Deep Conformance in Java Layers

Richard Cardone    Calvin Lin
Department of Computer Sciences
University of Texas at Austin
{richcar, lin} @cs.utexas.edu

September 23, 2000

Abstract
Java Layers supports a programming methodology of stepwise refinement in which feature implementations are encapsulated in separate classes and then these classes are composed to build applications. Inheritance, using mixins, provides a practical way to implement Java Layers. This specification describes two enhancements to Java's inheritance mechanism that allow stepwise refinement to work with nested types with greater expressiveness and with better static type checking.

1 Introduction
Layer composition in Java Layers (JL) is based on the use of mixins [2,3]. Mixins are types whose supertypes are parameterized. Before describing how mixins are used in JL, we briefly review why mixins are useful in general.

1.1 Mixins
Mixins are useful because they allow a set of classes to be specialized in the same manner, with the specializing code residing in a single class definition. For example, suppose we wish to extend three unrelated classes—Car, Chest and House—to be "lockable" by adding two methods, lock() and unlock(). Without mixins, we would define subclasses of Car, Chest, and House that each extended their respective superclasses with the lock() and unlock() methods. The lock code would be replicated in three places. With mixins, we could instead write a single class called Lockable that would extend any superclass, and we could instantiate the Lockable class to extend Car, Chest, and House. The lock() and unlock() methods would only be defined once. In JL syntax, the Lockable mixin would be defined as follows:

```java
class Lockable<T> extends T {
    public lock(){...}
    public unlock(){...}
}
```

Lockable<Car>, Lockable<Chest> and Lockable<House> are three instantiations of Lockable.

1.2 Using Mixins in JL
The key idea behind the JL component model is that each layer encapsulates exactly one design feature, which is a high-level requirement that defines some application attribute or capability. In JL, code that implements a single feature is encapsulated inside a single class, usually a mixin class. Applications are built by composing, or mixing and matching, the classes that implement the required set of features.

In JL, interfaces can be used as type parameter constraints to restrict which classes can be composed. For instance, the type parameter that represents a mixin’s superclass can be constrained to implement a
specified set of interfaces. This allows the mixin developer to make assumptions about the inherited structure and behavior of the mixin. The Lockable mixin from the previous section is modified below to only accept physical objects (interface not shown) as actual type parameters:

```java
class Lockable<T implements PhysicalObjectIfc> extends T {
    public lock(){...}
    public unlock(){...}
}
```

Mixins promote JL’s programming methodology of stepwise refinement in which classes are built up feature by feature, eventually constructing whole applications. New classes are composed using type equations, which another form of parameterized type instantiation in JL. A type equation usually starts with a non-mixin base or anchor class, which is then composed with some number mixins to extend basic functionality. Figure 1 depicts a base class, a mixin class, and a type equation that creates a new class using the base and mixin classes.

```
// Base node class.
class Node
{
    Node _next, _prev;
    Node getNext(){return _next;}
    Node getPrev(){return _prev;}
    ...}

// Mixin that records each time a node is traversed.
class TraceTraversal<T extends Node> extends T
{
    Node getNext()
    {
        Node next = super.getNext();
        record(next);
        return next;
    }

    Node getPrev()
    {
        Node prev = super.getPrev();
        record(prev);
        return prev;
    }
}

// Type equation creating class C.
class C = TraceTraversal<Node>;
```

**Figure 1 - Simple Type Equation**

In the type equation for class C, TraceTraversal is guaranteed to be a subclass of Node because of the constraint on its type parameter T. A subclass of Node could have been used in the type equation instead of Node itself. In either case, TraceTraversal can safely call its superclass’s getNext() and getPrev() methods. The use of type parameter constraints in TraceTraversal means that references to superclass methods can be type checked at compile time. Note that this static check could have been enforced using an interface constraint, but a class constraint was used to simplify the example.

The example in Figure 1 shows that mixin composition is enhanced when guarantees about superclass type signatures can be statically made. Composition is enhanced because more type checking is performed before compositions are defined, increasing the likelihood that compositions will succeed. This is one of the main benefits of constrained parametric polymorphism. We now describe how we lose the ability to perform this kind of early type checking in Java when nested types are involved.

### 1.3 Mixin Layers

Mixin layers [3] are mixins that have nested types. The example in Figure 1 in the last section shows how a mixin can modify multiple methods from the same class. Mixin layers also use inheritance and are able to modify methods from multiple classes. Figure 2 shows a nested base class, a mixin layer, and a type
equation that creates a new class. The `TraceCreation` code shows how a mixin layer affects the code of more than one class; `TraceCreation` implements its tracing feature by modifying both the nested `Node` and `Container` classes.

```java
// Nested base class.
class Collection {
    class Node {Node(); …}
    class Container {Container(); …}
}

// Mixin layer.
class TraceCreation<T extends Collection> extends T {
    class Node extends T.Node {
        Node() {super.Node(); recordNodeCreation();} …}
    class Container extends T.Container {
        Container() {super.Container(); recordContainerCreation();} …}
}

// Type equation.
class D = TraceCreation<Collection>;
```

**Figure 2 - Mixin Layer**

In Java, subtypes do not inherit their supertype’s nested types. That is, inheritance does not propagate a supertype’s nested classes or nested interfaces, just as inheritance does not propagate constructors. Thus, the mixin layer in Figure 2, `TraceCreation`, cannot be completely type checked at compilation time. Even though the actual parameter to `TraceCreation` must extend the `Collection` class, we don’t know if the actual parameter contains the `Node` and `Container` nested types.

Imagine, for example, that a shallow subclass of `Collection`—one that does not subclass `Collection`’s nested classes—is used as the actual parameter to `TraceCreation` in the type equation for class `D`. The generation of class `D` would fail because supertype references `T.Node` and `T.Container` in `TraceCreation` would not be resolved. We are not able to catch this problem when `TraceCreation` is compiled, but instead have to wait until `TraceCreation` is used in a composition. Of course, the actual example in Figure 2 does not suffer from this problem because `Collection` itself is used as the type parameter.

The existence of nested types in superclasses can only be guaranteed at compile time if a more precise form of inheritance is used, a form that explicitly specifies whether nested types are inherited or not. As in the case of non-nested mixins, the ability to compose mixin layers is enhanced when static type checking is improved. Better class composition is the impetus for defining deep type conformance; we now describe JL’s implementation of deep type conformance.

## 2 Deep Processing

JL introduces the `deeply` modifier in class and interface definitions to support *deep processing*. Deep processing implements the following two properties defined by Smaragdakis [3]:

- **Deep Subtyping**\(^1\) – Type C is a *deep subtype* of another type B if C is a subtype of B, and for every publicly accessible nested type B.N, there is a publicly accessible type C.N that is a deep subtype of B.N.
- **Deep Interface Conformance** – Class C *conforms deeply* to interface I if C implements I, and for each publicly accessible nested interface I.N, there is a publicly accessible class C.N that conforms deeply to I.N.

\(^1\) Deep Subtyping is a slight generalization of Deep Subclassing in that it encompasses Java interfaces.
The `deeply` keyword can modify an extends clause by appearing immediately after the extends keyword. When an extends clause is used to specify the supertype(s) of a class or interface, using `deeply` means that the class or interface being defined must be a deep subtype of its supertype(s). When the extends clause appears in a parametric type constraint, using `deeply` requires that any actual type parameter is a deep subtype of the named supertype(s). Figure 3 shows the use of `deeply` in the extends clauses of a class, an interface and a mixin.

```
class C extends deeply Super {}  
interface I extends deeply I1, I2 {}  
class D<T extends deeply Super> extends deeply T {}  
```

**Figure 3 - Deeply in Extends Clauses**

The `deeply` keyword can also modify an implements clause by appearing immediately after the implements keyword. When an implements clause is used in a class definition to specify a supertype(s), using `deeply` means that class being defined must deeply conform to the listed interface(s). When the implements clause appears in a parametric type constraint, using `deeply` requires that any actual type parameter deeply conform to the named interface(s). Figure 4 shows the use of `deeply` in the implements clauses of a two classes.

```
class C implements deeply I1, I2 {}  
class D<T implements deeply I1, I2, I3> {}  
```

**Figure 4 - Deeply in Implements Clauses**

The `deeply` keyword can also be used more selectively in extends or implements clauses by appearing after individual supertype names. In this usage, the deep property is specified only for types followed by `deeply` in the extends or implements clauses. In Figure 5, `deeply` is applied only `I2` in the class and interface definitions.

```
class C implements I1, I2 deeply, I3 {}  
interface I extends I1, I2 deeply, I3 {}  
```

**Figure 5 - Selective Use of Deeply**

Deep Subtyping and Deep Interface Conformance preserves nested names at every level within inheritance hierarchies. Public nested types that appear in supertypes are propagated in deep subtypes. When `deeply` is used in non-constraint clauses, the JL compiler will automatically generate any missing nested types in the subtype being defined to preserve deep structure. This alleviates the programmer from having to copy into the subtype the parts of the supertype’s deep structure not being modified.

JL also introduces the `propagate` modifier to allow deep processing to be selectively applied to non-private nested types. By default, the `deeply` keyword initiates deep processing only on public nested types. Nested types with protected or package scope are deeply processed if they are explicitly marked `propagate`. `propagate` can also be used to deeply process interface definitions nested within class definitions and class definitions nested within interface definitions. `propagate` works only in conjunction with `deeply`—if `deeply` is not specified, deep processing does not occur.

The `deeply` and `propagate` keywords could be added to standard or parameterized Java as a standalone extension. Deep processing does not rely on any other JL extension and has meaningful semantics in any type hierarchy that includes nested types. A type of deep processing similar to that described in this section was implemented in the first version of Java Layers [4].

### 3 References

3. Y. Smaragdakis. *Implementing Large-Scale Object-Oriented Components*. Ph.D. dissertation, University of Texas At Austin, Department of Computer Sciences, December 1999.