.

### 5.1 Using class `Object`

A generic class parameter in a C++ template can be straightforwardly translated into Java by turning it into a parameter of type `Object`, the root of the Java class hierarchy. However this translation is only approximate since the Java class types are reference types [27] as explained below.

- A C++ generic class parameter can be instantiated to a primitive type such as `int`, `char`, etc, as well as a class type, while, in Java, only the latter can be simulated. To instantiate a generic class parameter to a primitive type, Java requires the primitive type to be *objectified* using the corresponding *wrapper* class such as `Integer`, `Character`, `Boolean`, etc. Because these wrapper classes implement *immutable* objects, their use does *not* impact the semantics of assignment, the parameter passing mechanism, and the interpretation of `final` variables: for immutable objects these operations notice no difference between “copy semantics” vs “reference semantics”, “call by value” vs “call by reference”, and “immutable variable” vs “immutable reference to immutable object”. However, equality is not captured well by the translation: whereas `1 == 1` is true, `new Integer(1) == new Integer(1)` is false. Here the difference between “value semantics” vs “reference semantics” can only be concealed if we use the (value based) `equals()` method instead of “`==`”. In Java, `equals()` is a method of class `Object`, and thus available on any object.
- Simulation of instantiation of a generic class parameter with a class type (as opposed to a primitive type) is also convoluted. For instance, to guarantee type safety, Java requires explicit type casts in expressions that use functions returning values of the generic parameter type. (Observe that, in the Java translation, such functions have a return type `Object`.) Consider the following template, which illustrates *unconstrained genericity*:

```

template <class T>
class Ccpp {
    private:      T t;
    public:
        Ccpp (T x) { t = x; }
        T f(T x) { return t; }
}

```

and a first approximation to it in Java:

```

class Cjava {
    private Object t;
    public Cjava (Object x) {
        t = x;
    }
    public Object f(Object x) {
        return t;
    }
    public static void main (String[] args) {
        Integer io = new Integer(5);
        Cjava c = new Cjava(io);
        Integer jo = (Integer) c.f(io);
    }
}

```

In the above Java translation, the method `f` is not restricted to `Integer` argument. To impose the necessary type constraint, one can use *inheritance* and *redefinition*. However, to override a parent method in a subclass `CIntjava`, Java requires that the signature of the overriding method be the same as that of the corresponding parent method. (Otherwise, the new definition is treated as a valid overload.) So extra type checking code is needed to preserve the behavior. Observe also that an instance of `CIntjava` can be assigned to a variable of type `Cjava`.

Even though the subclass has “method-stubs” of the order of the number of methods in the parent class, it does not require duplication of method bodies.

```
class CIntjava extends Cjava {
    public CIntjava (Integer x) {
        super(x);
    }
    public Object f(Object x) {
        if (x instanceof Integer) {
            return super.f(x);
        } else return null;          // "Erroneous Call"
    }
}
```

This approach resembles *homogeneous translation* of Pizza’s polymorphism into Java [24].

Alternatively, one can change the signature of the method `f` in `CIntjava` to accept and return only `Integer` objects. This can be accomplished using *composition* and *forwarding*. Unfortunately, all this can clutter up the code and introduce run-time overheads.

```
class CIntjava {
    protected Cjava c;
    public CIntjava (Integer x) {
        c = new Cjava(x);
    }
    public Integer f(Integer x) {
        return (Integer) c.f(x);
    }
}
```

Observe that Java does not permit *covariant typing* of Eiffel [5] where a parent method can be overridden in the subclass using a method whose signature is more “restrictive” than the corresponding parent method. Observe also that Pizza’s support for parameteric polymorphism improves readability, but has similar overheads due to translation into Java [24].

Refer to the Java utility classes/interfaces such as `java.util.List`, `java.util.LinkedList`, `java.util.Hashtable`, `java.util.HashSet`, etc for more examples.

## 5.2 Using interface/class parameters

Many parameterized modules require specification of additional semantic constraints or signature constraints on a generic parameter. The former corresponds to the expectations on the services provided by the actual argument and the latter on the requirements on the interfaces supported by the actual argument.

Languages such as SML, C++, Eiffel, Ada-95, etc all support such constrained genericity. In Java, such constraints can be expressed using *class type method parameters* to capture semantic constraints, and *interface type method parameters* to capture signature constraints in place of `Object` type parameters discussed earlier.

Refer to the uses of Java classes/interfaces such as `java.awt.event.MouseListener`, `java.util.Iterator`, `java.util.SortedSet`, `java.util.TreeSet`, `java.awt.Container`, etc for additional examples.

The Java approach has one important limitation — certain type equality constraints cannot be expressed or enforced at compile time. For example, let `S` be a subclass of `C`. Then, a collection of instances of `C` may potentially contain a mix of “direct” instances of `C` and “direct” instances of `S`. However it is not possible to specify and verify statically (that is, without resorting to type tests and casts) a collection class that is made up entirely of either “direct” instances of `C` or “direct” instances of `S`, but not a mix of the two. See [26, 24] for proposals to extend Java language to incorporate such *constrained polymorphism*.

The language GJ, which is a conservative extension of Java 2, supports generics including covariant overriding [25]. The translator for GJ into Java incorporates generalization of the examples sketched above.

## 5.3 Using Java Reflection

As observed earlier, changes to a generic class (such as adding a method or a parameter to a method) can require explicit changes to all its “instantiation” subclasses. To minimize the impact of such changes, the *Reflection* API can be used. This is illustrated by the following `Dictionary` class example that is “parameterized” with respect to a *linear order* class as follows. (We have included the details for the interested readers; however, this example can be safely skipped. The `Dictionary` defined here is not to be confused with the obsolete abstract class `java.util.Dictionary`.)

```
import java.util.*;
import java.lang.reflect.*;

class Dictionary {
    private Class    orderClass;
    private Method  lessThan,  equal;
    private Vector  vec = new Vector();

    public Dictionary(Class cl) {
```

```

    orderClass = cl;          /* generic parameter */
    try {
        lessThan = cl.getMethod("lessThan", new Class[]{cl});
        equal     = cl.getMethod( "equal" , new Class[]{cl});
    } catch (Exception e) {System.out.println(e);}
}

public void insert(Object o) {
    int i;
    if (orderClass.isInstance(o)) {
        try {
            for (i = 0; i < vec.size(); i++) {
                if ( auxInvoker( lessThan, o, new Object[]{vec.elementAt(i)} ) ) break;
            }
            vec.insertElementAt(o,i);
        } catch (Exception e) {System.out.println(e);}
    }
}

public Object member(Object o) {
    if (orderClass.isInstance(o)) {
        try {
            for (int i = 0; i < vec.size(); i++) {
                if ( auxInvoker( equal, o, new Object[]{vec.elementAt(i)} ) ) return o;
                if ( auxInvoker( lessThan, o, new Object[]{vec.elementAt(i)} ) ) break;
            }
        } catch (Exception e) {System.out.println("*2* " + e);}
    }
    return null;
}

private static boolean auxInvoker(Method m, Object o, Object[] arr)
    throws IllegalAccessException,
    IllegalArgumentException, InvocationTargetException {
    Boolean b = (Boolean) m.invoke(o, arr);
    return b.booleanValue();
}
}

```

The above class can be used to implement a Dictionary of strings as follows.

```

class EGLinearOrder {
    String s;
    EGLinearOrder(String s) {
        this.s = s;
    }
    public boolean lessThan(EGLinearOrder t) {

```

```

        return (s.compareTo(t.s) == -1);
    }
    public boolean    equal(EGLinearOrder t) {
        return (s.compareTo(t.s) == 0);
    }
    public String toString() {
        return ( "Instance of EGLinearOrder with field s = " + s );
    }
}

public class ReflectGenerics {
    public static void main (String[] args) {
        EGLinearOrder    o    =    new EGLinearOrder("Jill");
        EGLinearOrder    p    =    new EGLinearOrder("Jack");
        Dictionary        c    =    new Dictionary(p.getClass());
        c.insert(o);
        c.insert(p);
        System.out.println("Result ---> " + (EGLinearOrder) c.member(p));
    }
}

```

Even though the code for the generic `Dictionary` class using reflection looks complicated (in fact even downright “ugly”), the code for instantiation is straightforward. Unlike the non-reflective solution presented first, changes to the `Dictionary` class — such as adding a method `public void delete(Object o) ...` — do not require explicit changes to its instantiations. However, this approach cannot be used to instantiate generic parameters to primitive types, and will lose out on compile-time error checks<sup>3</sup>. It is also clearly slower than the non-reflective simulation.

## 6 Conclusions

In this paper we reviewed Java idioms used by programmers to approximate generics and multiple inheritance of classes, to accrue reusability benefits.

Multiple inheritance was simulated by the joint use of *forwarding* as a means to achieve code reuse, *interfaces* as a means to achieve polymorphism, and *back-references* as a means to approximate overriding. The application of these techniques to various examples was demonstrated, shedding light on what can and cannot be achieved by the simulation.

An alternative proposal to simulating delegation in Java, as incorporated in Lava, which extends Java with constructs for type-safe *automatic forwarding*, was also discussed.

Finally, two approximations to generic modules were analyzed: one that uses the class `Object` and reference types as the type of the generic parameter (to capture unconstrained genericity and constrained genericity respectively), and another that uses reflection. The former approach is not robust with respect to certain modifications to methods (such as

---

<sup>3</sup>Commercial software tools, such as PREFIXCO Corp.’s PREFIX, that are based on software component simulation technology, may provide means to detect some errors prior to software deployment that would otherwise be caught only at run-time.

adding a new method, or changing the signature of an existing method, etc), while the latter approach leads to inefficient and difficult to read code.

## Acknowledgments

We wish to thank the anonymous referees and Pascal Costanza for their comments and suggestions.

## References

- [1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture : A System of Patterns*, John Wiley and Sons, 1996.
- [3] J. Vlissides: *Pattern Hatching*. Addison-Wesley, 1998.
- [4] I. Joyner: *A C++?? Critique*, Eiffel Liberty Resources at <http://www.elj.com/eiffel/ij/>, 1998.
- [5] B. Meyer: *Object-Oriented Software Construction*. Second Edition. Prentice-Hall, 1997.
- [6] Laks V.S. Lakshmanan, and K. Thirunarayan: *Declarative Frameworks for Inheritance*, In: *Logics for Databases and Information Systems*, Eds: J. Chomicki and G. Saake, Kluwer Academic Publishers, 357-388, 1998.
- [7] B. Stroustrup: *The C++ Programming Language*. Third Edition, Addison-Wesley, 1997.
- [8] T. Budd: *An Introduction to Object-Oriented Programming*. Second Edition. Addison-Wesley, 1997.
- [9] G. Kniesel: *Multiple Inheritance and Delegation Revisited*, University of Bonn, Germany, 2000. (In preparation)
- [10] A. Appel: *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.
- [11] K. Driesen, U. Hölzle, and J. Vitek: Message Dispatch on Pipelined Processors, In: *Proceedings ECOOP'95*, Edited by: W. Olthoff, LNCS 952, Springer-Verlag, 253-282, 1995.
- [12] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
- [13] L. Cardelli and P. Wegner: On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
- [14] W. Harrison, H. Ossher, and P. Tarr: Using delegation for software and subject composition. Research Report RC 20946 (922722), IBM Research Division, T.J. Watson Research Center, 5 August 1997.

- [15] G. Kniesel: *Delegation for Java - API or Language Extension?* Technical Report IAI-TR-98-4, ISSN 0944-8535, University of Bonn, Germany, 1998.
- [16] A. Blewitt: *Java Tip 71: Use dynamic messaging in Java.* <http://www.javaworld.com/javaworld/javatips/jw-javatip71.html>, 1999.
- [17] B. Stroustrup: *The Design and Evolution of C++*. Addison-Wesley, 1994.
- [18] K. Arnold and J. Gosling: *The Java Programming Language*. Second Edition. Addison-Wesley, 1999.
- [19] T. Budd: *Understanding Object-Oriented Programming with Java*. Updated Edition. Addison-Wesley, 2000.
- [20] G. Kniesel: *Dynamic Object-Based Inheritance with Subtyping*, PhD thesis, University of Bonn, Computer Science Department III, 2000.
- [21] G. Kniesel: *Type-Safe Delegation for Run-time Component Adaptation.*, European Conference on Object-Oriented Programming, Springer LNCS 1628, p. 351–366, 1999.
- [22] J. D. Ullman, *Elements of ML Programming*. 2nd Ed. (ML97), Prentice Hall, 1998.
- [23] A. C. Myers, J. A. Bank, and B. Liskov: Parameterized Types in Java, In: *Proc. of 24th POPL*, 132-145, 1997.
- [24] M. Odersky, and P. Wadler: Pizza into Java: Translating theory into practice, In: *Proc. of 24th POPL*, 146-159, 1997.
- [25] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler: Making the future safe for the past: Adding Genericity to the Java Programming Language, In: *Proc. of OOPSLA-98*, October 1998.
- [26] K. Bruce: *Increasing Java's expressiveness with ThisType and match-bounded polymorphism*, Draft, 1997.
- [27] J. Gosling, B. Joy, G. Steele, and Gilad Bracha: *The Java Language Specification*. Second Edition. Addison-Wesley, 2000.