JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 1, No. 2, July-August 2002

Easing the Transition from C++ to Java (Part 1)

Timothy R. Culp,

Industry is in a dilemma. Most software developers are very interested in learning Java and finally using a truly cross-platform language. While we successfully develop and maintain large-scale C++ baselines across several platforms, C++ is only cross-platform to a point. Platform and vendor independence always break down when it is time to develop the graphical user interface. A pure Java solution might be a good answer for a startup company but is not practical for traditional IT companies that have significant investments in C++. These legacy baselines represent our core technology that keeps us in business. C++ is not going away any time soon. The marriage of Java GUIs and existing C++ libraries seems inevitable.

The problem to address is how to make Java and C++ peacefully co-exist in the same baseline. There are a number of communication mechanisms already available, most notably JNI and CORBA. All of these mechanisms come with a price in terms of complicating the development/runtime environment. This overhead is often the reason hard core C++'ers are not ready to accept Java as a viable partner. A simple and reliable mechanism is needed, preferably one that relies on well documented design patterns and idioms so there is little or no learning curve.

1 SIMPLIFYING COMMUNICATION

This paper introduces a simple transport layer that is based on the Model-View-Controller (MVC) [1] architecture. Our GUI controllers will be Java classes that will use existing models in legacy C++ libraries. The Java Swing package will be our view. The underlying mechanism for our transport layer will be the Java Native Interface (JNI). The complexities of JNI are hidden from developers by providing them two simple communication protocols using well-known design patterns. The Command pattern is used so Java controllers can directly command the C++ model to take some action. The Observer pattern is used so Java controllers can register for C++ events; consequently, C++ can indirectly notify java listeners when those events occur. The best thing about this interface is C++ developers do not have to understand JNI or Java to get started. Likewise, Java programmers do not need to know about JNI or C++. All they have to know is how to derive classes from their respective language, override a few virtual functions, and they are ready to communicate.

Cite this column as follows: Timothy Culp: 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 WHY NOT CORBA?

Let us begin by saying that CORBA is clearly an elegant solution to solving the problem of multi-language, enterprise development. The ability to invoke methods on a remote object without any concern of the client architecture is an extremely powerful tool.

However, you normally pay for this power with pricey development licenses or high training costs. Your development environment becomes much more difficult to debug and code. Is your ORB running? Are you sure you are launching the right services? Are they the latest installed version or the one you just modified trying to fix a problem? Installation and runtime environments become more complicated and typically require a savvy end user.

Granted, if you are working on a large project with an experienced team and plenty of schedule then these considerations are inconsequential. If you are a small team working on a research project for a customer demonstration next week, chances are you are going to want something simpler.

3 WHY JNI?

JNI is a free package bundled under the Java Development Kit. It introduces the keyword `native` to the Java Language Specification. This allows a Java class to defer the implementation of a particular method to another language compiled in a native environment.

This is a very simple interface because it allows a C++ object the ability to stand in for a Java object when efficiency is required or the code just already happens to be written. JNI also allows C++ to acquire a reference to a Java object, so it can invoke Java methods and access data members.

There is a strong resistance in literature [2] to the use of JNI because it sacrifices portability, one of the major selling points of Java's *write once, run everywhere* mentality. This does not really apply to industry where you most likely have C++ libraries ported to all the platforms required for your business. Given the option of reusing cross-platform C++ libraries or rewriting implementations in Java for the sake of a pure Java solution, the obvious business case is to reuse legacy implementations.

4 USING JNI

This article cannot begin to describe the complete API for JNI. There are plenty of good books, such as *Essential JNI* [3], with examples and reference material to solve most

problems you will come across. A simple problem to introduce some important concepts will help us get started.

Assume we have a C++ class called Battery that will maintain the current state of a battery and allow a user to draw some power when required:

```
class Battery {
public:
   Battery(int p) { thePower=p; }

   void drawPower(int p) { thePower-=p; }
   int getPower() { return thePower; }

private:
   int  thePower;
};
```

Now we want to develop a GUI controller-view called `JBatteryMonitor` that displays the current state of our C++ battery object. We choose to develop the controller in Java so the GUI will run under all the same platforms as our existing C++ library that contains the Battery model. The implementation for the battery already exists, so we provide an internal interface to our battery with the implementation deferred with the native keyword:

```
public class JBatteryMonitor extends JFrame {
   …
   private native void drawPower(int p);
   private native int getPower();
   …
}
```

Java native classes are compiled the same way as normal java classes using the `javac` command except they generate incomplete byte code:

```
%javac JBatteryMonitor.java
```

Trying to invoke a native method that is not yet connected to a native implementation results in an `UnsatisfiedLinkError`. In order to connect to our C++ implementation, we must generate a header file that provides a function prototype with the proper signature to use as a bridge for a native call. The `javah` command is used to generate the header:

```
%javah –classpath . –jni JBatteryMonitor
```

The `-jni` option is used to generate a JDK1.1 style function prototypes as opposed to the earlier JDK1.0 style. The resultant `JBatteryMonitor.h` header file looks something like this:

```
// Java_JBatteryMonitor.h
#include <jni.h>

JNIEXPORT void JNICALL
Java_JBatteryMonitor_drawPower(
   JNIEnv *, jobject, jint);

JNIEXPORT jint JNICALL
Java_JBatteryMonitor_getPower(
   JNIEnv *, jobject);
```

The `javah` command generates a C header file that provides the linkage between the two environments. The JNI interface provides a mapping between Java and C++ data types so it is easy to convert from one to the other. A simple implementation for getting the current power might look like this:

```
// Battery.cpp
#include “Battery.h”
#include “Java_JBatteryMonitor.h”

static Battery* battery = 0; // singleton

JNIEXPORT jint JNICALL Java_JBatteryMonitor_getPower(
   JNIEnv *, jobject)
{
   if (!battery) battery = new Battery(100);
   return battery->getPower();
}
```

The last step is for the Java application to load the C++ native library at runtime. While this can be done at any time prior to a user invoking a method from a Java battery object, the most appropriate place is to load it the first time the `JBatteryMonitor` class file is brought into memory. A Java static block makes this very simple since everything in the block will get executed the first time the class is loaded:

```
public class JBatteryMonitor extends JFrame {
   …
   static { System.loadLibrary("legacy"); }

}
```

You do not need to specify any prefix or postfix labels to your library name such as `*.dll` or `lib*. so` because the `System.loadLibrary` method will provide the proper notation for your particular platform.

5 WHY HIDE JNI?

In practice, you normally do not put JNI code directly in your models or your controllers. This is a violation of the separation of policy from mechanisms principle [4]. Since this is an area of our architecture that is likely to change, we want to isolate those details from the rest of the code.

Another reason for hiding JNI is that the API allows you to bypass encapsulation. Once a C++ object has a reference to a Java object, there are no more protected or private boundaries. You want to localize your code that uses JNI to prevent errors or poor programming practices.

Finally, you want to isolate JNI because the implementations often result in complicated code. The API calls are vastly different depending on whether you are passing primitives or strings, accessing data members or invoking methods. All of this complexity leads to ugly code that is hard to understand and even harder to maintain.

6 HIDING JNI WITH PROXIES

A more typical implementation is to use a proxy. A proxy provides a surrogate or placeholder for another object to control access to it [5]. We will use a pair of proxies for hiding the JNI layer.

On the Java side, we have a remote proxy [6] called `JBattery` that acts as a stand-in for the native C++ Battery. On the C++ side, we have a native proxy called `BatteryImpl` that provides the implementation for calls made through the remote proxy. We now have four classes needed to provide the same capabilities as our previous example but with the added benefit of encapsulating the details of JNI from our Battery model and `JBatteryMonitor` controller production code.

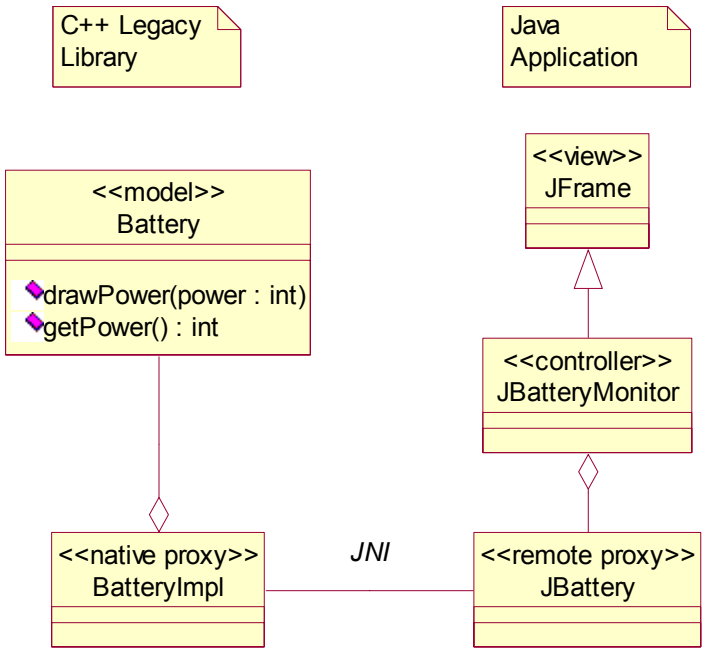

Figure 1: Using Proxies to Hide JNI

7 SCALABILITY ISSUES WITH PROXIES

So far we have discussed JNI as a simple and effective communication mechanism, especially for projects short on schedule or low on budget. We saw a brief example of how JNI is used and we demonstrated the use of proxy pairs to help isolate change from our production code.

So what is the problem? The problem is, over time baselines grow. Demonstration code turns into prototypes and prototype code turns into delivered systems. What works well for a demo does not necessarily scale well for industrial applications.

Our current design requires a proxy pair to be written whenever a Java interface defers to a C++ implementation. As the baseline grows, the number of Java objects deferring to legacy C++ implementations will grow as well. Every time another remote/native proxy pair is added to your system, your dependency on JNI increases. These dependencies will start crossing multiple software disciplines so more of your team will be required to understand the JNI API. More areas of your code will have to be changed when sweeping changes are made to JNI (that would never happen!). What started as a good design for a customer demonstration has evolved into a maintenance nightmare.

We should be strongly motivated to prevent a dependency on any API that grows linearly with our baseline…especially in the area of cross-language communication. We *know* this technology will change over time. Our goal is to leverage JNI today but plan for tomorrow so that our transport layer can be updated or replaced as the technology matures.

8 SIMPLIFYING JNI

In order to reduce the number of classes that are coupled with JNI and thus enhance our scalability, a better design would keep our dependency constant as our baseline grows and more developers take advantage of the interface. To achieve this goal, we will limit the number of remote/proxy pairs to a small subset of interfaces that can be used for a wide variety of needs. There are essentially three interfaces required to support an MVC architecture.

The first interface is the *Command* interface. This is when the Java controller wants to issue a command to the C++ model. This type of communication mechanism is acceptable in our architecture because the controller is allowed to know about the model and make explicit calls. An example of a controller command might be "construct a battery with 100% power" or "draw 1% of power from the battery".

The second interface is the *Event* interface. This interface is used when a Java controller wishes to attach to public C++ model events. For example, our C++ battery may have events such as FULL_POWER, NO_POWER, CHANGE_POWER or LOW_POWER. The `JBatteryMonitor` controller may choose to attach to one or more of these events whenever it becomes visible and then detach when the window is minimized.

The third interface is the *Listener* interface. This is used when a Java object is interested in listening for events that are generated by the C++ model.

These three mechanisms essentially provide 2-½ means of communication. Java objects can *explicitly* command C++ objects. Java objects can *explicitly* attach/detach to C++ events. C++ objects can *implicitly* notify Java listeners when events occur. It is important that the C++ objects are not aware that they are notifying Java listeners so the same C++ objects can be used to notify observers in other views as well (whether they be C++ or Java).

9 JNI COMMANDS

The Command pattern encapsulates a request as an object letting you parameterize clients with different requests [7]. To use this pattern in conjunction with our proxy-pair

approach, we will have a Java remote proxy called `NativeCommand` which will contain an execute method. This method will pass through to a native method that will defer to its native implementation to perform the real execution. It is named `NativeCommand` because this class can be used for any native implementation, not just C++. The corresponding C++ native proxy will be called `JavaCommand` and it will also contain a virtual execute method. The real implementation will be deferred to an appropriate subclass of `JavaCommand`. It is named `JavaCommand` because it is explicitly intended to implement a Java command.

As in most abstract frameworks, it is difficult to come up with method signatures in the base class that are general enough for a variety of needs. This is definitely the case in the execute method since we have no idea at this level what commands are going to be executed, what the method's arguments should be and what an appropriate return value might be.

We will use an approach similar to XML by allowing subclasses to pass user specific information as tagged textual data. This is a very simple approach but incredibly expressive. Our design will pass keyword-value pairs as the argument for the native execute method. The return values will also be a series of keyword-value pairs. Our native method then is a method that takes a string and returns a string. The string content is made up of keyword-value pairs determined by subclasses:

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

The capability to put keyword-value pairs as both method and return arguments is going to be common across the Command, Event, and Listener interfaces. Therefore, instead of trying to capture the string manipulation logic in `NativeCommand`, we will have a base class called `NativeMessage`.

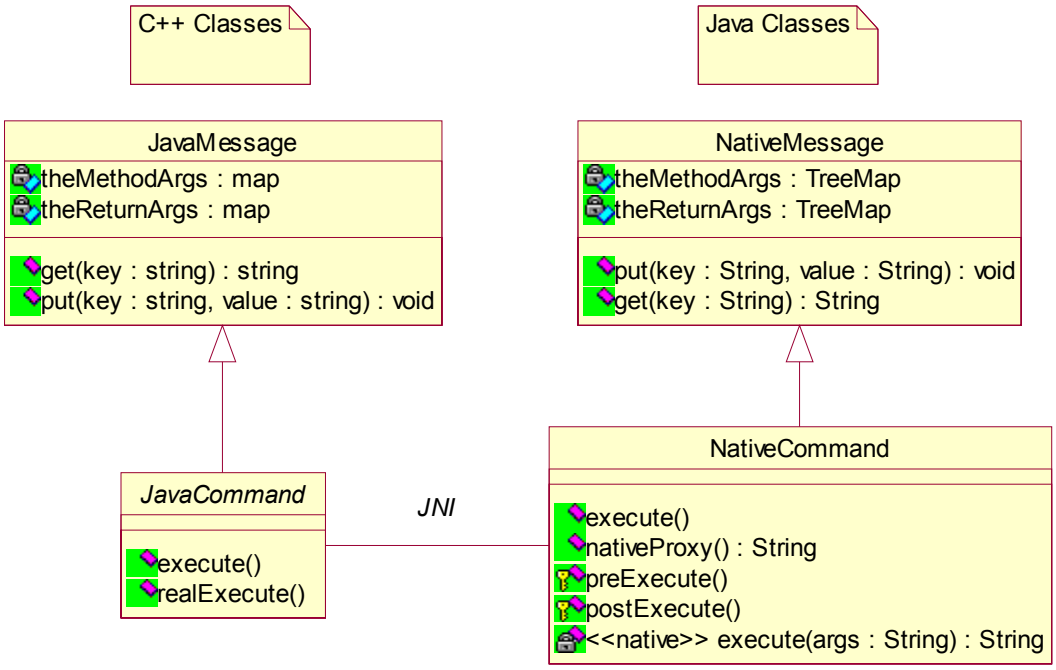

Figure 2: JNI Command Interface

10 GENERALIZATION ISSUES

Forcing every command through a single interface brings up several issues that must be considered. The first issue is efficiency. By creating a signature that only deals with strings you are forced to spend processing time converting to and from textual representations.

Fortunately for us efficiency is not an issue for most GUI designs since most of the time they are being driven at human speeds and the computation time for parsing becomes trivial. Following the 80-20 rule [8], 80% of the time we are not going to care. For the 20% of the time when we see degradation in response time, we drop back to the JNI API, simply trading maintainability for efficiency.

The second issue is type safety. By forcing everything into a string we lose valuable type information. This concern can be minimized by hiding most of the string manipulation logic in low-level base classes that are not used directly by anyone. In the derived classes our data will be strongly typed and the fact that it gets converted to strings before it is used will be transparent.

The last issue is runtime errors vs. compile time errors. I have to admit this one bothers me. Derived classes will have to put arguments into commands via keywords and

get return values from commands via keywords. If the keyword is not present, we can throw an exception. We have traded compile time errors for runtime errors by generalizing the signature. This is true of many Java package interfaces. If you are going to program in Java you have to get used to dealing with runtime error checks. This may become less of an issue if templates are added to the language but for now it is a harsh reality.

11 NATIVE MESSAGE

This class will handle put operations for various data types and tedious message formatting details. The keyword-value pairs will be stored as elements in an associative `TreeMap` container:

```
public NativeMessage {
   …
   public void put(String key, boolean value) {…}
   public void put(String key, long value) {…}
   public void put(String key, double value) {…}
   public void put(String key, String value) {…}
   …
   protected String formatMethodArgs() {…}
   …
   private TreeMap theMethodArgs;
   …
}
```

Java makes string manipulations very easy so these operations result in single line implementations:

```
public void put(String key, long value) {
   put(key, (new Long(value)).toString());
}

public void put(String key, String value) {
   theMethodArgs.put(key, value);
}
```

We also defer get operations along with parsing the return arguments to NativeMessage. The keyword-value pairs will also be stored in a TreeMap. Note that since we cannot overload based on return type, all the get methods must be unique:

```
public NativeMessage {
   …
```

```
    public boolean  getBoolean(String key) {…}
    public long     getLong(String key) {…}
    public double   getDouble(String key) {…}
    public String   getString(String key) {…}
    …
    protected String parseReturnArgs() {…}
    …
    private TreeMap theReturnArgs;
    …
}
```

You can extend this interface to cover as many primitive data types as desired. The more that are available the easier it is for derived command classes to put and get arguments. For instance, implementing pass-throughs for different array types is very useful.

12 PASSING USER-DEFINED TYPES

The interface (at this level) is very data-oriented and not conducive to passing higher-level user defined types. For the MVC architecture, primitives suffice most of the time. GUI widgets typically deal only with low-level primitives as opposed to user-defined class types. For example, "the mouse is at this (X,Y) location" or "the edit field contains this string".

Sooner or later, you will need to pass a higher-level object across the interface. To support this, we provide a Memento interface that a Java class can choose to implement. A memento captures an object's internal state so it can be restored later [9]. For our interface, it is sufficient to have an object that knows how to save its state information to a string and restore itself from a string:

```
public interface Memento {
    public String   saveTo();
    public void     restoreFrom(String);
}
```

We can modify the `NativeMessage` class to allow any class that implements the Memento interface to save its state into the method arguments string or restore its state from the return arguments string. This assumes that you have a corresponding C++ class that can restore and save itself using the same format:

```
Public class NativeMessage {
    …
    public void put(String key, Memento value);
    …
```

```
    public String getMemento(String key);
    …
}
```

13 NATIVE COMMAND

With the string manipulation logic properly hidden in the `NativeMessage` base class, we can focus on the details of the `NativeCommand` class. Its execute method will defer to a native `realExecute` method that takes and returns a string.

The execute operation can be broken up into four discrete steps. There is the *pre-execution* phase where the derived command class puts the appropriate keyword-value pairs into the base container. There is the *execute* phase where the method arguments are formatted into a string and passed to the native `realExecute` method. There is the *return phase* where the return arguments are parsed and stored into keyword-value pairs. Finally, there is the *post-execution* phase where the derived class gets the appropriate keyword-value pairs from the base class's return arguments container. We control these steps by making abstract methods for the pre and post execution phases and letting `NativeCommand` provide the rest of the control flow:

```
public class NativeCommand {
    public void execute() {
        preExecute();  // put method args

        String args = formatMethodArgs();
        String returnArgs = execute(args);
        parseReturnArgs();

        postExecute();  // get return args
    }

    protected abstract void preExecute();
    protected abstract void postExecute();
    protected abstract String nativeProxy();
    …
}
```

Notice we also have an abstract method for the native proxy. The derived class knows exactly which C++ class provides the implementation for its command and that information needs to be part of the message passed through JNI so the proper C++ object responds to the command.

14 EXAMPLE COMMAND

Given this framework, issuing a Java Command to a C++ object becomes trivial. We simply need to know what keyword-value pairs need to be passed to our C++ implementation for it to do its work. We also have to know what arguments the implementation will send us in response. Finally, we need to know the name of the C++ class that provides the implementation as well as the library that needs to be loaded for it to work.

For our battery example, the first command the Java JBattery will execute is to construct the corresponding native C++ Battery. Our style will be to create a subclass of `NativeCommand` called `ConstructBattery` that is nested in JBattery:

```
public class JBattery {
   public JBattery(int power) {
      NativeCommand cmd =
         new ConstructBattery(power);
      cmd.execute();
   }
   …
   class ConstructBattery extends NativeCommand {
      private int  thePower;
      private long theBattery;

      public ConstructBattery(int power) {
         thePower = power;
      }

      public void preExecute() {
         put(“power”, thePower);
      }
      public void postExecute() {
         get(“this”, theBattery);
         if (theBattery == 0) throw…
      }

      public String receipient() {
         return “BatteryImpl::ConstructBattery”;
      }
   }
   …
   static { System.LoadLibrary(“legacy”); }
}
```

The formatted string that is sent to C++ as the method argument looks like this:

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

The resultant string returned from the native method would look like this:

```
BatteryImpl::ConstructBattery { this 67588000 }
```

15 SUMMARY

We have created a simple mechanism for Java to execute native commands. The interface is general enough to send a wide variety of arguments with an intuitive interface. Inspection of the `ConstructBattery` class shows the Java programmer does not need to know anything about the JNI or C++. You just need to know the class that provides the implementation, the input and output arguments, and the library it resides in.

Next month we will show what happens on the C++ side to service this native command. We will show how to use pluggable factories [10] to map a Java native call to a C++ object. We will go beyond passing commands and show how to attach controllers to events for notification of state changes in your model.

Complete source listing and readme files are available for download at http://www.jot.fm/issues/issue_2002_07/column7/jnis.zip.

REFERENCES

[1] Buschman, et. al. A System of Patterns, Wiley, 1996.

[2] Linden, Just Java 2, 4th Ed., Sun Microsystems, 1999.

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

[4] Silberschatz and Galvin, Operating System Concepts, 5th Ed., Addison-Wesley, 1998.

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

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

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

[8] Meyers, More Effective C++, Addison-Wesley, 1996.

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

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

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