JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, no. 4, September-October 2002

Easing the Transition from C++ to Java (Part 2)

Timothy R. Culp, Harris Corporation ORIGIN Laboratory and Rollins College

In Part I of this article [1], we discussed how to address the problem of making Java and C++ co-exist in the same baseline. We introduced the Java Native Interface Simplified (JNIS) Framework as a simple mechanism that relies on common design idioms such as the Command and Observer patterns [2]. Instead of depending directly upon the JNI interface, we hide the details of JNI behind a set of C++ and Java wrappers for Commands, Events and Listeners shown in Figure 3.

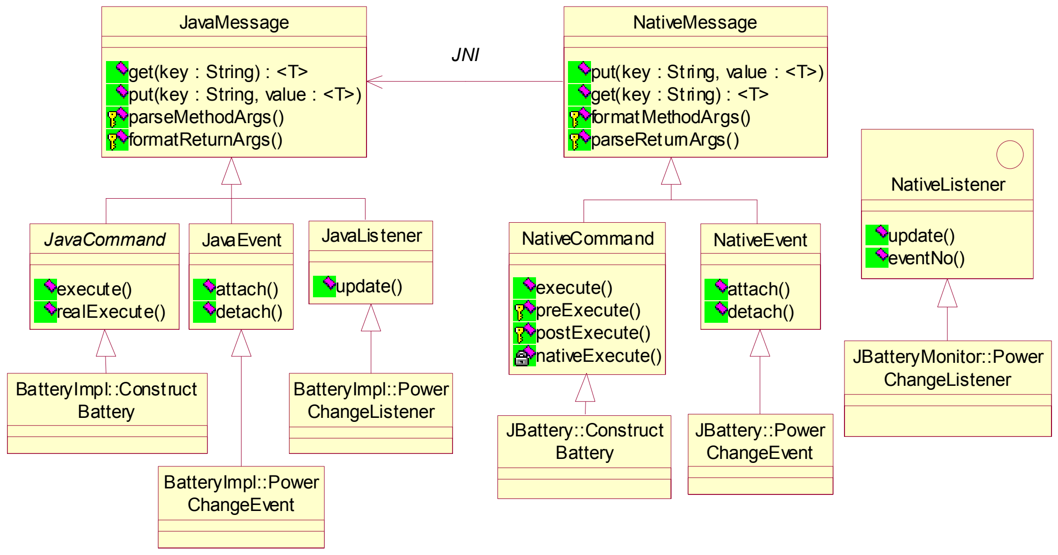

Figure 3: Class Diagram for JNIS Framework

In Part II of this article, we will show specifically how C++ libraries service Java native commands. We will show how to use pluggable factories to map Java native calls to C++ objects. We will also show how to attach Java listeners to C++ events for notification of state changes in the C++ library.

1 CREATING JAVACOMMANDS

Part I takes us through the Java side of the JNIS Command hierarchy by discussing the class relationships from top to bottom. We talked about the `NativeMessage` base class that provides "get" and "put" operations for mapping keyword-value pairs into messages for the *method* and *return* arguments. We also talked about the abstract `NativeCommand` class that provides a native "execute" method. Finally we gave a concrete example of a `NativeCommand` called `ConstructBattery` with an argument for the starting power level. We will continue walking through the class relationships on the C++ side and work our way from the bottom to the top.

If you have a Java `NativeCommand` to construct a battery then there has to be a corresponding C++ `JavaCommand` that knows how to execute the command. Notice JNIS does not get away from writing proxy pairs. There is still a remote proxy on the Java application side and a native proxy on the legacy C++ library side. The advantage is, we are declaring these proxies as subtypes of `NativeCommand` and `JavaCommand`. In this way our proxy pairs are not directly dependent on the JNI interface. Therefore programmers responsible for setting up these pairs are shielded from the details of JNI. By encapsulating the details of the transport layer, our baseline dependency on JNI remains constant regardless of the number of proxy pairs added to our system.

2 USING NAMESPACES

There are numerous commands, events and listeners we will use to service the remote Java requests on our C++ battery. The easiest way to group all these related classes is within a C++ namespace. We can create a namespace that wraps all of the commands, events and listeners associated with the implementation of a C++ battery. The combination of the namespace and class names guarantees a unique signature for identifying a particular command or event going through JNI.

```
namespace BatteryImpl {

class ConstructBattery : public JavaCommand {
public:
   virtual void realExecute() {…}
};

class DrawPower : public JavaCommand {
   virtual void realExecute() {…}
};
…
} // namespace BatteryImpl
```

3 THE “REAL” EXECUTION

The C++ ConstructBattery command class is symmetrical to the Java ConstructBattery class. Just like the Java ConstructBattery command inherits all the *put* and *get* methods from the NativeMessage base class, the C++ ConstructBattery command inherits *get* and *put* methods from the JavaMessage base class. On the Java side, we put in a value for keyword “power” and are expecting to get a return value for keyword “this”. Therefore, we will have a corresponding get for keyword “power” and a put for keyword “this” on the C++ side:

```
namespace BatteryImpl {

static Battery* battery = 0;

void ConstructBattery::realExecute() {
   if (!battery) {
      int power = getInt(“power”);
      battery = new Battery(power);
      put(“this”, battery);
   }
}
…
} // namespace BatteryImpl
```

Why does Java want a “this” reference to a C++ battery object when Java cannot really do anything with a C++ pointer? It turns out to be convenient for Java to hold onto the reference of native objects it creates even if it is just an opaque handle. One reason is the pointer could be used to indicate an error in lieu of exceptions (ie. null return indicates the construction failed). Another reason is Java may want to pass the reference along to other native commands. This turns out to be such a common idiom that we added getReference() and putReference() methods to both the NativeMessage and JavaMessage base classes where the keyword is implicitly known to be “this”.

4 THE JNI EXECUTE STUB

So far we’ve explained how Java puts its method arguments into a NativeCommand. We have also explained how C++ gets those arguments from a JavaCommand, executes the command with the given arguments and puts any return arguments back. We have not explained how the arguments are transferred between Java and C++.

If you remember from the first article, the Java compiler generates a C prototype when using the jni option for any method declared with the native keyword. The native method in our NativeCommand class is called execute:

```
public class NativeCommand {
   …
   private native String execute(String args);
   …
}
```

The java compiler produces the following stub:

```
JNIEXPORT jstring JNICALL
Java_com_org_jni_NativeCommand_execute
   (JNIEnv*, jobject, jstring);
```

This is a prototype for a native C function that services the Java NativeCommand method called execute. It takes a jstring as a method argument and returns a jstring as a return argument. The jobject argument is a reference to the Java object for which this method was invoked. The JNIEnv argument is the handle we must use for performing any work on this Java object.

5 PARSING INCOMING ARGS

What works needs to be done? Recall that when the native execute method gets invoked from Java, the format of the command string is going to be the name of the C++ native proxy class followed by a list of keyword-value pairs for the incoming arguments. The ConstructBattery command is formatted into the following string:

```
BatteryImpl::ConstructBattery { power 100 }
```

This formatted command string will come into our native C function as the jstring argument. The jstring can be converted from a 16 bit UTF to an 8 bit C string by using the GetStringUTFChars() method on the JNIEnv class [3]. Since we know we will be parsing the string, it's best to store it immediately into a C++ istringstream.

```
JNIEXPORT jstring JNICALL
Java_com_org_jni_NativeCommand_execute
   (JNIEnv *env, jobject obj, jstring args) {
   jboolean owner;
   istringstream myArgs(
      env->GetStringUTFChars(args, &owner));
...…
```

6 USING FACTORIES

Now that we have the command wrapped in an `istringstream`, how do we know which C++ object to invoke? We need the commands to be open ended so we can support a wide variety of commands. A hard-coded switch statement of commands is not going to be acceptable. This is a perfect application for a pluggable factory [4]. A factory defers the creation of the proper subtype of an object to another class. In this case, we will use an STL map that associates a command name to a particular instance of an object that knows how to execute that command [5]. All command objects will register themselves into this map during static initialization. This is typically accomplished by having each command class own a static instance of itself, which enables it to register while constructing during static initialization. We can find the right command object by looking up the unique command class name in the registry.

```
string cmdName;
JavaCommand* cmd = 0;

myArgs >> cmdName;
cmd = dynamic_cast<JavaCommand*>(
   theRegistry[cmdName]);
```

The reason for the dynamic cast is the registry actually contains a set of `JavaMessage` pointers. This allows the registry to store `JavaEvents` and `JavaListeners` as well as `JavaCommands`.

7 EXECUTING THE COMMAND

Once we have a handle to the proper `JavaCommand`, we pass the remainder of the stream to the execute method:

```
    string returnArgs;
    if (cmd)  returnArgs = cmd->execute(myArgs);
```

The default behavior of the execute method is to first parse the incoming arguments, invoke the polymorphic `realExecute()` method, then return the formatted arguments back to Java:

```
string JavaCommand::execute(istream& is) {
   parseMethodArgs(is);
   realExecute();
   return formatReturnArgs();
}
```

The two methods, `parseMethodArgs()` and `formatReturnArgs()`, are helper functions in the `JavaMessage` base class that handle the details of parsing/formatting the keyword value pairs. The `realExecute()` method is defined by the derived class that knows how to execute the command.

8 RETURNING TO JAVA

Once the command is executed, there is some small cleanup that needs to be performed. We will use the `JNIEnv` object to release the `jstring` so we do not prevent the JVM garbage collector from recovering the memory [6]. We also use the `JNIEnv` object to return the formatted return arguments back to Java:

```
if (owner==JNI_TRUE) {
   env->ReleaseStringUTFChars(
      args, myArgs.str().c_str());
}

   return env->NewStringUTF(returnArgs.c_str());
}
```

We have now covered the execution of a native command from the point it is first created in Java through the C++ library and back again. The interaction diagram in Figure 4 shows the sequence of operations from the Java programmer's perspective while Figure 5 shows the sequence of native operations in C++:

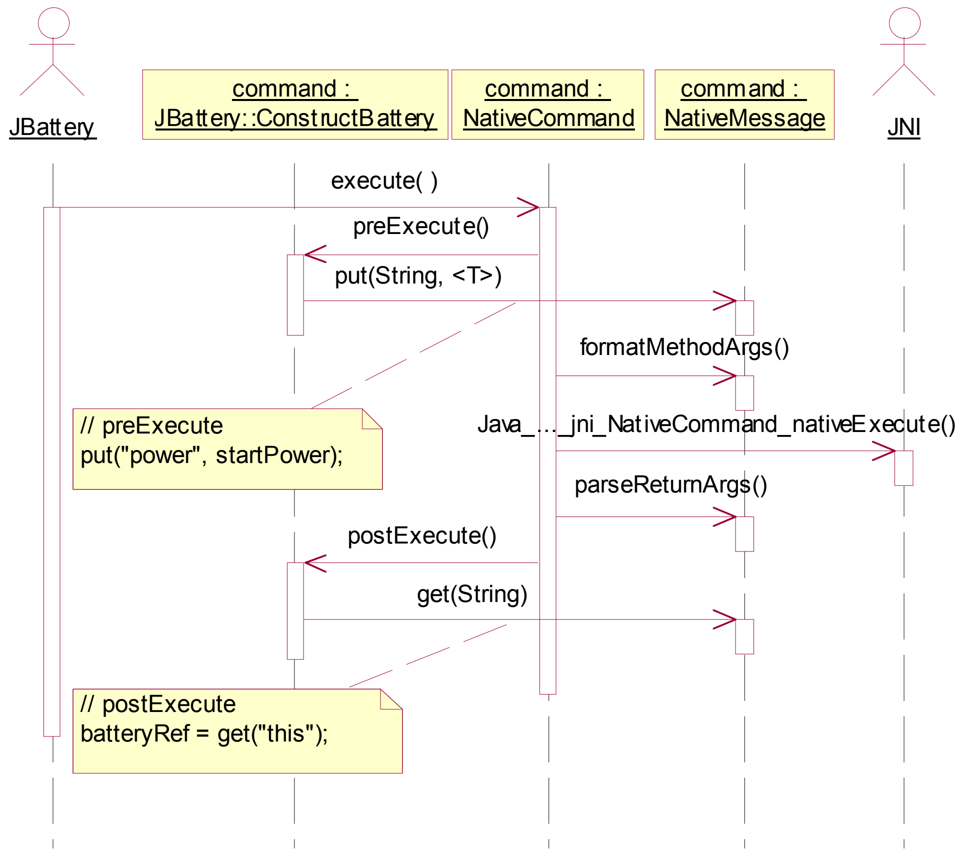

Figure 4: Sub-Object Sequence Diagram for ConstructBattery Command

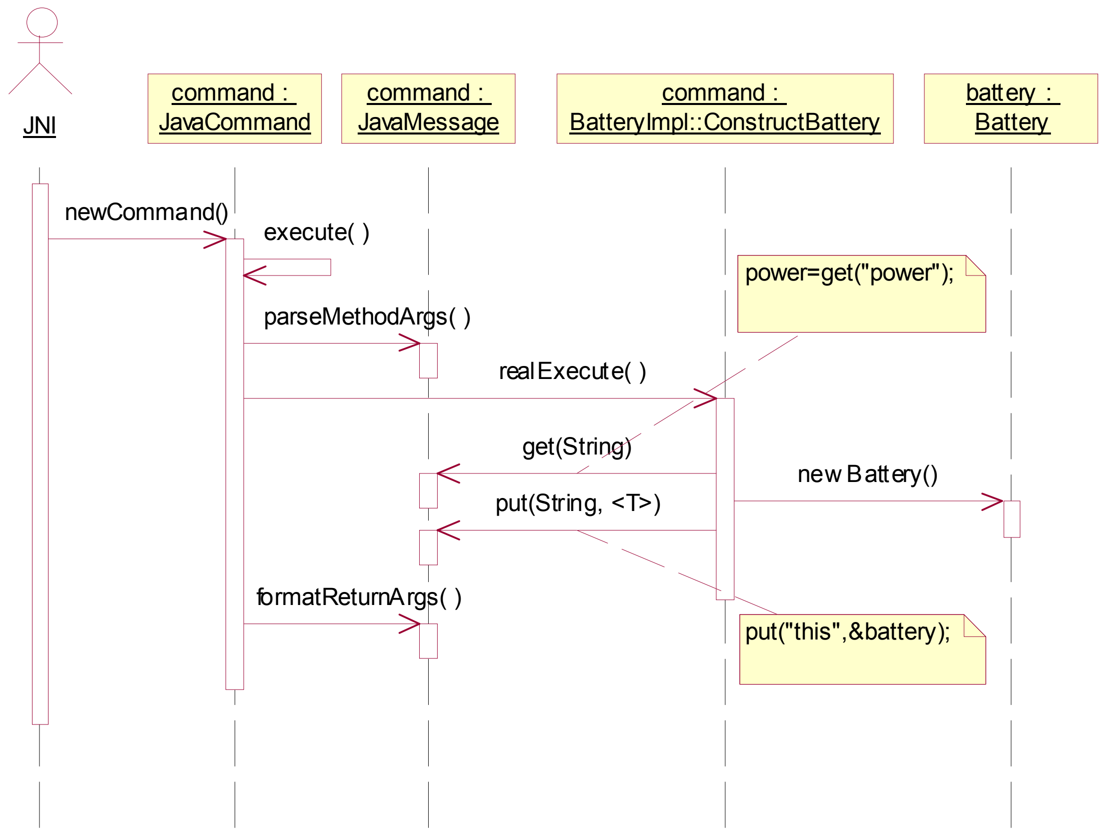

Figure 5: Sequence Diagram for Native ConstructBattery Command

9 JAVAEVENTS AND JAVALISTENERS

Now that we understand the basic mechanics for executing commands, attaching and listening to events are not much different. To create an event on the Java side, we derive a specific event from `NativeEvent`. The derived event class knows the name of the specific native proxy that will implement this event. Likewise, we derive a specific event from `JavaEvent` on the C++ side that is responsible for its implementation. In addition to the event classes, we also define a `NativeListener` and a `JavaListener` that knows what actions need to be taken when the event takes place.

As an example let's follow the flow of control when propagating a "power change" event through the JNIS framework. The `JBattery` class will define an event called `PowerChangeEvent` and the `JBatteryMonitor` class will define a listener called `PowerChangeListener`.

```
public class JBattery {
   class PowerChangeEvent extends NativeEvent {
      public String nativeProxy() {
         return “BatteryImpl::PowerChangeEvent”;
      }
   }
   …
}

public class JBatteryMonitor {
   class PowerChangeListener implements NativeListener
   {
      void update(NativeEvent e) { … }
      int eventNo() { return JBattery::PowerChange; }
      void String nativeProxy()
         { return “BatteryImpl::PowerChangeListener; }
   }
   …
}
```

During the construction of the `JBatteryMonitor`, our controller is going to create a battery, create a listener for a specific battery event and then attach the listener to the battery.

```
public JBatteryMonitor(int startPower) {
   theBattery = new JBattery(startPower);
   theBattery.addNativeListener(
      new PowerChangeListener());
   …
```

10 DEFINING PROXY EVENTS

The `JBattery` remote proxy class contains a list of enumerations that describe the type of events it supports. It also contains a private list of all these events. These events are populated during the construction of a `JBattery`. It provides a method for attaching a given listener by calling the `addNativeListener()` method on the appropriate event. Note that the `PowerChangeListener` does not derive from `NativeListener` since `NativeListener` is an interface. The only restriction on any native listener is that it implements an `update()` and an `eventNo()` method:

```
public class JBattery {
   public static final int FullPower=0;
   public static final int PowerChange=1;
   …
   private NativeEvent[] theEvents;
   …
   public JBattery(int startPower) {
      theEvents = new NativeEvent[4];
      theEvents[0] = new FullPowerEvent();
      theEvents[1] = new PowerChangeEvent();
      …
   }

   public void addNativeListener(NativeListener l){
      NativeEvent event =  theEvents[l.eventNo()];
      event.addNativeListener(listener);
   }
   …
}
```

11 ATTACHING PROXY LISTENER

The actual attaching is not done by the derived `PowerChangeEvent` class but by the base class `NativeEvent`. It handles the details for mapping a native event to a native listener. In the `NativeCommand` class, we had a native execute method that forwards a message to the native API containing the command name and the list of arguments. For an event, we have a native `attach()` method that will forward the name of the event and the listener to the native API so they can be mapped together. Likewise, there will be a `detach()` method that will forward the name of the event and the listener to the native API so the association can be unmapped when the listener is no longer interested in participating:

```
public NativeEvent {
   public void addNativeListener(NativeListener l) {
      attach(nativeProxy(), l.nativeProxy());
   }

   private native String attach(
      String eventName, String listenerName);

   private native String detach(
      String eventName, String listenerName);
   …
}
```

12 MAPPING REMOTE/NATIVE PROXIES

Unlike the native execute method, the native attach function is not concerned about incoming arguments. Since it is only interested in the names of the event and listener, we can convert them directly into strings and look them up in the registry. If we find valid C++ events and listeners registered for those names, then we can attach them together. The event `attach()` method just adds the given listener to an internal list of objects to be notified when the event occurs. We also store the reference to the `jobject NativeEvent` for which this attach is being called because this Java reference will forward updates to Java listeners.

```
JNIEXPORT jstring JNICALL
Java_com_org_jni_NativeEvent_attach(
   JNIEnv* env,  jobject evtRef, jstring evt,  jstring lst)
{
   …
   string evtName(
      env->GetStringUTFChars(evt, &owner1));
   string lstName(
      env->GetStringUTFChars(lst, &owner2));

   JavaEvent* event = 0;
   event = dynamic_cast<JavaEvent*>(
      theRegistry[evtName]);

   JavaListener* listener =  0;
   listener = dynamic_cast<JavaListener*>(
      theRegistry[lstName]);

   if (event && listener) {
      event->attach(listener);
      listener->setEvent(evtRef);
   }
   …
}
```

The C function for the native detach is identical except a `detach()` method is invoked on the event object instead of an `attach()`. The native attach and detach functions can easily be combined into a single function with a flag that denotes whether you are attaching or detaching.

13 KEEPING PROXIES IN SCOPE

One subtle gotcha with the JNI interface is the lifetime associated with a `jobject` reference. Even though a java object may stay in scope throughout our java application, there is no guarantee that the `jobject` reference will remain valid beyond the call of the native method in which it was invoked. It is maintained by the JVM as a *local* reference on an environment call stack.

Our attach method associates a `jobject` event reference to a native C++ listener during one native call. The notification of updates happens during a subsequent native call. Thus the `jobject` event reference will be stale by the time we are ready to use it. The JNI interface provides a mechanism to allow you to continue to use the same `jobect` reference through multiple native calls by creating a *global* reference. We can bury the details of creating and releasing global references inside the `setEvent()` and `clearEvent()` methods which get invoked during the native attach and detach operations:

```
void JavaListener::setEvent(jobject evtRef) {
   theEvent = theJNIEnv->NewGlobalRef(evtRef);
}

void JavaListener::clearEvent() {
   if (theJNIEnv && theEvent) {
      theJNIEnv->DeleteGlobalRef(theEvent);
      theEvent = 0;
   }
}
```

14 FORWARDING NOTIFICATIONS

Now that we have a mapping between a native C++ listener and `jobject` event reference, we can use this mechanism to forward state changes to Java. A model change on the battery forces the event to notify all of its observers. For instance, a `drawPower()` call on our C++ battery invokes a `PowerChange` event. It might also invoke a `LowPower` or `NoPower` event as well, depending on the current state of the battery. One of the observers in the `PowerChange` event is the native C++ Listener that holds a reference to the `jobject` event reference. The update method on the native C++ Listener gives the listener an opportunity to format a message by using put calls for any arguments we need to forward:

```
namespace BatteryImpl {
…
void PowerChangeListener::update(Event* e){
   PowerChangeEvent* evt = 0;
   evt = dynamic_cast<PowerChangeEvent*>(e);
   if (evt) {
      put("power", evt->battery()->getPower());
   }

   forwardNotify();
}
…
} // namespace BatteryImpl
```

15 INVOKING REMOTE METHODS

Forwarding the notify to the `jobject` event reference is interesting because this is the first time we see C++ invoking a method on a Java object. We use our `JNIEnv` reference to get the class type for the `jobject` we stored at the time we attached the C++ listener to a Java event. We use the same `JNIEnv` reference to get the method ID on the `NativeEvent` class type for forwarding the notification. We must pass in the name of the method we wish to call as well as a string that describes the method signature (the signatures of Java methods are found by using "javap –s"). Finally, we make the call through the `JNIEnv` reference using the `jobject` reference, the method ID and a new UTF String with the return arguments:

```
void JavaListener::forwardNotify() {
   jclass NativeEvent =
      theJNIEnv->GetObjectClass(theEvent);

   jmethodID notify = theJNIEnv->GetMethodID(
      NativeEvent,"notify","(Ljava/lang/String;)V");

   theJNIEnv->CallObjectMethod(
      theEvent,
      notify,
      theJNIEnv>NewStringUTF(
         formatReturnArgs().c_str()));
}
```

16 UPDATING THE CONTROLLER

Once the `jobject’s` method is called, control is transferred to our specific Java `NativeEvent` object; in our case, the Java `PowerChangeEvent` object. This object will get any return arguments, such as the “power” argument that was put in by the C++ `PowerChangeEvent` object. The Observer pattern will continue by having the Java `NativeEvent` update all of its `NativeListeners` so the listener can perform whatever actions are appropriate. The `PowerChangeListener` in our `JBatteryMonitor` controller has an update method that gets the new power level and updates a progress bar:

```
public class NativeEvent {
   …
   protected void notify(String returnArgs) {
      parseReturnArgs(args);
      theListener.update(this);
   }
}

class PowerChangeListener
   implements NativeListener {
   public void update(NativeEvent event) {
      int power = event.getInt("power");
      theProgressBar.setValue(power);
      …
   }
}
```

17 CONCLUSION

Walking through the entire framework and trying to understand all the details of JNI and the method of communication through the patterns is challenging. Once you have worked through a few examples, the framework itself is very simple to use. Java commands can execute C++ libraries simply by deriving from `NativeCommand` and overriding the execute method to put arguments into a command and get return parameters back. C++ can respond to these commands simply by deriving from `JavaCommand` and overriding the `realExecute()` method to get arguments from the command and put return parameters back. C++ commands must also register themselves on a list so the number of commands supported is open ended. The mechanism for defining events and listeners is essentially the same.

We continue to work on the JNIS framework to move it out of the prototype stages and more into an industrial strength solution. We are currently adding an exception handling mechanism to propagate exceptions through the JNI transport layer. We are also adding environment stacks to handle multi-threaded calls through the API.

Once you have a framework for Java to access existing C++ services, the next logical extension is to distribute these legacy services to network applications. Lightweight Java clients using RMI can invoke remote methods on Java servers. These Java servers can either implement the services directly or choose to use JNIS if an existing C++ implementation is already available.

ACKNOWLEDGEMENTS

To Katee, for all the things in life that can't be programmed.

REFERENCES

[1] Culp, T. *Easing the Transition from C++ to Java (Part 1)*, in Journal of Object Technology, vol. 1, no. 2, July-August 2002, pages 79-93. http://www.jot.fm/issues/issue_2002_07/column7

[2] Gamma, *et. al. Design Patterns*, 1st Ed., Addison-Wesley, Reading, MA. 1995.

[3] Gorden, *essential JNI*, Prentice Hall, 1998.

[4] Culp, T. "Industrial Strength Pluggable Factories", *C++ Report*, 11(9), Oct. 1999.

[5] Gamma, *et. al. Design Patterns*, 1st Ed., Addison-Wesley, Reading, MA. 1995.

[6] Gorden, *essential JNI*, Prentice Hall, 1998.

About the author

Timothy Culp is the Chief Software Architect for the Harris Corporation ORIGIN Laboratory and an Instructor at Rollins College. He can be contacted at timothy.culp@computer.org.