Safe Structural Conformance for Java

Konstantin Läufer[*] Gerald Baumgartner[**] Vincent F. Russo[***]

[*] Department of Mathematical and Computer Sciences
Loyola University Chicago
6525 N. Sheridan Road
Chicago, IL 60626, USA
laufer@cs.luc.edu

[**] Department of Computer and Information Science
The Ohio State University
395 Dreese Lab., 2015 Neil Ave.
Columbus, OH 43210–1277, USA
gb@cis.ohio-state.edu

[***] NobleNet, Inc.
337 Turnpike Road
Southboro, MA 01772–1709, USA
vince@noblenet.com

June 17, 1998

Technical Report OSU-CISRC-6/98-TR20
Computer and Information Science Department, Ohio State University

Abstract

In Java, an interface specifies public abstract methods and associated public constants. Conformance of a class to an interface is by name. We propose to allow structural conformance to interfaces: Any class or interface that declares or implements each method in a target interface conforms structurally to the interface, and any expression of the source class or interface type can be used where a value of the target interface type is expected. We argue that structural conformance results in a major gain in flexibility in situations that require retroactive abstraction over types.

Structural conformance requires no additional syntax and only small modifications to the Java compiler and optionally, for performance reasons, the virtual machine, resulting in a minor performance penalty. Our extension is type-safe: A cast-free program that compiles without errors will not have any type errors at run time. Our extension is conservative: Existing Java programs still compile and run in the same manner as under the original language definition. Finally, structural conformance works well with recent extensions such as Java remote method invocation.

We have implemented our extension of Java with structural interface conformance by modifying the Java Developers Kit 1.1.5 source release for Solaris and Windows 95/NT. We have also created a test suite for the extension.

1 Introduction

Java (Gosling et al., 1996) differs from many statically typed object-oriented languages by offering two separate constructs, the *class* and the *interface*, for user-defined abstract types. An interface contains only public abstract methods and public final fields (constants), but no method implementations. This separation allows independent class and interface hierarchies and solves some of the shortcomings of object-oriented languages that use classes for defining both interfaces and implementations.

We argue that Java is still limited by the requirement that classes must be declared to conform to their interfaces. A class declared to conform to an interface must provide implementations for each method in the interface or leave the method abstract. We identify interfaces in Java with signatures, an extension of C++ (Baumgartner and Russo, 1995), and propose allowing a class to conform structurally to an interface without explicitly associating the class with the interface.

The remainder of this paper is structured as follows. First, we describe the relevant aspects of the Java type system. Next, we present our extension of Java with structural conformance. Then we give examples in the extended language. Thereafter, we describe the implementation of our extension based on the Java Developers Kit (Sun Microsystems, 1997b). Finally, we discuss the integration of structural conformance with Java's Remote Method Invocation (Sun Microsystems, 1997c) and explore possible future extensions to the language.

2 Classes, Interfaces, and Conformance in Java

In many statically typed object-oriented languages, most notably C++ (Ellis and Stroustrup, 1990), a single construct, namely the *class*, is used to define and implement new types and to provide type abstraction, code reuse, and subtyping (conformance). This overuse of a single construct limits the expressiveness of the type system (Baumgartner and Russo, 1995).

By contrast, Java (Gosling et al., 1996) offers two separate constructs, the *class* and the *interface*. Every class has exactly one immediate superclass and zero or more immediate interfaces. The root class `Object` does not have an immediate superclass (represented by a `null` superclass); the same is the case for interfaces. Classes that provide only a partial implementation are called *abstract*; interfaces are a special case of fully abstract classes without code or data. Java thus provides single inheritance from classes for the purpose of code reuse, along with a limited form of multiple inheritance from interfaces for the purpose of specification.

Most object-oriented languages provide some form of conformance, allowing an instance of a class to be used wherever an instance of the class's superclass is expected. In Java, this mechanism is called *(implicit) reference conversion* and allows conformance not only to superclasses, but also to (immediate or indirect) interfaces. (An indirect interface is an interface of a superclass or an interface from which an interface of the class was extended.) Since the relationships that entail conformance are established by declaration, this kind of conformance is called *conformance by name*.

Separate type and class hierarchies are possible

The separation of the class and interface constructs overcomes some of the problems caused by having a single construct (cf. Baumgartner and Russo, 1995). In particular, it is possible in Java to build abstract type hierarchies separate from the corresponding implementation hierarchies. This capability is important because the two hierarchies often evolve in opposite directions.

As an example, consider two abstract types `Queue` and `Dequeue` for FIFO queues and double-ended queues, respectively (a similar example was presented by Snyder (1986)). The abstract type `Dequeue` provides the same operations as `Queue` as well as two additional operations for inserting at the head and for removing from the tail of the queue. Therefore, `Dequeue` conforms to

`Queue`. On the other hand, a `Dequeue` is easily implemented in terms of the array class `java.util.Vector` (Sun Microsystems, 1997a), or as a doubly linked list. A `Queue` is trivially implemented by extending `Dequeue` and ignoring the superfluous operations.

```
interface Queue {
    Object dequeueHead();
    void enqueueTail(Object x);
    boolean isEmpty();
}

interface Dequeue extends Queue {
    void enqueueHead(Object x);
    Object dequeueTail();
}

class DequeueImpl extends java.util.Vector implements Dequeue {
    public final void enqueueHead(Object x) { insertElementAt(x, 0); }
    public final Object dequeueHead() {
        Object x = firstElement();
        removeElementAt(0);
        return x;
    }
    public final void enqueueTail(Object x) { addElement(x); }
    public final Object dequeueTail() {
        Object x = lastElement();
        removeElementAt(size() - 1);
        return x;
    }
}

class QueueImpl extends DequeueImpl implements Queue {
    // We do not want enqueueHead() and dequeueTail() here,
    // but there is no way to avoid inheriting them.
}

public class Hierarchies {
    public static void main(String[] arg) {
        Queue q1 = new QueueImpl();
        Dequeue q2 = new DequeueImpl();

        q1.enqueueTail("Hello");
        q1.enqueueTail("World");
        System.out.println(q1.dequeueHead());

        q2.enqueueHead("World");
        q2.enqueueHead("Hello");
        System.out.println(q2.dequeueTail());
    }
}
```

The clause `implements Queue` in the declaration of the class `QueueImpl` is redundant, since its superclass implements an interface that extends `Queue`. Nevertheless, we retain this clause to document the intended conformance relationship. Furthermore, because of transitivity, there are unintended implicit relationships `DequeueImpl implements Queue`, which is acceptable, and `QueueImpl implements Dequeue`, which is undesirable. These relationships are shown in Figure 1. This example illustrates that a true separation of the abstract type and implementation hierarchies is not possible in Java since the language does not allow code reuse without defining a conformance relationship. (Section 6 discusses how structural conformance together with a renaming mechanism could be used without introducing unwanted conformance relationships.)

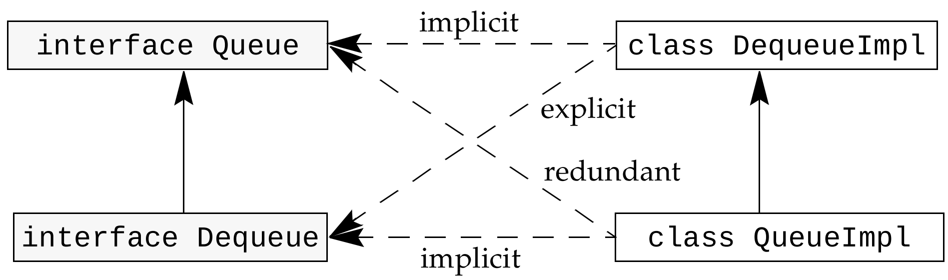

Figure 1: Explicit, implicit, and redundant conformance relationships

Retroactive type abstraction is hard

The integration of different existing class hierarchies may require abstracting over the types of these class hierarchies. This usually involves designing a new, common abstract type hierarchy on top of the existing implementation hierarchies. The problem is that one would have to modify the sources of the existing hierarchies to declare their classes as implementations of the interfaces of the new abstract hierarchy. Since class libraries are often available in compiled form only, the scope of this approach is limited. Retroactive abstraction over existing code was discussed in the context of a C++ extension (Baumgartner and Russo, 1994 and 1997) and later rediscovered in a design pattern context (Cleeland et al., 1997).

As an example (Granston and Russo, 1991), consider two class libraries for X-Window widgets. One hierarchy is rooted at `OpenLookObject` and the other at `MotifObject`. Suppose that all widgets implement the operations `display` and `move`. Is it possible to construct a list containing widgets from both class libraries at the same time? The answer is yes, but the solution would involve either introducing a discriminated union for the widgets, or using multiple inheritance to implement a new set of leaf classes in each hierarchy, or building a hierarchy of forwarding classes. All three solutions are undesirable or problematic: The first one is inelegant and not extensible, while the other two introduce many superfluous class names. Furthermore, Java does not support multiple inheritance of classes anyway. It does not support templates either, which could have facilitated the creation of forwarding classes.

In another, perhaps more realistic, scenario that calls for retroactive type abstraction, we want to abstract over some types from an existing class hierarchy and provide an alternative implementation in the form of a new class hierarchy. If the existing class hierarchy was not designed to support this form of reuse or if the alternative implementation uses different data structures, then we are in the same situation as above.

Within the same scenario, assume that the desired abstractions are similar but less specific interfaces than the ones of the existing class hierarchy. We could then provide an alternative implementation with respect to these existing interfaces. For operations that are part of these interfaces, but not of our desired abstraction, we would provide dummy implementations that raise a run-time exception. However, this approach defeats strong typing because those operations are specified in the interface but not really understood by the objects that are supposed to conform to the interface.

For example, suppose we have an interface `File` with operations `read` and `write` but no interface for read-only files. If we write a device driver for a CD-ROM that implements the `File` interface, applications that use the `File` interface to write to a device cannot detect statically if the device is actually a CD-ROM.

3 Structural Class-to-Interface Conformance

We propose to extend Java by allowing structural conformance to interfaces. Specifically, any class or interface that declares (or implements) each method in a target interface *conforms structurally* to the interface, and any expression of the source class or interface type can be used wherever a value of the target interface type is expected. The source class or interface is no longer required to declare conformance to the target interface by name. Structural conformance also applies to the `instanceof` operator and to casts from the target interface down to a (structurally conforming) source class or interface.

Concretely, structural conformance applies in three kinds of situations. (We assume that the class `QueueImpl` conforms structurally to the interface `Queue`.)

- Reference conversions from the source class or interface to the target interface:

```
Queue q = new QueueImpl();
```

- Casts from the target interface down to the source class or interface:

```
QueueImpl qi = (QueueImpl) q;
```

- Expressions involving the `instanceof` operator between an object of the target interface and the source class or interface:

```
if (q instanceof QueueImpl) { /* ... */ }
```

  This case also includes the related methods `java.lang.Class.isAssignableFrom` and `java.lang.Class.isInstance`.

We propose to allow structural conformance to designated interface types as targets because the purpose of interfaces is to specify behavior without giving an implementation. On the other hand, we make no attempt to infer class-subclass relationships based on the public methods of a class because the purpose of subclassing is explicit code reuse at the implementation level. Mechanisms for designating which interfaces allow structural conformance are discussed below in the subsection on accidental conformance.

The proposed extension does not require any changes to the language syntax. Only minor modifications to the Java compiler and, optionally, the virtual machine are required as described below. Our extension is type-safe in the sense that any cast-free program that compiles without type errors will not cause any type errors at run time. Our extension is conservative in the sense that existing Java programs still compile and execute in the same manner as under the original language definition.

Formal definition of conformance

In the following definition of conformance, `I` is an interface, and each of `X` and `Y` is either a class or an interface. Every class other than `java.lang.Object` has one immediate superclass. Every class or interface has zero or more immediate interfaces.

`X` *conforms* to `Y` if and only if

- `X` conforms by name to `Y`, or
- `Y` is an interface that allows structural conformance, and `X` conforms structurally to `Y`.

`X` *conforms by name* to `Y` if and only if

- `X` is identical to `Y`, or
- `X`'s immediate superclass, if it exists, conforms by name to `Y`, or

- some immediate interface `I` of `X` conforms by name to `Y`.

`X` *conforms structurally* to `I` if and only if `X` conforms by name to `I` or all of the following three conditions simultaneously hold:

- `I` is an interface that allows structural conformance, and
- `X` overrides each method specified in `I`, and
- `X` conforms to each immediate interface of `I`.

`X` *overrides* a method `Y.f` specified in `Y` if and only if there is a method `f` in `X` that

- is at least as accessible as `Y.f`,
- has the same type signature as `Y.f`, and
- throws only checked exceptions that are subclasses of, or the same class as, the checked exceptions that `Y.f` throws.

This definition of overriding is equivalent to the one given in the Java language specification (Gosling et al., 1996) and includes implementing the method as well as leaving it abstract. Furthermore, since `I` must be an interface, all its methods are public, and those methods in `X` that override methods in `I` must be public as well.

Type safety

For the purpose of this paper, we define type safety as follows:

> *Any program without casts that compiles correctly will not raise a* `NoSuchMethodError` *or any other type-related error or exception.*

The idea behind this definition is that the compiler is able to prove from the type information available at compile-time that the run-time behavior of a program is safe. Hence we allow a reference conversion only when the source type provides at least the public methods of the target type. To be on the safe side, the Java[1] compiler verifies that a new class is a suitable implementation of an interface or a subclass of an existing class at the point where the new class is defined. This is the conformance-by-name approach.

However, this is not the only way to guarantee type-safe reference conversions. Instead of requiring an `implements` declaration between classes and interfaces, it is equally safe but much more flexible to examine each intended reference conversion individually. Since the compiler knows both the source type and the target type in a reference conversion, it can check whether the source type provides at least the public methods required by the expected interface type without requiring a prior declaration. This is the structural conformance approach.

Controlling accidental conformance

The usual objection to structural conformance is that it creates the possibility of *accidental* (non-semantic) conformance. Such conformance occurs when a class provides all the methods required by an interface, but no semantic relationship between the two is intended. In the following example, `CardPlayer` conforms structurally to `Shape`, although it makes no sense to treat a `CardPlayer` as a `Shape`.

1. Actually, Java is not statically type-safe with respect to the conversion of array types (Gosling et al., 1996, Cook, 1995), but relies on a run-time type check. This choice was made to support polymorphic functions on arrays via subtyping in the absence of generic classes (templates).

```
interface Shape {
    void draw(); // draw the shape on a drawing area
}

class CardPlayer {
    public void draw() { /* draw a hand of cards */ }
    // ...
}
```

Marker interfaces are empty interfaces that represent semantic properties. Certain marker interfaces have special meanings in Java; for example, `java.lang.Cloneable` is implemented by classes that support the `clone` method, and `java.rmi.Remote` is extended by interfaces of remote objects (Sun Microsystems, 1997c). Under structural conformance, every class or interface would accidentally conform to an empty interface. Therefore, marker interfaces should generally be excluded from structural conformance.

Clearly, any language with structural conformance needs a way to control accidental conformance. The standard solution is to include *properties* (America and van der Linden, 1990, Jenks and Sutor, 1992) in an interface to represent a semantic specification of the interface type. For example, the `Shape` interface might include the property `Graphical`, and the `Stack` interface might include the property `LIFO`. Any shape implementation conforms to the `Shape` interface only if it also includes the property `Graphical`, just as any stack implementation conforms to the interface `Stack` only if it also includes the property `LIFO`. This mechanism is a variant of conformance by name, where the names are now property names instead of class and interface names. A programmer cannot be prevented from putting the `LIFO` property into a queue implementation, but, on the other hand, a programmer could also write a queue implementation and say it "`implements Stack`".

In general, the design space for languages with controlled structural conformance can be organized as follows:

1. Structural conformance is the default, and there is an explicit way to require conformance by name.
   a. Interfaces as properties. Some interfaces play the role of properties and thus require conformance by name. There are three choices as to which interfaces are treated as properties:
      (i) All empty interfaces.
      (ii) Only interfaces immediately extending a special interface, e.g., `Property`.
      (iii) Only interfaces declared with a new keyword, e.g., `property`.
   b. Explicit properties. There are two choices:
      (i) Dummy method declarations are used as properties.
      (ii) Special syntax is provided for property declarations included in interfaces and classes.
2. Conformance by name is the default, and there is an explicit way to allow structural conformance. This branch involves two orthogonal decisions:
   a. Syntax. There are two choices for declaring interfaces to allow structural conformance:
      (i) Only interfaces extending a special interface, e.g., `Structural`.
      (ii) Only interfaces declared with a new keyword, e.g., `structural`.
   b. Inheritance. There are two choices for interfaces extending an interface that already allows structural conformance:

(i) Such interfaces allow structural conformance as well.

(ii) Such interfaces do not implicitly allow structural conformance.

Our extension to the Java language follows choice 2.a.(i)/b.(i): Structural conformance is optional and enabled only for interfaces that (immediately or indirectly) extend the special marker interface `java.lang.Structural`; conformance by name is required for all other interfaces. The main benefit of this approach is that it results in a *conservative extension* to the language: All programs that are correct according to standard Java are also correct within our extension; conversely, all programs that are incorrect according to standard Java are also incorrect within our extension. Another benefit is that extension from the interface `Structural`, like extension from other interfaces, is transitive. Furthermore, no special case is needed for predefined marker interfaces. A potential drawback in practice is that legacy interfaces that do not extend `Structural` can never be used with structural conformance, but this is a small price to pay for keeping our extension conservative.

The following example illustrates our extension. `Shape` extends `Graphical`, but `Graphical` does not extend `Structural`. Hence any class or interface intended to conform structurally to `Shape` must not only provide a `draw` method, but also conform by name to `Graphical`. Consequently, `Circle` conforms structurally to `Shape`, but `CardPlayer` does not.

```
interface Graphical { /* an (empty) marker interface */ }

interface Shape extends Graphical, Structural {
    void draw(); // draw the shape on a drawing area
}

class Circle implements Graphical {
    public void draw() { /* ... */ }
    // ...
}
class CardPlayer {
    public void draw() { /* draw a hand of cards from a deck */ }
    // ...
}
Shape s1 = new Circle();     // OK: Circle implements Graphical
Shape s2 = new CardPlayer(); // error: CardPlayer does not implement Graphical
```

4 Examples

In this section, we present examples illustrating the capabilities of structural conformance.

A simple example: queues revisited

Our first example resembles the queue/double-ended queue example from above, but uses structural conformance. The interfaces and classes are defined as above, but the classes no longer have `implements` clauses.

```
interface Queue extends Structural {
    Object dequeueHead();
    void enqueueTail(Object x);
    boolean isEmpty();
}

interface Dequeue extends Queue {
    void enqueueHead(Object x);
```

```
    Object dequeueTail();
}

class DequeueImpl extends java.util.Vector {
    // ...
}

class QueueImpl extends DequeueImpl {
}
```

Now instances of the implementation classes conform structurally to the corresponding interfaces:

```
Queue q3 = new QueueImpl();
Dequeue q4 = new DequeueImpl();
// ...
```

Alternate implementation of a full interface

A common scenario is that we already have an implementation of a class but would like to provide alternate implementations with the same interface as the original class. If we need the capability to switch implementations in the application, all implementation classes must conform to the same interface.

Consider the example of a class for a random-access file on a local disk, which is provided in the `java.io` package:

```
class RandomAccessFile implements DataOutput, DataInput {
    public void close() throws IOException { /* ... */ }
    public FileDescriptor getFD() throws IOException { /* ... */ }
    public long getFilePointer() throws IOException { /* ... */ }
    long length() throws IOException { /* ... */ }
    void seek(long pos) throws IOException { /* ... */ }
    // implementations of methods from the interfaces DataOutput and DataInput
}
```

We would like to add an alternate implementation for a remote random-access file, perhaps using the Proxy pattern (Gamma et al., 1995). We proceed in two steps, as illustrated in Figure 2. First, we distill the full interface from the existing class. The existing class conforms structurally to this interface.

```
interface RandomAccess extends DataInput, DataOutput, Structural {
    void close() throws IOException;
    FileDescriptor getFD() throws IOException;
    long getFilePointer() throws IOException;
    long length() throws IOException;
    void seek(long pos) throws IOException;
}
```

Next, we implement the new alternate class. We also state that the new class implements the distilled interface. Besides serving documentary purposes, this results in early conformance checking at the time the class is defined instead of late checking when an instance of the class is assigned to an interface variable.

```
class RemoteRandomAccessFile implements RandomAccess {
    // same methods, but with different implementations suitable for remote use
}
```

Without structural conformance, we would have to write a forwarding class that implements the interface `RandomAccess` by name and forwards requests to the class `RandomAccessFile`.

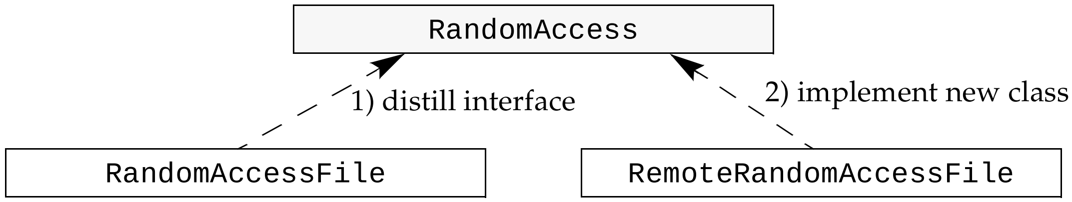

Figure 2: Alternate implementation of a full interface

Abstraction to a reduced interface

In another scenario, we want to deal uniformly with different existing implementations. This requires retroactive abstraction over those implementations to a possibly reduced interface.

For example, consider an application that performs read-only random access to databases located either on a disk drive or on a CD-ROM drive. Again, we access disk files using the class `java.io.RandomAccessFile`, but using only the methods for reading from the file, not the ones for writing to the file. We also provide a class for accessing information on CD-ROM drives. Since CD-ROM drives are read-only devices, the corresponding class implements only the `java.io.DataInput` interface and provides additional methods for random access:

```
class CDRomDrive implements DataInput {
    long length() throws IOException { /* ... */ }
    public void seek(long pos) throws IOException { /* ... */ }
    // implementation of methods from interface DataInput
}
```

Besides the methods for reading data, the application uses the random-access method `seek`. We therefore abstract retroactively over both classes, with a reduced interface as the result. This process is illustrated in Figure 3.

```
interface ReadOnlyRandomAccess extends DataInput, Structural {
    long length() throws IOException;
    void seek(long pos) throws IOException;
}
```

Again, without structural conformance, we would have had to write a forwarding class that implements the interface `ReadOnlyRandomAccess` by name and forwards requests to the class `RandomAccessFile`.

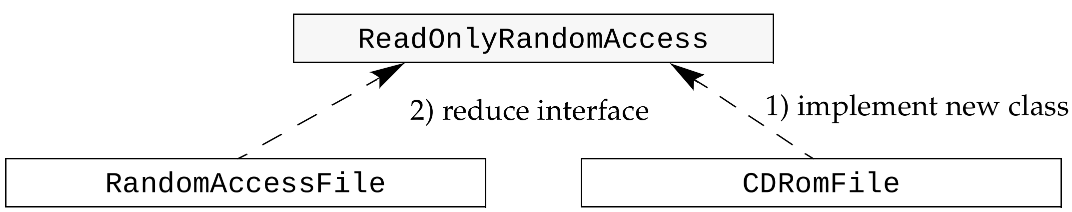

Figure 3: Abstraction to a reduced interface

Structural conformance is also useful in the context of persistent objects. Suppose that we have some objects of class `RandomAccessFile` stored on a disk together with a representation of their type. (Although the example applies to other persistent data structures equally well, files are an obvious choice because they are commonly stored on disks.) We later realize that the original class hierarchy was not well-designed and add an interface `ReadOnlyRandomAccess` and a class `ReadOnlyRandomAccessFile` as a superclass of `RandomAccessFile`. If conformance by name is used, we are no longer able to read the old files after the system is upgraded to the new class and interface hierarchies. By contrast, if the persistent data structure is type-checked structurally (Connor et al., 1990, Morrison et al., 1996), evolving the system in this way does not cause any problems.

5 Implementation

While the proposed extension requires no changes to the Java syntax at all, it does require subtle changes to the Java compiler and optionally, for performance reasons, the virtual machine. This section describes the changes we have made to the Java Developers Kit 1.1.5 source release (Sun Microsystems, 1997b). In addition, we have developed a test suite for structural conformance based on the test suite for signatures in the GNU C++ compiler (Baumgartner, 1995). Changes and test suite are available from http://www.cs.luc.edu/~laufer/papers/Java/. In the remainder of this section, `JAVASRC` refers to the installation directory of the source release.

Modifying the compiler front end

We have modified the `javac` compiler to allow structural conformance for interfaces extending the new interface `java.lang.Structural`. The following files are affected:

`$JAVASRC/src/share/java/java/lang/Structural.java`

- This new file contains the empty marker interface `java.lang.Structural`.

`$JAVASRC/src/share/sun/sun/tools/java/Constants.java`

- An identifier for `java.lang.Structural` has been added.

`$JAVASRC/src/share/sun/sun/tools/java/ClassDefinition.java`

- The method `ClassDefinition.implementedBy` represents the conformance relation between classes or interfaces. This method has been modified to fall back to structural conformance if the receiver is an interface that extends `Structural`. However, interfaces that do not extend the interface `Structural` must still be implemented by name.
- The following methods have been added:

  `ClassDefinition.implementedByNameBy`

  This method checks for conformance by name and is identical to the original method `ClassDefinition.implementedBy`.

  `ClassDefinition.implementedStructurallyBy`

  This method checks that each public method of the receiver is provided by the argument. This requires first checking each public method specified in the receiver itself. Next, the method checks recursively that each interface of the receiver is also implemented by the argument.

`$JAVASRC/src/share/sun/sun/tools/javac/SourceClass.java`

- The method `SourceClass.check` now uses `ClassDefinition.implementedByNameBy` to check for cyclical interface definitions. This is necessary because `ClassDefinition.implementedBy` would allow structural conformance.

Modifying the virtual machine

In principle, no changes to the Java virtual machine (Lindholm and Yellin, 1996) should be necessary. However, conformance is checked at run time for array elements; consequently, the changes to the method `ClassDefinition.implementedBy` must be duplicated in the virtual machine. Furthermore, as of JDK 1.1, conformance is checked at run time when invoking a method through an interface; this requires additional changes to the virtual machine. The following files are affected:

`$JAVASRC/src/share/java/runtime/interpreter.c`

- The function `ImplementsInterface` has been changed to fall back to structural conformance if the interface allows it. This function gets called indirectly by the function `is_instance_of`.
- The following functions have been added in analogy to the new methods in the class `ClassDefinition`:

  `ImplementsInterfaceByName`

  `ImplementsInterfaceStructurally`
- The function `HasPublicMethod` has been added to check whether a class has a method with a given type signature.

`$JAVASRC/src/share/java/runtime/executeJava.c`

- In the function `ExecuteJava`, the branch `opc_invokeinterface_quick`, which handles the invocation of a method through an interface, checks conformance by name directly without invoking `ImplementsInterface`. This branch has been changed to fall back to structural conformance, if the interface allows it, by invoking `ImplementsInterfaceStructurally`.

Generating adapter classes

It is possible to add structural conformance to the Java compiler in such a way that the virtual machine need not be changed at all. Concretely, the modified compiler would generate an *adapter class* for each structural conformance relationship between a source class or interface and a target interface. The adapter class implements the target interface and contains an instance of the source class or interface. The methods of the target interface are forwarded to the corresponding methods of the original instance. The adapter class provides a suitable constructor and an accessor method to the instance.

For example, the following adapter class could be generated for class `QueueImpl` and interface `Queue` from the first example in Section 4:

```
final class QueueImplAsQueue implements Queue {
    final QueueImpl orig;
    public QueueImplAsQueue(QueueImpl obj) { orig = obj; }
    public QueueImpl getOrig() { return orig; }
    public Object dequeueHead() { return orig.dequeueHead(); }
    public void enqueueTail(Object x) { orig.enqueueTail(x); }
    public boolean isEmpty() { return orig.isEmpty(); }
}
```

For assigning an object of class `QueueImpl` (the source object) to an interface variable of interface type `Queue` (the target interface), the object needs to be wrapped by an instance of the adapter class. This wrapping of objects can be implemented in the compiler as a transformation on syntax trees as follows:

- Each reference conversion from the source expression to the target interface is replaced by wrapping an instance of the adapter class around the source expression. For example,

  ```
  Queue q = new QueueImpl();
  ```

  is translated to

  ```
  Queue q = new QueueImplAsQueue(new QueueImpl());
  ```

- Each cast from the target interface back down to the source class is replaced by a cast to the adapter class followed by an invocation of the accessor method. For example,

  ```
  QueueImpl qi = (QueueImpl) q;
  ```

  is translated to

  ```
  QueueImpl qi = ((QueueImplAsQueue) q).getOrig();
  ```

- Each `instanceof` expression between an object of the target interface and the source class is replaced by an `instanceof` expression between the object and the adapter class. For example,

  ```
  if (q instanceof QueueImpl) { /* ... */ }
  ```

  is translated to

  ```
  if (q instanceof QueueImplAsQueue) { /* ... */ }
  ```

Furthermore, the interface `java.lang.Structural` is meaningful only at compile time for deciding when adapter classes are needed. Therefore, any references to this interface are transformed by the compiler in the following way:

- Each occurrence of `java.lang.Structural` as the type of a variable or parameter is replaced by the semantically equivalent `java.lang.Object`.

- Each occurrence of `java.lang.Structural` in a list of interfaces implemented by a class or extended by another interface is removed from the list.

The major advantage of this approach is that the code generated by the modified `javac` compiler runs on any existing Java virtual machine, such as the ones embedded in Java-capable browsers. We are currently incorporating structural conformance based on these adapter classes into the JDK 1.1.5 source release.

A disadvantage of this approach is the large number of adapter classes needed: potentially one per interface-class pair and one per interface-interface pair. If code space is at a premium, the compiler could generate a single adapter class per interface by using Java's reflection API for implementing the forwarding methods:

```
final class QueueAdapter implements Queue {
    final Object orig;
    // ...
    public Object dequeueHead() {
        try {
            return
                orig.getClass()
                    .getMethod("dequeueHead", new Class[] { })
                    .invoke(orig, new Object[] { });
        } catch (InvocationTargetException e1) {
            throw (RuntimeException) e1.getTargetException();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
```

```
        }
        // ...
    }
```

Such a solution would optimize space at a higher cost of method invocation.

A cast from the target interface back down to the source class and an `instanceof` expression would now have to be implemented as

```
QueueImpl qi = (QueueImpl) ((QueueAdapter q).getOrig();
```

and

```
if (((QueueAdapter) q).getOrig() instanceof QueueImpl) ...
```

respectively.

Efficiency of the modified virtual machine

At compile time, the method `ClassDefinition.implementedStructurallyBy` is invoked for each occurrence of structural conformance (reference conversion, `instanceof`, or downcast). The resulting overhead per occurrence consists of checking that each method of the target interface is provided by the source class or interface. This overhead is proportional to the total number of methods and ancestors of the target interface and to the number of methods of the source class or interface. To avoid repeating the same structural conformance check, pairs of matching target interfaces and source classes or interfaces could be stored in a cache.

At run time, the virtual machine repeats the structural conformance checks that were already done by the compiler. Again, caching can be used to avoid repeated structural conformance checks. The first structural conformance check for each interface-class or interface-interface pair is, therefore, expensive. Additional structural conformance checks reduce to a table lookup in the cache, which is the same cost as testing name conformance.

As an additional optimization, the compiler could record all the structural conformance checks it performed as annotations in the byte code file. The corresponding check in the virtual machine could then be performed by the class loader instead of at run time. This would result in better timing behavior for real-time application.

Efficiency when using adapter classes

At compile time, there is overhead for each target interface and source class or interface between which structural conformance takes place in the program. This overhead consists of checking for structural conformance and generating a suitable adapter class. It is proportional to the number of methods in the classes and interfaces.

At run time, overhead is limited to the following:

- The adapter class for each target interface and source class or interface is loaded once, resulting in a small amortized overhead.
- For each reference conversion, a new instance of the corresponding adapter class is created on the heap.
- Each method invocation via the target interface is forwarded through an instance of the adapter class. This results in an additional interface method or virtual method invocation, depending on whether the type of the source expression contained in the adapter class is an interface or a class.
- Expressions involving `instanceof` do not cause additional overhead.
- For each cast down from the target interface, the enclosed instance is accessed via a `final`

method, causing minimal overhead.

6 Possible Extensions

This section discusses several possible extensions and modifications to Java that would make the language more expressive and integrate well with structural conformance. All three have the advantage of being conservative extensions. Method renaming and class import add a manageable amount of syntactic complexity to the language. Deep structural conformance, however, is based on contravariance and adds some semantic complexity.

Method renaming

Frequently, we would like to establish structural conformance to an interface when only the types, but not the names of corresponding methods match. The usual way to deal with this situation is to write a forwarding class according to the Adapter pattern (Gamma et al., 1995). We could avoid this unnecessary cluttering of the class name space if we had a mechanism for renaming methods while establishing structural conformance. Such a mechanism could be included with a cast-like notation.

The following example shows a general container interface that could be implemented by any class that allows inserting and removing an element and checking if the container is empty. If we wanted to use the `DoublyLinkedList` class as an implementation, we could rename its methods to the names used in the interface:

```
interface Queue {
    Object dequeueHead();
    void enqueueTail(Object x);
    boolean isEmpty();
}

interface Dequeue extends Queue {
    void enqueueHead(Object x);
    Object dequeueTail();
}

class DoublyLinkedList extends Vector {
    public final void addFirst(Object x) { /* ... */ }
    public final Object removeFirst() { /* ... */ }
    public final void addLast(Object x) { /* ... */ }
    public final Object removeLast(Object x) { /* ... */ }
}

DoublyLinkedList d1, d2;
// ...
Queue q5 = (Queue { dequeueHead = removeFirst, enqueueTail = addLast }) d1;
Dequeue q6 = (Dequeue { enqueueHead = addFirst, dequeueHead = removeFirst,
                        enqueueTail = addLast, dequeueTail = removeLast}) d2;

q5.enqueueTail("This string will be added at the end of the queue");
```

This example shows how structural conformance together with a renaming mechanism could be used without introducing an unwanted conformance relationship.

Importing classes for code reuse

Some classes combine the functionality of two or more existing classes and are implemented naturally by reusing the code of the existing classes. This situation occurs, for example, when combining functionality provided by library classes with user-defined class hierarchies. The technique of incorporating an existing implementation into a new class is called *mixin* and is usually expressed through a disciplined use of multiple inheritance. However, to avoid the semantic complexity of multiple inheritance as in C++ (Ellis and Stroustrup, 1990), Java provides code reuse only in the form of single inheritance. Workarounds for reusing code from more than one class in languages without multiple inheritance exist, but involve code duplication or forwarding methods.

For example, suppose that we want to provide a remote version of a class `C` from an existing hierarchy. The Java library provides the class `java.rmi.server.UnicastRemoteObject` that implements remote objects. We could define a subclass of `UnicastRemoteObject` and duplicate the methods of class `C`. Alternatively, we could define a subclass of our class that contains an instance of `UnicastRemoteObject`, to which the relevant methods are forwarded. Using structural conformance, we can retroactively provide a common interface for the existing class and the remote class, but we cannot do without code duplication or forwarding methods.

Such workarounds could be avoided if we had a language mechanism for combining multiple existing classes. Part of the complexity of multiple inheritance arises because inheritance usually entails conformance of a class to its superclass(es). Instead, a mechanism for importing code without establishing a conformance relationship would support the mixin programming style cleanly and directly. Structural conformance of an extended class and a mixin class to a common interface could then be established separately. We claim that this separation between reuse and conformance would make the language simpler yet more flexible.

Concretely, the class construct in Java could be extended to allow *importing* zero or more classes.

```
class C extends D imports E, F, ... implements I, J, ... {
    // ...
}
```

Importing a class would have the same effect as inclusion together with forwarding methods for all the public methods provided by the included class, but not by the new class. For example,

```
class RemoteChatServer extends ChatServer imports UnicastRemoteObject {
    // methods and variables of RemoteChatServer
}
```

would be equivalent to

```
class RemoteChatServer extends ChatServer {
    private UnicastRemoteObject remote = new UnicastRemoteObject();
    // forwarding methods for each method in UnicastRemoteObject:
    public Object clone() throws CloneNotSupportedException {
        RemoteChatServer theClone = new RemoteChatServer();
        theClone.remote = remote.clone();
        // ...
        return theClone;
    }
    public static void exportObject(Remote obj) {
        return UnicastRemoteObject.exportObject(obj);
    }
    // ...
    // methods and variables of RemoteChatServer
}
```

The proposed import mechanism might have the following properties, among others:

- A method defined in the new class takes precedence over imported methods with the same signature.

- An imported method may provide an implementation of an abstract method from the (abstract) superclass or from an interface.
- Java provides field access through a qualified name or by casting `this`, for example, when accessing a variable in the superclass shadowed by a variable in the subclass. Casting `this` could also be used to disambiguate between methods and variables from imported classes.
- As in Java, the receiver `super` would provide static access to methods and variables in the immediate superclass.
- Each imported class would have to provide a default constructor to enable the initialization of the resulting instance variable.

Flatt, Krishnamurthi, and Felleisen (1998) propose a mixin function construct that maps a class into an extended class. While their construct is semantically cleaner, we argue that our import construct would be easier to use and more efficient to implement. The mixin construct extends a given class in a type-safe manner. By contrast, our import construct is a pure code reuse mechanism; any desired conformance relationships may be established separately via structural conformance.

Deep structural conformance

Our definition of structural conformance requires an exact match between the signature of an overriding method and the signature of the corresponding method in the interface. This strict notion of *shallow* structural conformance sometimes gets in the way of retroactive abstraction. For example, suppose a class over which we want to abstract has a binary method, that is, a method whose argument type is the class itself. In the interface distilled from this class, we would change the argument type of the method to be that interface. However, the class would not conform to the resulting interface because the signatures of the binary method do not match exactly. (A similar mismatch would occur if the method signature contained a different class that we also want to abstract over.)

The following example illustrates this problem. Suppose we need a unifying abstraction that supports both graphical and telephone interfaces for menu-based applications. To achieve this abstraction, we would like to reduce the interfaces of the menu-related Java Abstract Window Toolkit (Sun Microsystems, 1997a) components to a least common denominator for the two platforms. Among others, we would define interfaces for menus and menu bars:

```
package common;

interface Menu extends Structural {
    // ...
}

interface MenuBar extends Structural {
    Menu add(Menu m);
    // ...
}
```

The problem is that neither of the corresponding AWT classes, `java.awt.Menu` and `java.awt.MenuBar`, conform structurally to our `common.Menu` and `common.MenuBar` classes, respectively. The reason is that the argument type of `common.MenuBar.add` is `common.Menu`, while the argument type of `java.awt.MenuBar.add` is `java.awt.Menu`.

It is possible to fix this problem by using *deep structural conformance* instead of shallow structural conformance. This form of conformance would no longer require an exact match between the type signatures of the overriding and overridden methods, but instead would require the type of the overriding method to conform to the type of the overridden method (Amadio and Cardelli, 1993).

Deep structural conformance has two disadvantages. First, since it is based on contravariance, it may be less intuitive than shallow conformance. Second, the rules for method overriding under deep conformance are not consistent with the current rules for method overloading.

7 Conclusion

We have presented an extension of the Java language that allows structural conformance to interfaces. Any source class or interface that declares or implements each method in a target interface conforms structurally to the interface, and any instance of the source type can be used where a value of the target type is expected, without having to declare conformance of the class to the interface.

We have argued that structural conformance makes the type system much more flexible in situations that require retroactive abstraction over types, and we have given examples to support this view.

Furthermore, our proposed extension requires no additional syntax and only small modifications to the Java compiler and optionally, for performance reasons, the virtual machine, resulting in a minor performance penalty. Our extension is conservative: Existing Java programs still compile and run in the same manner as under the original language definition. Our extension is typesafe: A program without casts that compiles without errors will not have any type errors at run time. Finally, our extension works well with Java's remote method invocation (Sun Microsystems, 1997c).

A minor drawback of structural conformance is the performance penalty, as discussed in Section 5. At compile-time, performing structural conformance checks and, optionally, creating adapter classes results in a minor overhead. If the virtual machine is modified, the run-time overhead is relatively small. It is mostly due to performing structural conformance checks, which can be moved out of loops into the class loader. If adapter classes are used to avoid modifying the virtual machine, the run-time overhead is higher. The reason is that memory needs to be allocated for the adapter objects and that there is an extra indirection for method calls. In either case, however, the performance penalty only has to be paid if structural conformance is actually used.

References

Amadio, R. M. and Cardelli, L. (1993). Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631.

America, P. and van der Linden, F. (1990). A parallel object-oriented language with inheritance and subtyping. In *Proceedings of the OOPSLA '90 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 161–168, Ottawa, Canada. Association for Computing Machinery. *ACM SIGPLAN Notices*, 25(10), October 1990.

Baumgartner, G. (1995). C++ signature and class-scoping tests. Included in the Cygnus GNU C++ test suite. Available from ftp://ftp.cygnus.com/pub/g++/.

Baumgartner, G. and Russo, V. F. (1994). Implementing signatures for C++. In *Proceedings of the 1994 USENIX C++ Conference*, pages 37–56, Cambridge, Massachusetts. USENIX Association.

Baumgartner, G. and Russo, V. F. (1995). Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software—Practice & Experience*, 25(8):863–889.

Baumgartner, G. and Russo, V. F. (1997). Implementing signatures for C++. *ACM Transactions on*

Programming Languages and Systems, 19(1).

Cleeland, C., Schmidt, D. C., and Harrison, T. (1997). External polymorphism – an object structural pattern for transparently extending C++ concrete data types. Submitted to Pattern Languages of Programming Languages, Vol. 3.

Connor, R. C. H., Brown, A. B., Cutts, Q. I., Dearle, A., Morrison, R., and Rosenberg, J. (1990). Type equivalence checking in persistent object systems. In Dearle, A., Shaw, G. M., and Zdonik, S. B., editors, *Implementing Persistent Object Bases, Principles and Practice*, pages 151–164. Morgan Kaufmann. Available from http://www-fide.dcs.st-and.ac.uk/Publications/1990.html#type.equiv.

Cook, W. R. (1995). Array subclassing and IncompatibleTypeException. Posted to the java-interest@java.sun.com mailing list. Available from http://java.sun.com/archives/java-interest/0463.html.

Ellis, M. and Stroustrup, B. (1990). *The Annotated C++ Reference Manual*. Addison-Wesley.

Flatt, M., Krishnamurthi, S., and Felleisen, M. (1998). Classes and mixins. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages (POPL)*, San Diego, CA.

Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. M. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts.

Gosling, J., Joy, B., and Steele, G. (1996). *The Java Language Specification*. Sun Microsystems, Mountain View, California, 1.0 edition. HTML version available from http://java.sun.com/docs/books/jls/.

Granston, E. D. and Russo, V. F. (1991). Signature-based polymorphism for C++. In *Proceedings of the 1991 USENIX C++ Conference*, pages 65–79, Washington, D.C. USENIX Association.

Jenks, R. D. and Sutor, R. S. (1992). *AXIOM: The Scientific Computation System*. Springer-Verlag, New York, New York.

Lindholm, T. and Yellin, F. (1996). *The Java Virtual Machine Specification*. Sun Microsystems, Mountain View, California. HTML version available from http://java.sun.com/docs/books/vmspec/.

Morrison, R., Connor, R., Kirby, G., and Munro, D. (1996). Can Java persist? In *Proc. Persistent Java Workshop*, Scotland.

Snyder, A. (1986). Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–45, Portland, Oregon. Association for Computing Machinery. *ACM SIGPLAN Notices*, 21(11), November 1986.

Sun Microsystems (1997a). *Java API Documentation*. Mountain View, California. Current version available from http://java.sun.com/products/jdk/1.1/docs/api/packages.html.

Sun Microsystems (1997b). Java Developers Kit Release 1.1.5. Available from http://java.sun.com/products/jdk/1.1/.

Sun Microsystems (1997c). Java Remote Method Invocation. Documentation available from http://java.sun.com/products/jdk/rmi/.