Java Native Interface Idioms for C++ Class Hierarchies

This paper appeared in the journal *Software—Practice and Experience*, 2000; 30:1641-1660, December, 2000.

Dale Parson, Ph.D.
Bell Laboratories, Lucent Technologies
1247 South Cedar Crest Blvd.
Allentown, Pa. 18103
dparson@lucent.com
610-712-3365
fax: 610-712-3940

Zhenyu Zhu
Lehigh University
232 West Packer Avenue
Bethlehem, Pa. 18015
zhz2@lehigh.edu

SUMMARY

The Java™ Native Interface (JNI) provides a set of mechanisms for implementing Java methods in C or C++. JNI is useful for reusing C and C++ code repositories within Java frameworks. JNI is also useful for real-time systems, where compiled C/C++ code executes performance-critical tasks, while Java code executes system control and feature tasks. Available JNI literature concentrates on creating Java proxy classes that allow Java clients to interact with C++ classes. Current JNI literature does not discuss Java proxies for entire C++ inheritance hierarchies; that is the topic of this paper. Our experience in reusing C++ class hierarchies within a Java framework has uncovered a set of useful techniques for constructing Java proxy class hierarchies that mirror their C++ counterparts.

This report gives both high level design guidelines and specific programming idioms for constructing Java class hierarchies that serve as proxies for C++ counterparts. We begin by discussing opportunities for reuse within a proxy class hierarchy, as well as problems caused by differences between the Java and C++ approaches to inheritance. The two most significant differences are due to C++ support for invocation of a member function based on the static type of its class, and C++ support for multiple implementation inheritance. Two example C++ class hierarchies provide the basis for a set of sections that present the design guidelines and that codify the programming idioms. This work could serve as the basis for an automatic generator of Java proxy class hierarchies.

Keywords: Java Native Interface, C++, class hierarchy, proxy class, design pattern

JAVA PROXIES FOR C++ CLASS HIERARCHIES

Java Native Interface for C++ reuse and run-time efficiency

The *Java Native Interface* [1] is a set of software mechanisms that support interaction between Java modules on one side of JNI and C/C++ modules on the other. A Java method declaration marked with the keyword *native* is identified as a native method written in C or C++. Within a Java source file a native method appears strictly as a method signature declaration; it contains no Java code. Invoking the generator program *javah* with a compiled Java class as an argument creates a C/C++ function declaration equivalent to each native method declaration. Each generated declaration serves as the starting point from which a programmer can write a C or C++ implementation of its method. Native methods can construct, delete and interact with C data structures and C++ objects. A Java system can dynamically load native methods. Conversely, a C or C++ system can construct a Java Virtual Machine (JVM) and interact with it via JNI.

Many current Java software projects require the services of JNI at some point in their lives. Object-oriented engineering depends on reusing mature software modules, and many mature software modules have been written in C++. Despite the fact that JNI programming does not have the portability of *100% Pure Java* [2], JNI remains important for the masses of developers with useful C++ code in their repositories.

Reuse of legacy C++ code is not the only reason to use JNI. Performance-critical embedded communications systems promise to remain bilingual. Modern communications systems often use C and C++ modules running on microcontrollers to manage system control and user features, while using assembly language modules running on digital signal processors to manage communications data streams. Microcontroller architectures have begun to add Java support [3] at the same time that fixed-point digital signal processors are improving their run-time

support for C and C++ [4]. We anticipate a “lane shift” in communications systems that begin to use Java for microcontroller modules and C++ for signal processing modules.

Our own need for JNI is the result of redesigning the upper layers of an embedded system simulator and debugger [5] to take advantage of Java user interface, networking, threading, dynamic loading and general library infrastructure. Stable lower layers of this framework, for example processor simulation models and hardware interface drivers, already exist in a C++ code base. We are engaging JNI integration of several legacy C++ class hierarchies, and this article summarizes important lessons learned from our experience.

Java-to-C++ proxy class hierarchies: reuse opportunities and problems

There are several useful sources of JNI programming information available [6,7,8], all of which concentrate on wrapping a single C++ class with a Java proxy class. The Proxy Design Pattern interposes a proxy class between client code and a target class used by that client code [9]. A *remote proxy* provides access to a target object in a different address space. A *virtual proxy* creates expensive target objects on demand. A *protection proxy* controls client access to a target object. A JNI-based Java proxy for a C++ class can fulfill all of these roles. It acts as a remote proxy by mapping Java object references and method calls to C++ object pointers and method calls [7]. It acts as a virtual proxy whenever it buffers, within the Java object, results from queries to the target C++ object. Buffering eliminates repeated JNI call overhead for repeated client access to stable data. It acts as a protection proxy by providing access only to public methods and data fields of the target C++ object. Java client code typically does not require access to private and protected C++ methods and fields; these methods and fields are part of the implementation of the target class, not part of the implementation of the proxy.

We have found no reference that focuses on mapping a C++ class hierarchy to an

equivalent Java class hierarchy. While basic JNI mechanisms and Java proxy classes are necessary elements in reusing a C++ class hierarchy under Java, they are not sufficient. There are patterns of JNI usage that become evident only when reusing an entire C++ class hierarchy. A *Java proxy class hierarchy* adds mechanisms along two dimensions to the mechanisms of simple proxy classes. The first dimension is one of *intra-proxy reuse*. It arises because the existence of a Java class hierarchy that mirrors a C++ hierarchy gives the Java class designer an opportunity to reuse proxy mechanisms within the Java hierarchy. Every Java proxy class that is not part of an inheritance hierarchy must provide itself with all necessary JNI mechanisms such as native methods and C++ object references. A Java proxy class that is part of a proxy class hierarchy, on the other hand, can rely on a proxy base class to provide most of its JNI mechanisms. Intra-proxy reuse represents a set of opportunities rather than a set of problems. Intra-proxy reuse organizes proxy class design in the following ways:

- Common code for manipulating a C++ target object goes into a proxy base class. For a given Java proxy object there is only one C++ target object, no matter how many C++ classes are part of its definition, and no matter how many Java classes access the target. Proxy code to manage this object resides in a common proxy base class.
- A Java method that is a proxy for a C++ virtual function requires definition at only one place in its proxy class hierarchy, and that is in the proxy class that corresponds to the most basic, least derived C++ class in which the virtual function appears. Even though virtual function redefinition may occur in derived C++ classes, the Java proxy hierarchy need not redefine its methods because C++ virtual function invocation always selects the correct, most derived function definition. There is no need for the Java proxy hierarchy to duplicate work already performed by C++ virtual function invocation.
- With respect to flavors of inheritance, proxy class hierarchies provide *interface inheritance* [9] for their corresponding C++ class capabilities, and they provide

implementation inheritance for their proxy implementation mechanisms. Each derived proxy class adds a new method signature for each new public virtual function introduced by its target C++ class, but the proxy plays no part in implementing the semantics of the method. Each proxy class does, however, participate in implementation inheritance of proxy mechanisms built up in ancestor proxy classes. Implementation inheritance of code to manage the C++ object and implementation inheritance of JNI-based proxy method definitions are two examples already given. Other examples include inheritance-based access to constructors that initialize proxy base class management of the C++ object, and implementation inheritance of a *finalize* method that calls a C++ destructor.

The second dimension of complexity added in going from singleton proxy classes to proxy hierarchies is caused by *programming language divergence*. C++ provides a number of inheritance-related features that have no direct counterparts in Java. Unlike the first dimension that offers opportunities for reuse, this dimension presents problems to be solved:

- C++ provides non-virtual functions [10]. Unlike virtual functions and unlike Java methods, the static type of client code's C++ object pointer or reference determines the function that is invoked. Derived classes may redefine non-virtual functions, but function invocation is based on the static type of the invoking pointer or reference, not on the dynamic type of the target object. This issue does not arise with a singleton class proxy, because only one non-virtual function variant is visible when considering only a single class.
- C++ allows client code to use the scope resolution operator "::" to override virtual function invocation. An example is the use of "object->foo::method()" to force the invocation of "method" as defined statically for class "foo," regardless of the actual, possibly derived class of "object." Static resolution of a virtual function is similar to resolution of a non-virtual function based on pointer type, in that it allows client code to select the function to invoke. In contrast, Java does not provide any means for client code to override dynamic method selection. A

Java method definition may invoke its parent's definition of "method" by invoking "super.method()," but this limited form of method selection based on static type applies only to derived methods within an inheritance hierarchy [11].

- C++ supports implementation inheritance from multiple base classes, while Java supports only single implementation inheritance. Both languages support multiple interface inheritance, where a Java interface is equivalent to an abstract C++ class whose functions are all *pure virtual* and whose fields are all constant values [10]. Mirroring multiple implementation inheritance in a Java proxy class hierarchy requires using a combination of single implementation inheritance and multiple interface inheritance combined with method delegation.
- Extension of target C++ class hierarchies may result in fusion of two or more formerly independent hierarchies into a single class via multiple implementation inheritance. Fusion can occur some time after Java proxy hierarchies for the independent C++ hierarchies have been designed. Designing Java proxy hierarchies that are flexible in adapting to potential cases of multiple inheritance is more complex than designing Java proxy hierarchies that deal only with known, static cases of multiple inheritance. This flexibility forces us to consider *second-order proxies*, where Java interfaces serve as proxies to Java classes that serve as proxies to C++ classes. Java's support of multiple inheritance for Java interfaces gives a basis for architecting an interface-based proxy strategy with the required flexibility.

Our approach to JNI for C++ class hierarchies uses a set of programming idioms to achieve intra-proxy reuse and to solve these programming language divergence problems. Programming idioms solve detailed design problems such as the relationships of proxy constructors to their ancestor proxy constructors, relationships of proxy constructors to C++ target object constructors, relationships of Java methods to virtual, non-virtual, and statically resolved C++ member functions, and relationships of Java interface and implementation inheritance to C++ multiple implementation inheritance. Given a hierarchy of C++ classes with public member

functions and public data fields, it would be possible to apply these programming conventions automatically to generate a hierarchy of Java proxy classes. The analysis reported in this paper constitutes an important step towards that automation.

JAVA NATIVE INTERFACE MECHANICS

This section makes the discussion of Java proxy classes concrete by giving a small example of a JNI native method. Listing 1 shows one native method declaration, that of *nativeSend*, for a Java proxy class called *JgenericIPC*. As the next section discusses more thoroughly, C++ class *genericIPC* is a class involved in inter-process communication. Its *send* virtual function accepts an ASCII string as an argument and it communicates this string over a communication channel. JgenericIPC's nativeSend is a proxy method that accepts two arguments, a Java String for send and a Java long "ptr" parameter that is in fact a pointer to a C++ genericIPC object, type cast to a Java long for use by Java. An upcoming discussion of C++ constructors and JNI factory methods shows how to initialize this pointer as a Java long.

The signature of nativeSend in Listing 1 includes the *native* keyword and an empty function body. Applying the *javah* utility to the compiled JgenericIPC class file generates a function stub for each native method. The stub for nativeSend includes only the comment lines and the "Java_JgenericIPC_nativeSend" C++ signature declaration of Listing 1. A programmer adds the rest by hand.

Java_JgenericIPC_nativeSend performs four steps: it converts the Java UNICODE String parameter to a null-terminated ASCII string, it type casts the Java long pointer parameter to a genericIPC object pointer, it calls this object's C++ send method, and it frees the ASCII string. Other parameters for Java_JgenericIPC_nativeSend give access to the Java run-time environment and the JgenericIPC proxy object. These parameters are useful for accessing proxy object fields,

for invoking Java methods, and for constructing Java objects used as return values and Java method arguments.

JNI C++ code tends to be rather cryptic and stereotyped, concerning itself mostly with JNI library utilities for converting objects between Java-compatible and C++-compatible types. Intra-proxy reuse of JNI code helps to minimize the amount of this cryptic code that a programmer must write.

AN EXAMPLE C++ CLASS HIERARCHY

We now turn to the problem of mapping C++ class hierarchies into Java hierarchies. This section starts the discussion with an example C++ hierarchy that highlights most of the intra-proxy reuse and language divergence issues that we have outlined. We save issues of C++ multiple inheritance for a later example.

Figure 1 illustrates our example C++ class hierarchy in the Unified Modeling Language (UML) graphical notation [12]. We use UML because it is a standard notation that is less verbose than code examples. UML attributes such as genericIPC's *buffer* correspond to Java and C++ member fields, and UML methods such as genericIPC's *send()* correspond to Java and C++ methods.[1] We assume the reader has a working knowledge of Java and C++ constructs.

Figure 1 is a simplification of a real class hierarchy in our embedded system tools framework. The hierarchy uses abstract base class *genericIPC* to manage a transport mechanism-neutral communications protocol. This base class organizes messages and interacts with remote clients and servers in managing transactions, but it defers binding a communications transport mechanism to derived classes. Class genericIPC defines a number of abstract transport methods,

1. Figure 1 uses the UML "+" symbol to denote public attributes and methods, "#" to denote protected attributes and methods, and "-" to denote private attributes and methods.

represented by *send* and *recv* in Figure 1. A derived class implements these methods using medium-specific mechanisms. Figure 1 shows concrete class *socketIPC*, a class that uses TCP/IP sockets within the transport methods. An alternative derived class sharedMemoryIPC (not shown) could use shared memory for the transport mechanism. Figure 1 also shows class *secureSocketIPC*, derived from socketIPC, that wraps the send and recv methods of socketIPC with security audits.

The toy example of Figure 1 retains the class hierarchy of the original classes, but it implements only simple string storage and method call tracing. There are several points in this example that have ramifications for Java-JNI proxy classes.

- genericIPC is abstract — no direct construction of a genericIPC object is possible — while socketIPC and secureSocketIPC are concrete.
- genericIPC introduces abstract methods send and recv.
- genericIPC introduces concrete methods zapBuffer and printTrace and field buffer. Therefore inheritance from genericIPC constitutes *implementation inheritance*. Derived classes inherit methods and data in addition to method signatures.

A SINGLE INHERITANCE JAVA PROXY CLASS HIERARCHY

Overview of a Java proxy hierarchy

Figure 2 illustrates a Java proxy class hierarchy and its relationships to the C++ hierarchy of Figure 1. There are one-to-one correspondences between Java classes JgenericIPC, JsocketIPC and JsecureSocketIPC and their C++ counterparts genericIPC, socketIPC and secureSocketIPC. Each solid arrow signifies a reference from a Java object to a C++ object or to another Java object. This example contains no references from C++ to Java proxy objects because the proxy classes wrap existing C++ classes that encode no knowledge of Java. For designs that include navigation

from C++ objects to Java objects, the JNI code would maintain a C++-to-Java object reference map as discussed in Reference 7.

A proxy base class for C++ target object access

Figure 2 shows that, while genericIPC is the base class of the C++ class hierarchy, the Java proxy hierarchy is rooted at JgenericIPC's parent, *JNIbase*. JNIbase is our abstract Java class that manages any proxy hierarchy's reference to its C++ object. JNIbase defines a nested helper class *cppptr* that abstracts the notion of a C++ pointer. The main role of cppptr is one of hiding the implementation of a C++ pointer as a type-cast Java long value. Cppptr is a thin *protection proxy.* Java client code that interacts with C++ classes does so via proxies such as JgenericIPC, and these proxies in turn interact with cppptr. It hides the implementation of a C++ pointer as a Java long, thereby disallowing long value operations, and it exposes this implementation only at the point that a call to a native method occurs.

JNIbase itself houses a cppptr object that it receives via a constructor parameter. JNIbase can supply this object to derived class native methods via method getCppObj(). The *protected* level of access for JNIbase's constructor and getCppObj() restricts exposure of the C++ implementation to within the proxy hierarchy.

Proxy object construction poses a problem for execution order. A C++ client can construct a socketIPC or a secureSocketIPC object. There can be no construction of a genericIPC object as such because genericIPC is an abstract class. Similarly, a Java client can construct a JsocketIPC or JsecureSocketIPC proxy object that, in turn, must construct its C++ target object. Therefore constructor execution for JsocketIPC must construct a C++ socketIPC object for its cppptr, and JsecureSocketIPC must construct a C++ secureSocketIPC object for its cppptr, but in Java as in C++, base class constructors execute before derived class constructors. How is it possible for

JsocketIPC or JsecureSocketIPC to construct its C++ target object before passing a cppptr to JNIbase's constructor, given the fact that JNIbase's constructor runs first?

The answer comes with a programming idiom for class-static *factory methods* [9] and protected *inheritance constructors*. The C++ object factories FactoryJsocketIPC and FactoryJsecureSocketIPC of Figure 2 are class-static Java methods that take arguments corresponding to their counterpart C++ constructor arguments. The factories are native methods that construct their respective C++ objects, and return pointers to these objects as type-cast Java long values. The significance of the class-static qualification of these methods is the fact that they do not depend on Java proxy objects, and therefore their execution order is independent of constructor execution order. A derived class constructor can invoke its factory before its base class constructor runs. The significance of the *private* protection level of a factory is that only its proxy class can invoke it, and in fact only the constructor of its proxy class invokes it.

The public constructor for JsecureSocketIPC, for example, takes a string argument, as does its factory method and its C++ counterpart constructor. We call this public constructor the *proxy constructor*. This proxy constructor initializes its cppptr and passes it to base class JsocketIPC with the following line of code:

```
super(new cppptr(FactoryJsecureSocketIPC(startmsg)));
```

This line, which is the entire body of JsecureSocketIPC's proxy constructor, does three things. It invokes the static factory method which invokes the C++ constructor on its argument, it encapsulates the returned C++ pointer inside a cppptr object, and it passes the cppptr object to its base class *inheritance constructor*. Each class that inherits from JNIBase has an inheritance constructor, and its sole job is to accept a parameter "cppptr thisobj" and pass it to its base class inheritance constructor with the line:

super(thisobj);

Eventually the cppptr reaches the JNIbase constructor that stores it in the root of the proxy hierarchy. Note that unlike the public proxy constructors for JsocketIPC and JsecureSocketIPC in Figure 2, the inheritance constructors for JgenericIPC, JsocketIPC and JsecureSocketIPC have a *protected* protection level. They exist solely to pass the cppptr from derived through base classes to JNIbase.

The singular nature of JNIbase's cppptr object within the proxy class hierarchy of Figure 2 raises an issue about the one-to-one reference associations from the Java classes to their C++ counterparts, signified by the arrowed lines. There is in fact only one physical link from a proxy Java object to a C++ object, the cppptr object stored in JNIbase. The other reference associations in Figure 2 are class-specific interpretations of this link. Recall from Listing 1, for example, the native type cast:

genericIPC *cppobj = (genericIPC *) cppptr ;

At this point a JgenericIPC native method is interpreting the cppptr as a genericIPC pointer. A JsocketIPC native method could interpret the same cppptr value as a socketIPC pointer, and if the target object is a secureSocketIPC object, then a JsecureSocketIPC native method could interpret the cppptr value as a secureSocketIPC object. The only means by which a JsecureSocketIPC native method receives access to a cppptr is from its JsecureSocketIPC object, and the only means by which a JsecureSocketIPC object receives access to a cppptr is from its class-specific factory method or from its inheritance constructor as invoked from a derived class. Therefore, when a JsecureSocketIPC native method type casts its cppptr to a secureSocketIPC pointer, it is assured that the cast is safe.

Proxy method calls to C++ virtual functions

In the proxy classes that we have built so far, we have brought only *public* C++ member functions out to Java proxy classes via native methods. We have no public data members, but if we did, proxy access would take the form of public *get* and *set* methods for those member fields. We do not bring *private* members to Java proxy classes because there is no point in doing so. A C++ private member is part of the implementation of its class, but a proxy is interested only in providing access for clients. Clients cannot use private members.

An enhancement to our current approach would connect *protected* C++ members to Java proxies, and there may be some C++ class hierarchies for which protected member access is useful. The clients of protected members are derived classes. While our work has been limited to strictly proxy Java hierarchies, it is certainly possible to extend a Java proxy class by adding non-proxy, application-oriented capabilities in a derived class written entirely in Java. In this case the services offered by C++ protected members should be brought to the Java proxy while maintaining a *protected* level of protection. The mechanics of building protected proxy methods are the same as those for public proxy methods.

In the introduction to intra-proxy reuse we stated that a Java method that is a proxy for a C++ virtual function requires definition at only one place in its proxy class hierarchy, and that is in the proxy class that corresponds to the most basic, least derived C++ class in which the virtual function appears. Take C++ virtual function *send* in Figure 2 as an example. It first appears in class genericIPC, and it is redefined in both socketIPC and secureSocketIPC. Despite the fact that genericIPC::send is a pure virtual function — it is a virtual function declaration without any body — abstract proxy class JgenericIPC can implement proxy method JgenericIPC.send, and no subsequent redefinition is needed in JsocketIPC or JsecureSocketIPC.

Consider the invocation of C++ *send* from method nativeSend of Listing 1:

genericIPC *cppobj = (genericIPC *) cppptr ; cppobj->send(ascii) ;

Although cppobj is a genericIPC pointer, we are guaranteed that it does not point to a strictly genericIPC object, because there can be no strictly genericIPC object. Members of genericIPC exist only as part of a concrete, derived class object. Therefore “cppobj->send(ascii)” is guaranteed to invoke a genuine virtual function. Furthermore, it invokes the correct function because the identity of that virtual function is established at the time that the JNI factory method constructs the C++ object. C++ object construction builds virtual function dispatch machinery that invokes the correct virtual function for the object’s dynamic class. Since C++ has already accomplished this job, there is no need to duplicate it in the Java hierarchy. The illustrations of JsocketIPC and JsecureSocketIPC highlight the savings. They consist only of the single-line proxy constructors and inheritance constructors already discussed, along with the class-static factory methods that provide the link to this C++ virtual function invocation machinery.

Private native method parameters avoid complicated JNI library calls

Figure 2 shows that for the three public virtual functions first defined in genericIPC — *zapBuffer*, *send* and *recv* — proxy class JgenericIPC defines both public proxy methods zapBuffer, send and recv, and private native methods *nativeZapBuffer*, *nativeSend* and *nativeRecv.* We have adopted the idiom of implementing proxy methods in pairs, consisting of a public Java proxy method that invokes a private native method. For example, JgenericIPC.send is defined as follows:

public void send(String str) { nativeSend(str,getCppObj().getPtr()); }

When we first began using JNI we would declare native methods at the public class interface, but with a little experience we decided to make every native method private and to add a

value parameter for each object field that a native method uses. Passing field values via method parameters eliminates native method calls to the following JNI library functions for every field:

- GetObjectClass — retrieves the class of an object.
- GetFieldID — retrieves a field identifier for a given class.
- GetTYPEField — retrieves a field value for a given field identifier and object, where TYPE is one of Object, Boolean, Byte, Char, Short, Int, Long, Float or Double.

The C++ object address is a prime example of accessing a proxy object field via a native method parameter, since it is a field that every native method uses. Native methods becomes simpler because they use parameters instead of cumbersome library functions for reading their Java object fields. The cost is a small one of making each native method private and requiring a public Java wrapper method. The public Java wrapper method matches the signature of its corresponding C++ method, and it prepares parameters for its native method invocation.

In cases where a native method must access a different Java proxy's C++ object address, the native method can call JNIbase.getCppObj for the Java object. Using JNIbase as a universal proxy base class simplifies Java-to-C++ object reference correlation.

Proxy method redefinitions can document C++ method redefinitions

We have established the fact that we need to define a proxy method only in the most basic class where its signature appears. Nevertheless, there are times when we wish to document redefinition of a C++ method with Javadoc comments [13] in its proxy. A convenient way to add documentation comments is to introduce a redefined proxy method that consists of an invocation of its parent's proxy method. For example we could define JsocketIPC.send as:

```
public void send(String str) { super.send(str); }
```

Javadoc comments would precede this source code definition. In many cases this

documentation redefinition is not necessary, because the C++ code is adequately documented, and because redundant documentation requires synchronization. But in cases where proxy source documentation of method redefinition is desirable, this technique provides a simple way to accomplish it.

C++ non-virtual functions and explicit method overrides

As discussed earlier, C++ provides means for client code to select a specific target object function for a call statically, regardless of the dynamic type of the target object. A non-virtual function allows a caller to specify the function by specifying the static type of an object pointer or reference used to invoke the function. The scope resolution operator "::" allows a caller to specify an exact method directly, overriding the *virtual* keyword. Both techniques allow clients to bypass an object's most derived method in favor of a method defined earlier in an inheritance hierarchy. Java does not give client code this capability. If a client's use of a target C++ class depends on this capability, then a Java proxy hierarchy must take special steps to provide equivalent capability to Java clients.

Since a Java client cannot override dynamic method selection, Java proxy classes must do so within native code. Figure 2 again provides an example. Suppose a Java client wishes to invoke socketIPC::send for a secureSocketIPC object — in this example bypassing a security check. Or suppose, similarly, that socketIPC::send were a non-virtual function. A Java proxy class can provide similar capability to Java clients by defining dedicated methods socketIPC_Send and nativesocketIPC_Send:

```
public final void socketIPC_Send(String str)
        {nativesocketIPC_Send(str,getCppObj().getPtr());}
```

where nativesocketIPC_Send is a private native method that invokes C++ send like this:

socketIPC *cppobj = (socketIPC *) cppptr ; cppobj->socketIPC::send(ascii);

instead of like this:

socketIPC *cppobj = (socketIPC *) cppptr ; cppobj->send(ascii);

The salient points of this idiom are the following.

- The public proxy method is a *final* method that takes its name and static definition from its defining class. Derived proxy classes cannot redefine the method.
- The private native method uses the scope resolution operator to select the target method statically, e.g., socketIPC::send(ascii).

The problem with this approach comes from the fact that any C++ client can use the scope resolution operator to statically resolve any virtual function. The fact that our example represents a potential security breach for secureSocketIPC shows that Java's restriction on static method typing may often be desirable. But if we wish to provide the full C++ static method resolution capability to Java clients, then it is necessary to provide final proxy methods like socketIPC_Send for every public C++ member function, in addition to the dynamic proxy methods that follow the rules of dynamic method resolution.

Our experience has been that we do not often need this capability, and it is best avoided. We have added it only in special cases where client code requires it, and its use is often an indication of a need for redesigning a feature within a class hierarchy. This problem is best treated as an exception, and it is treated effectively with this idiom.

Java finalization and C++ destruction

Significant differences exist between Java's and C++'s approaches to object recovery. Java uses automatic garbage collection to reclaim an object after all references to that object have disappeared. C++ requires an explicit application of the *delete* operator. The Java mechanism that

ties these two approaches together for JNI is the *finalize* method [11]. The Java garbage collector invokes finalize just before recovering an object, and if an object's class redefines the default, empty definition of finalize inherited from java.lang.Object, then custom finalization takes place. Our approach ties finalize to the C++ object destructor.

Figure 2 shows our definition of finalize as a protected method within abstract proxy class JgenericIPC. It invokes private native method nativeDestruct:

nativeDestruct(getCppObj().getPtr());

that invokes the C++ destructor:

genericIPC *cppobj = (genericIPC *) cppptr ; delete cppobj ;

If the C++ class hierarchy uses *virtual destructors* — and this one does — then a single finalize method can appear in the application-proxy base class. Finalize uses the invocation properties of C++ virtual functions already discussed to avoid redefinition in derived proxy classes. If the C++ hierarchy uses non-virtual destructors, then every concrete Java proxy class must implement a finalize method that calls a native method that calls its C++ class destructor directly, e.g., "delete ((socketIPC *)cppptr)" in place of the generic native destructor. A non-virtual destructor hierarchy could not use the generic destructor given above because it is the *virtual* modifier that makes a C++ destructor generic.

MULTIPLE INHERITANCE JAVA PROXY CLASS HIERARCHIES

Multiple implementation inheritance

C++ multiple implementation inheritance presents a challenge for Java because Java does not support multiple implementation inheritance. C++ does most of the work of solving this problem, since C++ is really providing the multiple inheritance support. A Java proxy hierarchy can use interface inheritance and delegation to provide a proxy class with a set of methods that

look like they are the result of multiple inheritance. C++ provides the multiple-class virtual function tables that actually dispatch methods.

Our multiple inheritance examples start with the UML model of Figure 3. The upper center of Figure 3 shows C++ classes A and B along one axis, X and Y along another, and class BY that derives from both B and Y. A adds method Ma, B adds Mb, and so on. The C++ classes are all concrete.

The J-prefixed Java proxy classes on the left side of Figure 3 use single implementation inheritance to mirror their C++ counterparts as we have already discussed. JNIbase houses the C++ object address. Derived proxy classes add appropriate constructors, class scope factory methods, new proxy methods and their native helpers.

JBY is the proxy for C++ class BY that uses multiple implementation inheritance. JBY inherits from JB using single inheritance. To understand how JBY gains access to the C++ X-Y axis, first consider Java classes JX and JY. They, too, use the single inheritance idioms already discussed. In addition, Figure 3 introduces the use of *Java interfaces*. IJX is a Java interface that specifies the public methods for JX, and therefore for C++ class X. IJY *extends* IJX to add new public methods for JY, and therefore C++ class Y. JX *implements* IJX and JY implements IJY. The inheritance-like symbol with a dashed line in Figure 3 is the UML symbol for *realization*. Using UML terminology, JX *realizes* IJX and JY realizes IJY.

JBY realizes IJY as well, thereby asserting its proxy provision of both Y operations via interface inheritance, as well as B operations via implementation inheritance. JBY implements its IJY interface via delegation to JY. Figure 3 shows JBY's private field *other* of class JY and an association link to JY. JBY gets the implementation of its other inheritance axis by constructing an object of class JY using JY's JY(cppptr) *inheritance constructor.* JBY's *proxy constructor*'s

statement

super(new cppptr(FactoryJBY(...)));

constructs a C++ BY object and stores its address in the JNIbase portion of JBY using single inheritance idioms. JBY's proxy constructor's subsequent statement

other = new JY(getCppObj());

retrieves the C++ object address from JNIbase in the upper left and gives it to JY's constructor. This is not the JY proxy constructor. It is the JY inheritance constructor used to pass the cppptr up to the JNIbase of JY in the upper right of Figure 3. After construction of the JY object, both its JNIbase part as well as JBY's base part hold an identical C++ object address. All method calls go to a single C++ object of class BY.

Getting multiple inheritance to work requires some enhancements to the mechanisms of previous sections. First, in order to allow JBY to call JY's inheritance constructor, this latter constructor must have a *public* protection level. In previous examples it was *protected* because it was used only by subclasses, but JBY is not a subclass of JY. This relaxing of protection is not a problem because the definition of helper class cppptr is *protected* within JNIbase, so only classes derived from JNIbase can manipulate C++ object addresses. This is just the level of protection we need to mimic multiple inheritance.

JBY must implement a method *My* for every method that it delegates to JY by calling "other.*My*(...)". This delegation is simple to write, and it could be automated.

Finalization is a potential danger. JBY can allow a *finalize* method on either axis to delete the C++ object, but it cannot allow both axes to do so. The inheritance constructor must now have an additional boolean operand *deleteOnFinalize* that determines whether a finalize method should delete the C++ object. JBY's proxy constructor passes *true* to the JB inheritance constructor (i.e.,

the implementation inheritance axis) and *false* to the JY inheritance constructor. JNIbase stores *deleteOnFinalize* and provides method *isDeleteOnFinalize()* so that a finalize method in a proxy hierarchy can determine whether to delete its C++ object.

There is no danger that JBY's JY object will continue to use the C++ object after the JBY object is finalized, because Java clients cannot access the JY portion of JBY independently of a JBY object. When a JBY object becomes available for garbage collection, its JY object does so as well. Figure 3 shows public permission and the boolean parameter for the inheritance constructors of JY and JX.

The asymmetry of Figure 3 causes some problems. A Java client may refer to a JBY object via a JB reference, a JA reference, an IJY reference or an IJX reference, but it may not use a JY reference or a JX reference. Furthermore, assume that two single-inheritance paths similar to the JA-JB path of Figure 3 must be merged using this multiple inheritance idiom. Clearly one of the paths must supply appropriate Java interface definitions, but suppose neither of them does. It would then be necessary to create the interface definitions and to change the Java proxy classes to reflect their implementation of these interfaces.

Symmetric multiple inheritance and second order proxies

The previous section deals adequately with *implementing* proxy support for multiple inheritance, but it is inadequate for *using* proxy multiple inheritance in Java clients. This section deals with using proxy multiple inheritance.

Figure 4 diagrams the symmetric solution to the multiple inheritance problem. Since JNI clients deal exclusively with the public methods of their C++ objects, a Java interface can represent the C++ capabilities of interest to Java clients. Only construction requires client encoding of an exact proxy class, a requirement that from the client perspective presents no

asymmetry. The Java interfaces serve as second-order proxies. Each interface reference is a client-accessible proxy for its Java class, and each Java class is a proxy for its C++ class.

The single remaining asymmetry of Figure 4 comes from the fact that JBY *extends* JB but it *uses* JY. The asymmetry is invisible to clients that use Java interface references, but it does have a ramification for implementation of a multiple inheritance proxy. The previous section asserted the idea that a proxy should use implementation inheritance from its main base class axis and interface inheritance for any others. In cases where there is no discernible main axis, the proxy should use implementation inheritance from the axis that defines the most public methods. This practice minimizes the number of delegation methods that have to be written for the multiple inheritance proxy.

This approach usually deals adequately with multiply inherited methods of the same signature, as well as with C++ diamond inheritance, i.e., multiple inheritance that leads to a common base class along two or more axes. As long a given C++ object has only one binding for a virtual function defined for that object, it does not matter which Java proxy object calls the C++ function. It is possible to define a C++ class that inherits multiple methods of the same signature via multiple inheritance, without redefinition of the method in this class to disambiguate invocation. C++ client code must overcome this ambiguity by using the scope resolution operator "::" to statically disambiguate the invocation [10]. For C++ classes where this ambiguity exists, a Java proxy must provide class-specific, final methods that use the scope resolution operator within native code as previously discussed to allow Java clients to disambiguate invocation.

In our own C++ system we have been extremely sparing in the use of multiple inheritance. The only multiple inheritance we use is single implementation inheritance combined with single interface inheritance. We have some C++ network-proxy classes that inherit implementation from

a communications class that uses socketIPC, and that inherit method interfaces for their remote, concrete counterparts. Our restricted use of multiple inheritance has allowed us to stay with the asymmetric solution presented in the last section, and to avoid the more complex symmetric solution.

CONCLUSIONS

This paper goes beyond the published techniques of connecting single Java proxy classes to C++ counterparts, exploring the issues of entire Java proxy class hierarchies. We have examined design concepts and programming idioms that enable us to surround C++ inheritance hierarchies with Java proxy hierarchies using the Java Native Interface. We have given class diagram examples for the idioms. The idioms are:

Use a proxy base class for C++ target object access. Our examples use base class *JNIbase*. This class encapsulates C++ pointer manipulation and helps manage object destruction while maintaining application neutrality. Object construction takes the form of a *proxy constructor* that invokes a private, class-static *factory method* to construct the C++ object before proxy object construction, and then passes the encapsulated C++ object pointer via a protected *inheritance constructor* to base class JNIbase.

Define one Java proxy method for C++ virtual functions that may include redefinitions. This method appears in the Java counterpart of the most basic, least derived C++ class that introduces its virtual function. The proxy method may even appear where its C++ counterpart is a pure virtual function. It relies on C++ virtual function invocation to select the correct function from the C++ hierarchy.

Use public proxy methods that invoke private native methods with parameters to avoid complicated JNI library calls. Native methods can avoid complicated and expensive calls to JNI library functions if each public proxy method passes the C++ object pointer and other proxy object fields to the native method as parameters. Design the proxy method signature to match its C++ counterpart's signature, and design the supporting native method signature to

minimize the amount of work performed by native code.

Proxy method redefinitions can document C++ method redefinitions. In cases where Javadoc comments or other source documentation is useful to document a method redefinition, define a proxy method as a one-line method that invokes its *super* method, and put the documentation with this simple redefinition.

Use a final proxy method to compensate for cases of static function resolution in C++. Give this proxy method a name dedicated to its class, and have it call a native method that uses the C++ scope resolution operator to compensate for C++ non-virtual functions and explicit method overrides. This proxy method gives Java clients a means to specify C++ target functions statically. Use this mechanism sparingly and only as needed.

Define a proxy finalize method that invokes a native destructor on its C++ object. If the C++ destructor is *virtual*, then finalize can appear in the most basic proxy class. Otherwise there must be a destructor for each concrete proxy class, and each destructor must use statically determined function resolution.

Support multiple implementation inheritance using single inheritance on one axis and a combination of interface inheritance and delegation on the other axes. Each proxy axis is a single inheritance proxy hierarchy with its on JNIbase, but each JNIbase refers to the same C++ object. The multiply-derived proxy class constructs its delegation proxy hierarchies by invoking a public inheritance constructor with the C++ object pointer. It invokes delegated target functions by invoking proxy methods on those axes. Finalization requires additional bookkeeping to ensure that only one inheritance axis deletes the target C++ object.

Make client access to multiple inheritance symmetric by using Java interfaces as second order proxies to access proxy classes. Proxy classes *implement* proxy interfaces, so proxy interface definition must occur first. Java interfaces declare public method signatures and constants, so they offer an ideal proxy for client access of C++ public methods and constants. By defining Java interfaces for all proxy classes, we ensure that the necessary machinery for future creation of multiple inheritance proxy classes is in place.

This set of idioms scales from the simple C++ class hierarchies of our examples to complex, real-life application hierarchies in its handling of C++ inheritance mechanisms under

the Java Native Interface. C++ single and multiple implementation inheritance, interface inheritance, and static method resolution all work with Java using these techniques. This appears to be a comprehensive set of idioms for mirroring C++ hierarchies in Java.

The techniques of this paper apply not only to wrapping legacy C++ hierarchies for use within Java systems, but also to creating new C++ hierarchies for inclusion in Java systems. C++ continues to be appropriate for code with stringent performance requirements that cannot be met by Java. The idioms suggest C++ design guidelines that are useful for creating C++ code that is intended from the start to live in a Java world. Avoidance of static method resolution, and a preference for Java style multiple inheritance (i.e., multiple interface inheritance) over multiple implementation inheritance, are two example guidelines to consider in writing new C++ code for Java. C++ hierarchies that avoid C++ peculiarities can get by with less than universal proxy idiom coverage.

We have used Java proxy hierarchies based on these idioms successfully in a commercial software project. We believe that they scale to meet the needs of most C++ class hierarchies. The next logical step would be automation of these idioms in a Java proxy generator. We hope to use C++ analysis modules from our debugger tool set [5] as a starting point for creating such a generator.

REFERENCES

1. *Java™ Native Interface*, http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html

2. JavaSoft's *100% Pure Java™* certification web pages, http://www.javasoft.com/100percent/

3. *ARM Java™ Technology*, http://www.arm.com/SoftSys/java/index.html

4. *The StarCore SC140 Third Party Tools Roadmap*, http://www.starcore-dsp.com/road/partiesframe.html

5. D. Parson, P. Beatty, J. Glossner and B. Schlieder, “A Framework for Simulating Heterogeneous Virtual Processors.” Los Alamitos, CA: IEEE Computer Society, *Proceedings of the 32nd Annual Simulation Symposium*, IEEE Computer Society / Society for Computer Simulation International, San Diego, CA, April, 1999, p. 58-67.

6. C. S. Horstmann and G. Cornell, *Core Java 1.1, Volume II — Advanced Features*. Palo Alto, CA: Sun Microsystems Press / Prentice Hall, 1998.

7. R. Gordon, *Essential JNI*. Upper Saddle River, NJ: Prentice Hall, 1998.

8. S. Liang, *The Java™ Native Interface, Programmer’s Guide and Specification*. Reading, MA: Addison-Wesley, 1999.

9. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

10. B. Stroustrup, *The C++ Programming Language, Third Edition*. Reading, MA: Addison-Wesley, 1997.

11. K. Arnold and J. Gosling, *The Java™ Programming Language, Second Edition*. Reading, MA: Addison-Wesley, 1998.

12. J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.

13. *Javadoc 1.2*, http://java.sun.com/products/jdk/1.2/docs/tooldocs/javadoc/

```
/* JgenericIPC.java -- JNI proxy class for C++ class genericIPC */
public class JgenericIPC {
   private native void nativeSend(String str, long ptr);
   /* Other information not shown. */
}

/* JgenericIPC.cxx -- JNI portion of proxy class JgenericIPC.java */
#include <jni.h>/* JNI library */
#include “JgenericIPC.h”/* javah-generated declarations */
#include “genericIPC.h”/* C++ class declaration */
extern “C” {
/*
 * Class:     JgenericIPC
 * Method:    nativeSend
 * Signature: (Ljava/lang/String;J)V
 */
JNIEXPORT void JNICALL Java_JgenericIPC_nativeSend
  (JNIEnv *env, jobject thisobj, jstring str, jlong cppptr) {
  jboolean iscopy ;
  const char *ascii = (const char *)
    env->GetStringUTFChars(str,&iscopy);
  genericIPC *cppobj = (genericIPC *) cppptr ;
  cppobj->send(ascii);
  env->ReleaseStringUTFChars(str,ascii);
}
}
```

Listing 1: A Java native method declaration and its C++ counterpart

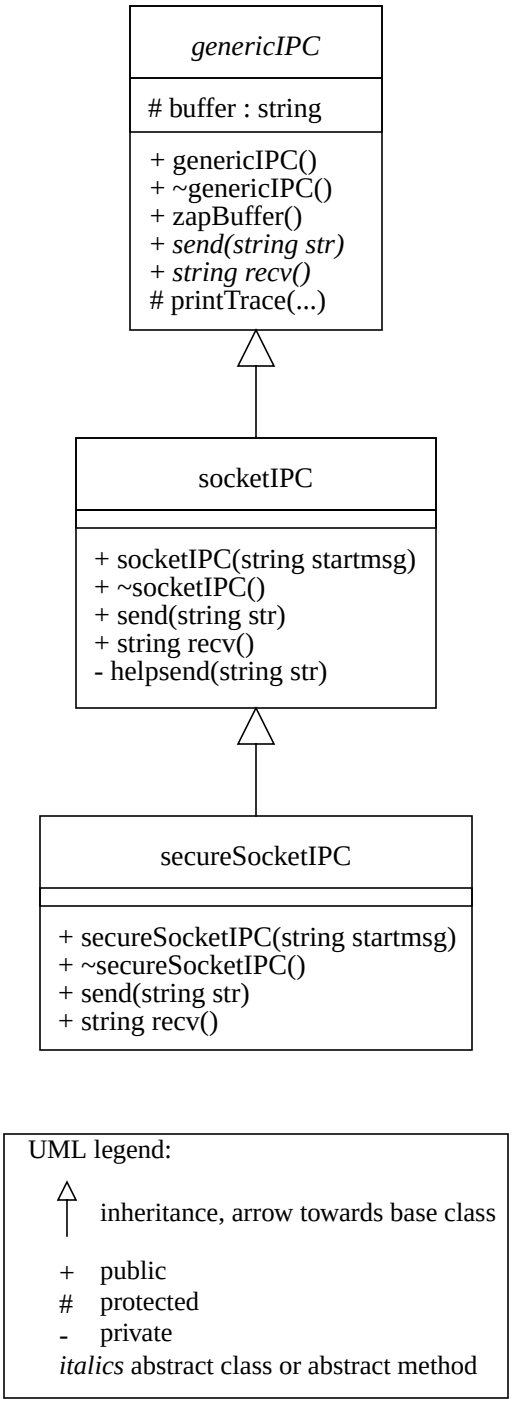

Figure 1: An example C++ class hierarchy that manages distributed communications

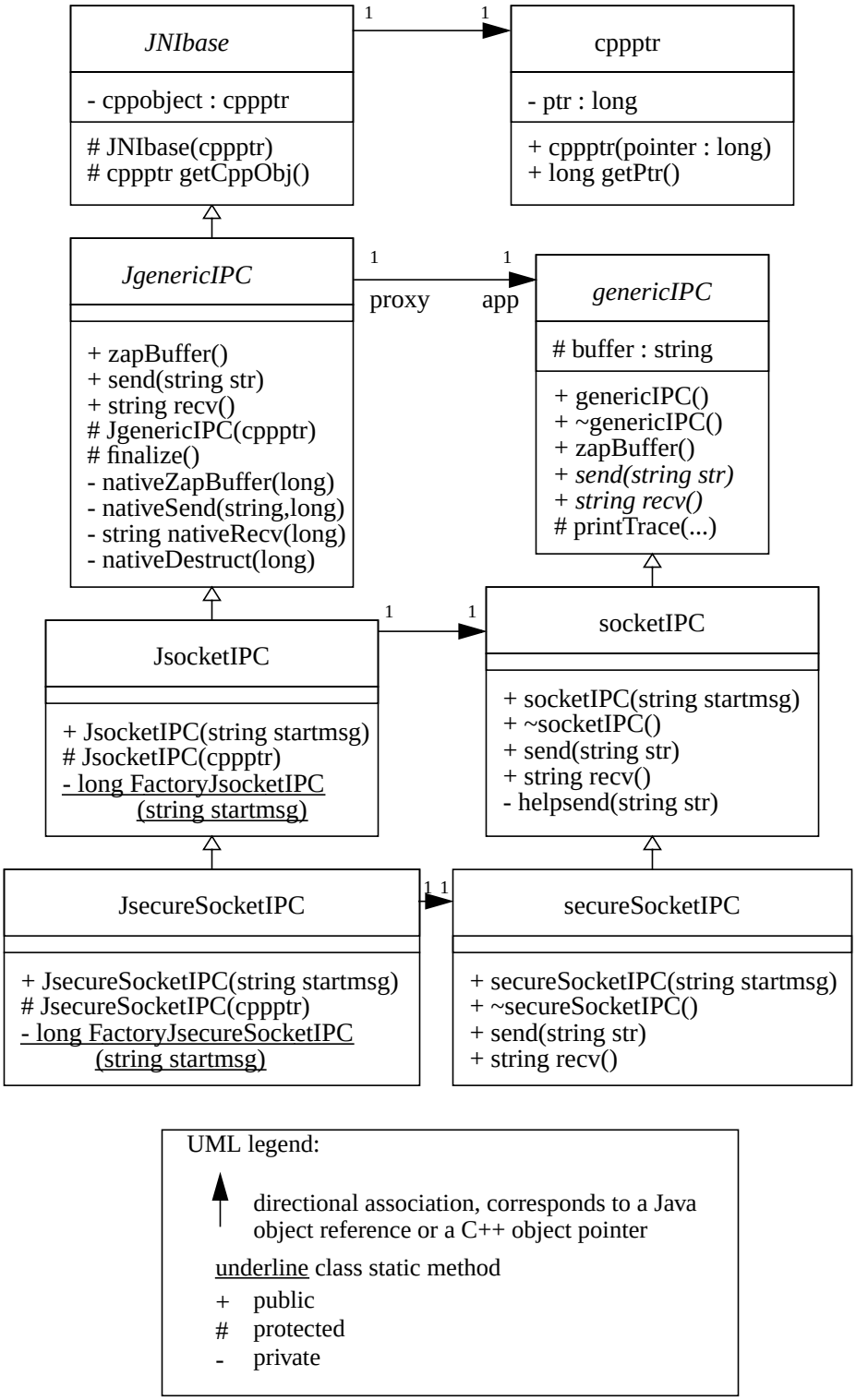

Figure 2: Java proxy inheritance mirrors C++ implementation inheritance

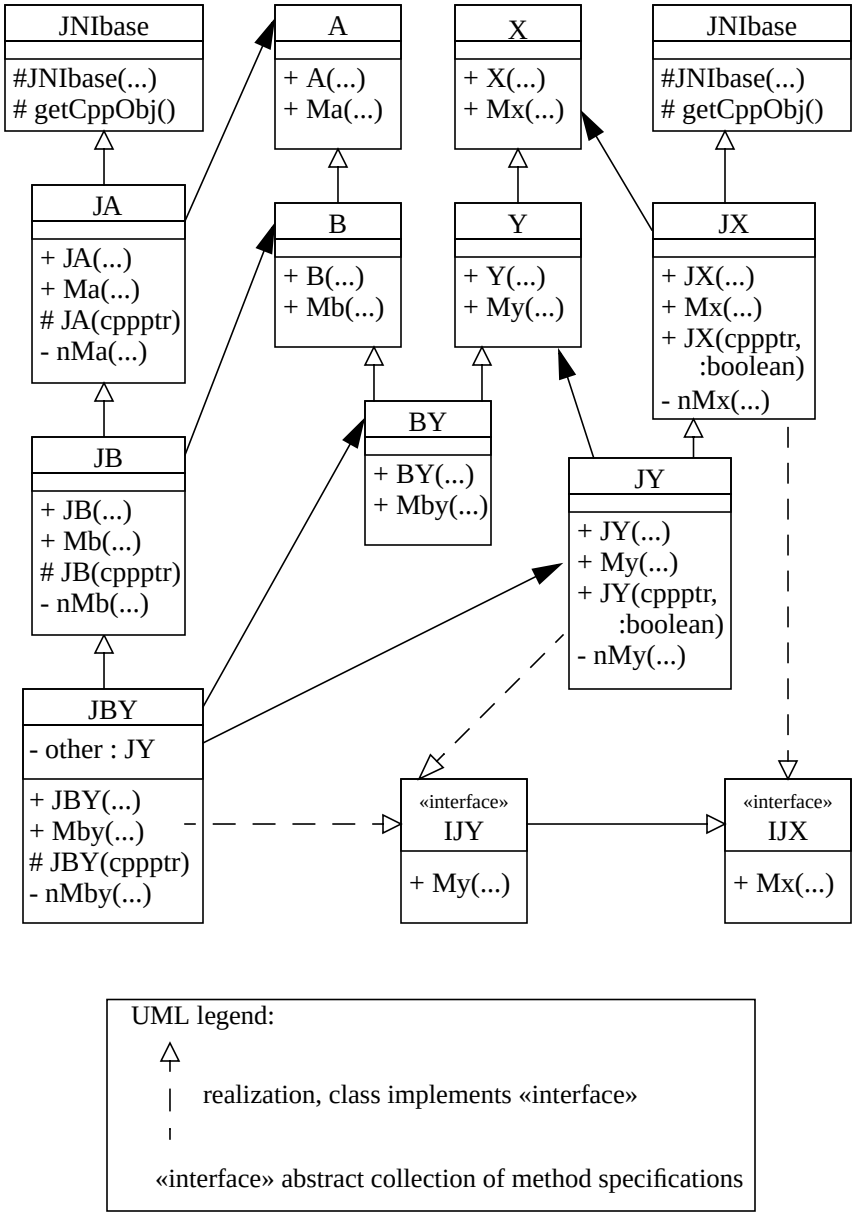

Figure 3: Java implementation proxies for C++ multiple inheritance

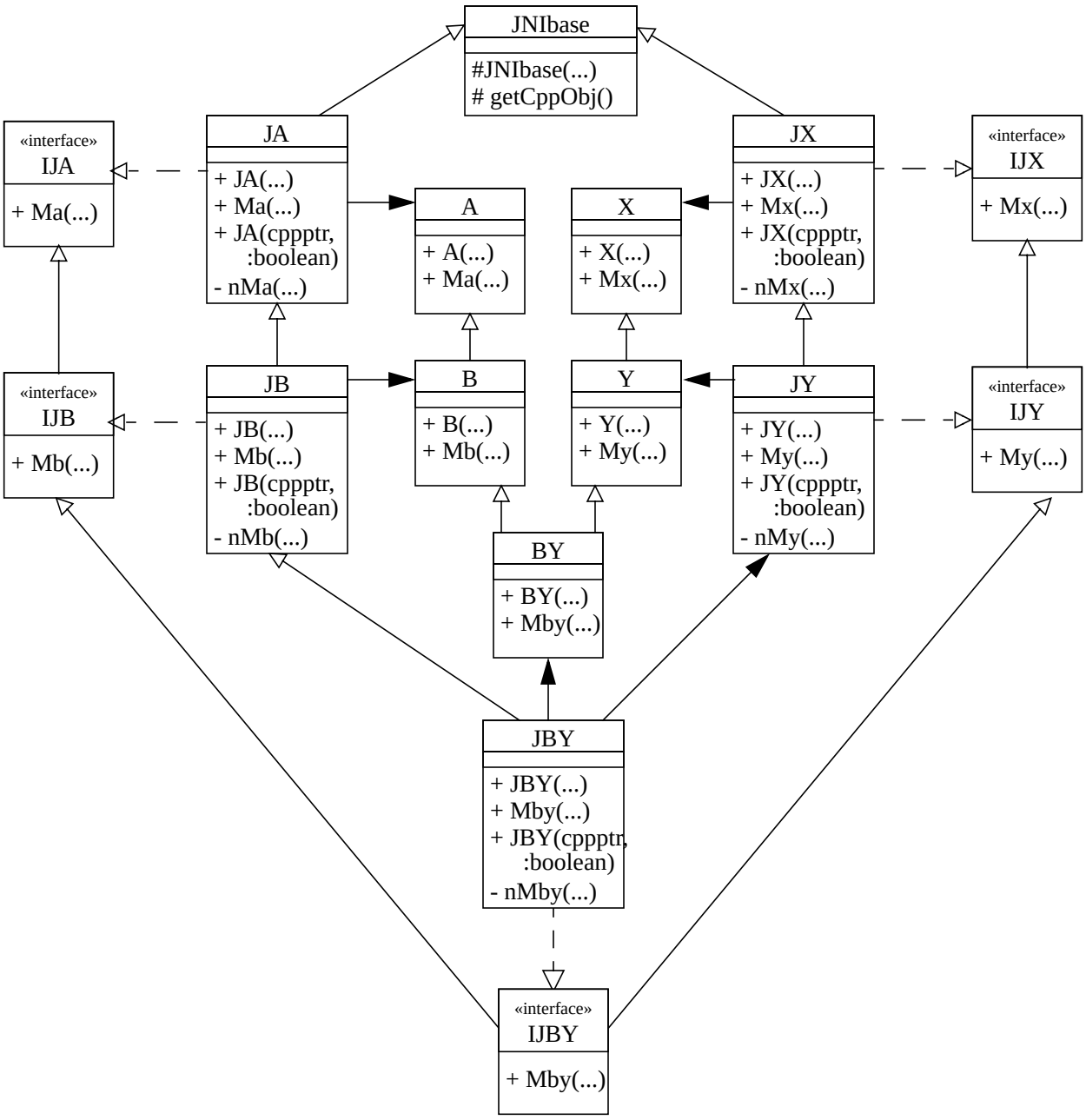

Figure 4: Symmetric Java proxies for C++ multiple inheritance