ACHIEVING PORTABILITY FOR LEGACY SOFTWARE USING JAVA

D. Kelly Cooper
TYBRIN Corporation
Air Armament Center
96th Communications Group
Eglin Air Force Base

ABSTRACT

Increasingly, many software developers are facing the challenge of adapting software applications developed on one platform to work on multiple platforms. While software standards have helped this effort, they do not go far enough, and many platforms only partially support these standards leaving many needed functions in platform specific libraries. This is particularly evident in the areas of graphics and user interfaces, threading and synchronization, and in network and file access. Fortunately, Java offers a common interface where native libraries diverge. This paper outlines a phased strategy for migrating platform specific applications to be platform independent while reusing the robust, existing algorithms.

KEY WORDS

Cross Platform, Java, JNI, C, C++

INTRODUCTION

For almost three decades, the 96th Communications Group of the Air Armament Center has developed thousands of telemetry (TM) and time-space-position information (TSPI) applications to support its customers. Most of these applications perform the same basic processing steps: read and decode TM and TSPI data, correlate and process data, and display the results in user defined interfaces. The oldest applications originated on VMS machines, were written in FORTRAN, and used DECWindows displays. Later, most of these applications were ported to C/C++ applications on the SGI Unix platform and used X/Motif and GL displays. After OpenGL released its standard, the displays were converted again in an effort to be more cross-platform compatible. More recently, as PCs have become more powerful, efforts are being made to distribute these applications among Linux and Windows platforms, requiring yet another port. Linux ports from SGI Unix applications have compatibility issues in threading interfaces, X windows calls, and some socket and file calls. Windows ports also require threading, I/O, and displays to be rewritten, but to a greater extent than Linux ports. In many cases, the requirement demands that these applications are capable of running on more than one platform, further complicating development and maintenance. The options are to maintain separate versions of the application for each platform, or to insert C/C++ preprocessor directives that select alternate processing paths based on the detected operating system. Rewrites are expensive and time consuming — the most costly part being regression testing. Many of the libraries used in this software have been tested over decades of service and have demonstrated a high level of reliability. Rewriting these routines would require years of rigorous and extensive testing to establish the same level of trust.

Because the costs and schedules are so great for complete software ports, a more desirable solution is sought. One solution is to rewrite only the platform-specific portions of the application and reuse the platform-independent libraries. These libraries contain the mathematical data models derived from the TM and TSPI data, which are true abstractions independent of any particular OS or language. With this approach, the complex, time tested data models are preserved intact while the problematic portions of the application are reduced to those truly platform-dependent issues. An investigation of the Java technology shows promise in resolving all these issues. Java provides platform-independent threads, sockets and files, and user interfaces. Furthermore, using the Java Native Interface (JNI), Java provides access to the native libraries.

A MIGRATION STRATEGY

The following describes a strategy being used in parts of the 96th Communications Group to migrate some platform-dependent applications to platform independence. This strategy leverages the cross-platform Java libraries against the well established legacy libraries and routines. In order to apply the technology, the legacy routines are divided into separate categories that work well with the Java GUI architecture. The high-level GUI library for Java is *javax.swing*, which is rooted in the model-view-controller (MVC) architecture, so the legacy routines are divided along these lines. The portions of the application that deal with displays and user interfaces belong in the view category, while the logic that contain the mathematical models, data processing, and real world abstractions belong in the model category. That leaves the logic dealing with flow control, threading, and file and socket IO, etc. for the control category. If the original application used a procedural design instead of an object-oriented (OO) design, then some functions might mingle logic that fits into multiple MVC categories. The task is to identify which portions of the logic belong in which category. After the logic is categorized, the interfaces are designed; the models are created, and the application is linked together with the native libraries using control structures. Figure 1 illustrates the relationship between the MVC components.

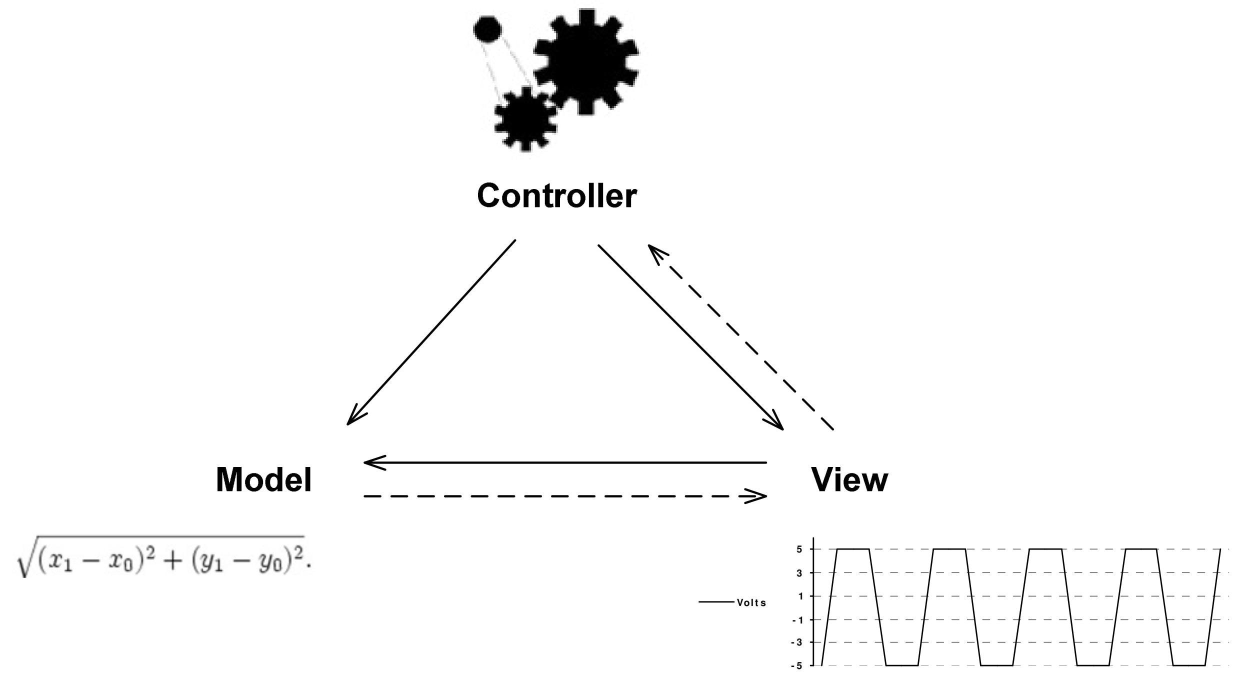

Figure 1. Model View Controller Pattern

The application centers on the information contained in the model. The controller receives input data and updates the model. The model notifies the view that it has been updated. User interaction with the view sends events to the controller which can update both the model and the view according to the user request. The view listens for changes to the model and the controller listens for changes to the view. The goal is to separate the original application logic into categories that follow the MVC pattern.

To start applying the migration strategy, the developer must create new interfaces and displays in Java, based on the layout of the original software screens. At first, no functionality is included in the interfaces. The components are statically laid out on the screens; with the behavior added later. This is the most straightforward step of the conversion. To accelerate this process, there are several free cross-platform tools and technologies available. For example, the Eclipse community provides an open source IDE that can manage Java projects as well as C/C++ and other languages. Another popular tool is NetBeans, a free Java IDE that is distributed with JDK that includes drag-and-drop GUI building. The Swing library was designed to work well with GUI building tools like NetBeans. Visualization displays also have extensive support in Java. Java comes standard with 2D graphics support, but there are also Java extensions for 3D graphics. For example, Java3D is an object-oriented 3D visualization API that uses scene graphs. If the original application being converted uses OpenGL displays, the Java OpenGL (JOGL) extension wraps OpenGL calls so they can be called directly from within Java. This can save time since the OpenGL calls in C can be reused in the Java source.

Once the displays and interfaces are created, the models are defined. Since the majority of the data processing is performed in the legacy libraries, only the data used for displays is needed in the Java models. The models contain functions for data retrieval and modification, but they do not contain the calculations that manipulate the data. The processing of the data will remain in the established native libraries.

Java and the Swing library provide strong support for data-centric application development. Many of the components in Swing support plug-ins for user-defined models. For instance, the JTable component is supported by a TableModel. If no model is specified, the JTable uses a default TableModel. In this step, built-in component models are replaced by custom models. The following code illustrates creating a table that uses a custom model where MyTableModel implements the TableModel interface or extends the AbstractTableModel class:

```
MyTableModel model = new MyTableModel();
JTable       table = new JTable(model);
```

For the components that do not contain built-in models, the Observer design pattern is implemented in the *java.util* library. The following code creates a custom JLabel that implements the Observer interface then creates a custom model that extends the Observable class and adds the label as an observer:

```
MyLabel label = new MyLabel();
MyModel model = new MyModel();
model.addObserver(label);
```

Both of these approaches support component notification when the model has changed. If these features are implemented, the visual components do not require management. Once the models are updated, the associated visual components are updated, so changes to the models are reflected automatically on the interface. This is a particularly useful feature for updating components that display changing data in real-time. The use of models, in general, benefits the application structure, making maintenance more manageable.

After the displays are laid out and the models are created, the application is linked together with control structures. In this final step, all the data processing threads and component listeners are defined to provide the functionality of the application. Because the existing native libraries provide the core processing of the data and models, JNI functions must be created to that link the Java objects to the native routines. (The full specification of JNI is cited in the references at the end of this text. Only the basic functionality of JNI is covered here.) To begin establishing the native interface, a Java member function is declared in the class definition, similar to the following example:

```
private native void save(byte[] b, int size);
```

First, notice the usage of the **native** keyword specified in the declaration of the *save()* function. Use of this keyword indicates that *save()* is defined in C code, using JNI. Furthermore, *save()* passes a byte array and an integer to the native function as arguments. Next the underling native code is created to implement this function, beginning with the C/C++ header file. Figure 2 illustrates the JNI data flow from source to runtime execution.

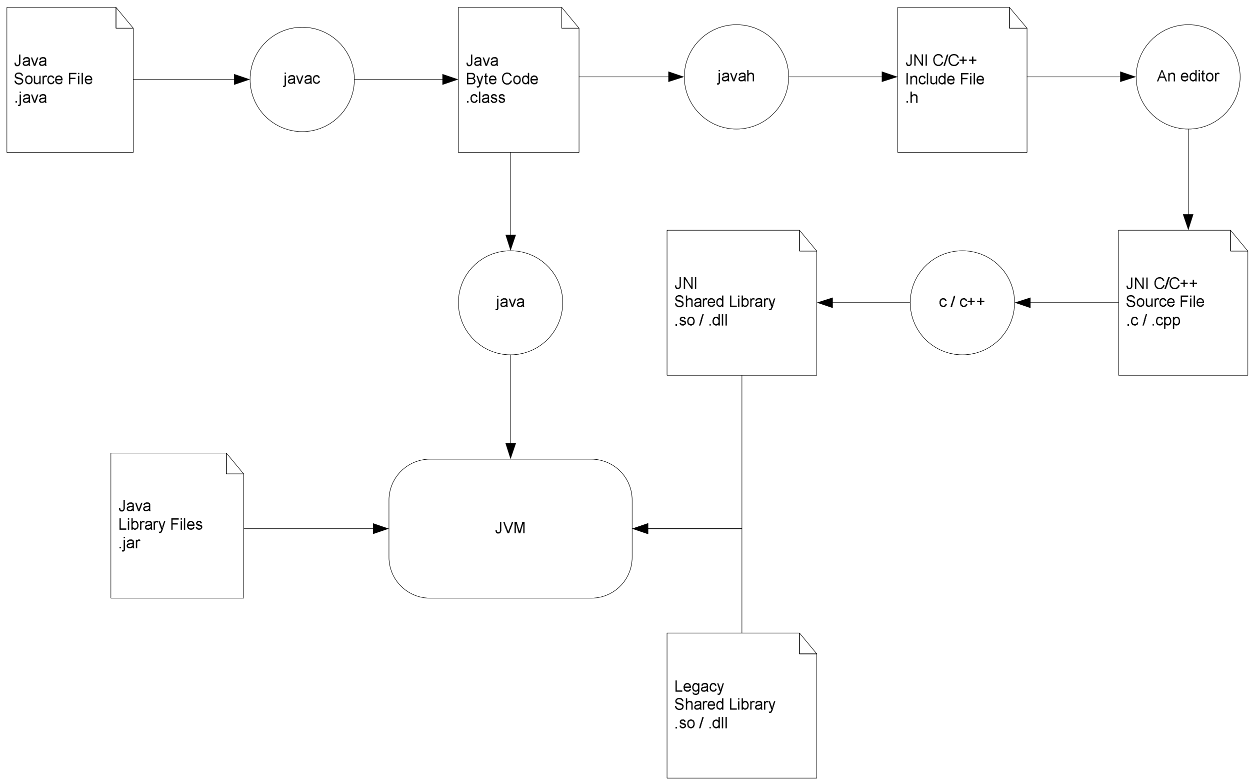

Figure 2. Java Native Interface Data Flow

To auto-generate the C/C++ header file for this Java class, first compile the class with *javac*, and then execute *javah* against the class file. The *javac* command takes a Java source file name with a .java extension, but the *javah* command requires a Java class file name without the .class extension. The *javah* command creates a file with an .h extension that contains C declarations of all the native functions declared in the Java class. The implementer manually creates a C/C++ source file that includes this header file and creates the definitions for all the functions declared in the header file. The native code includes the established libraries that decode and process the data stream. If the Java class is called TMReader and the native libraries are defined in a header called TMLib, the C++ implementation for the native function above might look something like the following:

```
#include “TMReader.h”
#include “TMLib.h”

JNIEXPORT void JNICALL Java_TMReader_save(JNIEnv* env, jobject,  jbyteArray jbuffer, jint jsize) {

    const char* cbuffer = (char*)env->GetByteArrayElements(jbuffer, 0);

    TMLib::getInstance()->decode(cbuffer, jsize);

    env->ReleaseByteArrayElements(jbuffer,(jbyte*)buffer, 0);
}
```

Here the TMLib is a Singleton class that decodes and stores the data for later retrieval. Again the JNI specification defines how to interface to a native library in detail, but the basics are simple. All JNI functions include in their definition a *JNIEnv* parameter, which is a handle to all the functions required for interfacing with Java. They also include a *jobject* parameter, which is a handle to the Java object that contains the native function. The *jobject* parameter is not used here, but it can be used to access any data member or call any member function in the Java object, using the functions included in the *JNIEnv* parameter. The remaining parameters are passed directly from the Java source; in this case a *jbyteArray*, corresponding to *byte[]* in the Java source, and a *jint* corresponding to the primitive Java type int. When using arrays or Java objects within the C code, the object must be locked before it can be used. In the example, *GetByteArrayElements()* locks the byte array so it can be used, and *ReleaseByteArrayElements()* updates any changes to the array and releases the lock.

Once the native functions have all been defined, they need to be compiled to a shared library. Shared libraries have platform-specific naming conventions. For example, if the shared library is a Win32 platform and the library is named MyJNI, the filename convention is MyJNI.dll. If the same shared library is on a Unix-like platform, the filename convention is libMyJNI.so. The library is statically linked to Java by including the following Java statement in the class definition:

```
static {
    System.loadLibrary("MyJNI");
}
```

The native function is accessible from within Java, like any other Java function. For better platform independence, the data streams must first be read with Java; then the data buffers are passed to the native routines for processing. This provides platform-independent IO while preserving the proven native libraries. To accomplish this, a class is created that implements the Runnable interface, and then the run function is defined in a manner similar to the following code:

```
public void run(){
    try {
        packet = new DatagramPacket(new byte[MAXSIZE], MAXSIZE);
    }catch (SocketException e){}

    if(packet == null) return;

    for(;;){
        try{
            socket.receive(packet);
        }catch(IOException e{}

        synchronized (syncObject){
            save(packet.getData(), packet.getLength());
            syncObject.notifyAll();
        }
        Thread.yield();
    }
}
```

In this example, the socket receives the TM or TSPI data and forwards the entire packet to the native interface for processing. Since native libraries that deal with synchronization and mutual exclusion are usually platform dependent, the synchronization is done in Java to avoid the platform specific C calls. Because the *save()* function is enclosed in the synchronized block, other synchronized threads can access natively stored data without requiring platform-dependent locking in the native library. The *notifyAll()* call tells the other synchronized threads that a packet was received and processed, while the *yield()* call allows other threads the opportunity to run.

Now supposing the native library which decoded the incoming data has a routine that returns a status string. Another Java class is created which implements Runnable and has a native function to retrieve the data, similar to the following:

```
private native String getStatus();

public void run(){

for(;;){
    synchronized (syncObject){
        try {
            syncObject.wait();
        } catch (InterruptedExceptione){}

        model.setStatus(getStatus());
    }
}
```

Here the *wait()* call blocks the threads execution until *notify()* or *notifyAll()* is called on the *syncObject*. If the model object is implemented as described in the MODEL section above, the associated GUI component is updated automatically by the *setStatus()* call. The native implementation might resemble the following:

```
JNIEXPORT jstring JNICALL Java_TMDisplay_getStatus(JNIEnv* env, jobject) {

    return env->NewStringUTF(TMLib::getInstance()->status());
}
```

Java component listeners can call native functions in the same way these illustrated threads do. To complete the migration strategy, the GUI functionality is added by implementing the component listeners.

CONCLUSION

This completes the cross-platform migration strategy. The Java interfaces and displays are created, and models back these components, where required, to display the TM and TSPI data. Synchronized threads read and process the data streams using JNI to interface to existing native libraries. The organizational native libraries are retained to preserve the algorithms reliability and trustworthiness and also to accelerate the platform-independent migration. Java IO classes read the data streams and pass the data to the existing native libraries for processing. Component listeners allow users to interact with the data, using JNI calls where necessary. The final implementation provides a data-centric architecture based on MVC principles. For the sake of time, the only portions of the original application that are ported to Java are those that derive a cross-platform benefit. This strategy leverages the platform independence of Java with the reliability of the time-tested organizational logic, while reducing the time to market for a cross-platform application migration.

The Test and Analysis Division of the 96th Communications Group is having success using Java to migrate applications from platform dependence to platform independence. Java is a modern OO language supported by a mature library suite. The large Java community contributes to this standard with many specialty packages that extend the capability or application of the language. For example, vendors are providing extension packages for specialized areas like graphics, useful development tools and utilities, and even Java compilers that support hard real-time development. In addition, JNI allows the advanced features of Java to interact with established native libraries that contain the organizational knowledge. The JNI feature is particularly powerful because it allows developers to leverage the best Java has to offer with the strength of established native libraries that have benefited from years of testing and refinement.

REFERENCES

http://www.eclipse.org/
http://www.netbeans.org/
http://java.sun.com/products/jfc/tsc/articles/mixing/
http://en.wikipedia.org/wiki/Java_3D
http://en.wikipedia.org/wiki/Java_OpenGL
https://jogl.dev.java.net/
http://java.sun.com/blueprints/patterns/MVC-detailed.html
http://java.sun.com/docs/books/tutorial/uiswing/components/model.html
http://java.sun.com/products/jfc/tsc/articles/architecture/
http://en.wikipedia.org/wiki/Model-view-controller
http://en.wikipedia.org/wiki/Java_Native_Interface
http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html