Astronomical Data Analysis Software and Systems VIII
ASP Conference Series, Vol. 172, 1999
D. M. Mehringer, R. L. Plante, and D. A. Roberts, eds.

A Java–IRAF Spectrum Visualization Tool

I. Busko

Space Telescope Science Institute, Baltimore, MD 21218, Email: busko@stsci.edu

Abstract. A Java application designed to display spectra from Hubble Space Telescope's STIS instrument relies on a Java–IRAF interface to access the data in a efficient way. This paper describes this interface implementation.

1. Introduction

A simple spectrum visualization tool is being developed by the Science Software Group at the Space Telescope Science Institute (STScI), basically to evaluate the capabilities (and weaknesses) of Java as a possible environment for development of data visualization and analysis tools tailored to Hubble Space Telescope (HST) data.

One of the problems with such a tool is to cope with the varied gamut of formats on which HST data exist. Since HST data in all their forms can be readily accessed by IRAF/TABLES I/O libraries, to save development time and to avoid current inefficiencies in Java I/O, we choose to link the Java code to the IRAF/TABLES I/O libraries using the Java Native Interface (JNI) facilities.

2. The Java–IRAF Interface Design

The interface must be implemented in two levels because currently the JNI only supports native code written in C/C++. The Java–IRAF interface functions must thus be coded in C. Since IRAF/TABLES libraries are written in SPP (a Fortran pre–processor), C bindings to allow the C native programs to access IRAF/TABLES libraries must be used as well. The availability of the C–VOS bindings developed by A. Farris at STScI conveniently solved this part of the problem. The layer structure of the resulting software is shown in Fig. 1.

The C–VOS is made of C–callable functions that map one–to–one to functionally equivalent IRAF/TABLE routines. C–VOS functions perform actions such as opening a table file, reading blocks of pixels from 2–D images, and so on. This can be accomplished with efficiency because both software layers, C and SPP, share the same memory address space (the physical machine memory where both run). Thus most of the actions involving data arrays can be performed by just passing memory pointers. An additional benefit of such "language binding" concept is that it becomes relatively easy to automate the process of generating the actual binding code.

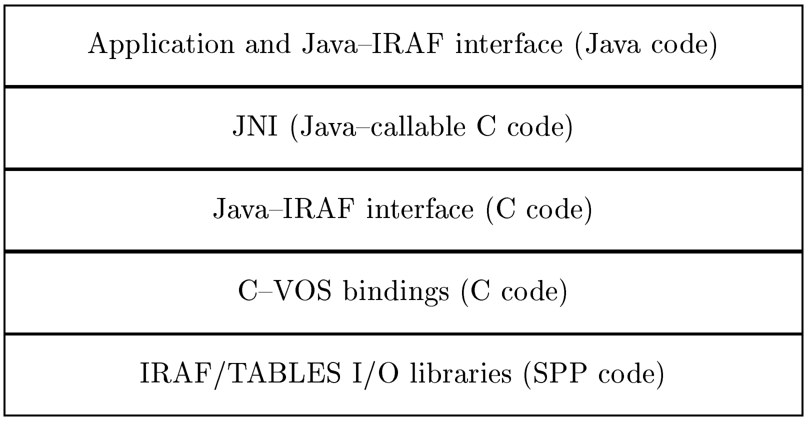

Figure 1. The layer structure of the software.

Because Java code is run by the Java Virtual Machine (JVM), the JNI has to cope with the problem of a virtual address space (the Java–accessible data) that interfaces with a physical address space (the C–accessible data). The methods provided in the JNI to handle data arrays rely on memory "pinning" and data copying, thus they can lead to inefficiencies, at least on some JVM implementations.

Because of these potential inefficiencies, the adopted approach in designing the Java–IRAF interface avoids the concept of language binding. Instead, the required interface functions to support the spectrum visualization application are defined at a much higher level. No low–level functions (in terms of IRAF/TABLES libraries) such as "open table" or "write array in row/column table cell" were implemented. Instead, functions such as "get spectrum" or "get table header contents" are used to minimize traffic between the Java and C portions of the code. Each native method is responsible, using IRAF/TABLES I/O functions, to access all required data in the file system, and, using JNI functions, to build or access high–level Java objects (such as "Spectrum") defined in the Java side. Thus they don't act as simple bindings between equivalent low–level functions, but perform quite a lot of processing and data formatting on their own.

3. Example: Reading a STIS 1–D Spectrum

The Application Side

The application code that reads a STIS 1–D spectrum resorts to a static method defined in a "SpectrumFactory" Java class. The spectrum is returned packaged inside a Spectrum object:

```
...
...
try {
    Spectrum sp = SpectrumFactory.MakeSpectrum (...);
} catch (Exception e) {
...
```

...

The Interface: The Java Side

The SpectrumFactory class performs two functions: it declares all native methods, and it loads the native library. The IRAFInit method is responsible for initialization of the IRAF/TABLES libraries after they are loaded. IRAFInit must be run only once throughout the library's life, thus it is called from the class' static initialization block:

```
public class SpectrumFactory {

    private native static void IRAFInit();

    public static synchronized native Spectrum MakeSpectrum (...)
        throws IOException, ArrayStoreException;

    static {
        try {
            System.loadLibrary ("iraf");
            IRAFInit();
        } catch (Exception e) {
            ...
            ...
        }
    }
}
```

The Interface: The C Side

The MakeSpectrum native method reads the spectral data from the STIS 1–D file and builds the Spectrum object. The IRAFInit method just calls the C–VOS initialization function. Notice the include file created by the *javah* header generator program.

```
# include <jni.h>
# include "spectrum_SpectrumFactory.h"           /* created by javah */
# include <stdio.h>
# include <c_iraf.h>
# include <xtables.h>

JNIEXPORT void JNICALL
Java_spectrum_SpectrumFactory_IRAFInit (JNIEnv *env, jobject obj) {
        c_irafinit(0,0);        /* IRAF initialization */
}

JNIEXPORT jobject JNICALL
Java_spectrum_SpectrumFactory_MakeSpectrum (JNIEnv *env, jobject obj, ...) {
    jclass clspec;             /* JNI data type   */
    jobject spectrum;          /* JNI data type   */
    jmethodID mid;             /* JNI data type   */
    IRAFPointer tp;            /* C-VOS data type */
    ...
    ...
    /* Open and read table using C-VOS functions. */
    tp = c_tbtopn (name, IRAF_READ_ONLY, 0);
    if (c_iraferr()) {
        (*env)->ThrowNew (env, "Input table could not be opened.");
        return ((jobject)NULL);
    }
    ...
    ...
    /* Build and populate Spectrum object using JNI functions. */
    clspec = (*env)->FindClass (env,"spectrum/Spectrum");
    mid  = (*env)->GetMethodID (env, clspec, "<init>", "(I)V");
    spectrum = (*env)->NewObject (env, clspec, mid);
        ...
        ...
    return (spectrum);
}
```

The Interface: The Makefile

Below is the Solaris makefile that builds the interface code in this example, using file path names from the STScI environment. The order in which the linker searches the libraries is very important.

```
INCLUDE=/usr/local/jdk1.2/include
LIBS=/usr/local/jdk1.2/jre/lib/sparc/libjava.so \
        /usr/ra/stsdasx/lib/libhstio.a  /usr/ra/stsdasx/lib/libcvos.a \
        /usr/ra/tables/lib/libtbtables.a /usr/ra/irafx/lib/libex.a \
        /usr/ra/irafx/lib/libsys.a /usr/ra/irafx/lib/libvops.a \
        /usr/ra/irafx/unix/hlib/libos.a  /usr/ra/irafx/unix/hlib/libboot.a \
        /opt/SUNWspro/SC4.0/lib/libF77.a /opt/SUNWspro/SC4.0/lib/libM77.a \
        /opt/SUNWspro/SC4.0/lib/libsunmath.a /usr/lib/libdl.so.1 \
        /usr/lib/libelf.so.1 /usr/lib/libnsl.so.1 -lm -lsocket

build: SpectrumFactory.h MakeSpectrum.o lib

SpectrumFactory.h:
        javah -jni spectrum.SpectrumFactory

MakeSpectrum.o: MakeSpectrum.c spectrum_SpectrumFactory.h
        cc -c -I$(INCLUDE) -I$(INCLUDE)/solaris -I/usr/ra/stsdasx/lib \
        MakeSpectrum.c -o MakeSpectrum.o

lib: MakeSpectrum.o
        ld -G -t MakeSpectrum.o $(LIBS) -o libiraf.so
```

The first step builds the include file with the header generator program *javah*. The C code is then compiled with appropriate paths for both Java and IRAF/TABLES include files. Finally, the object code is linked with the Java and IRAF/TABLES libraries into a single shareable library named libiraf.so. This name is used by the Solaris OS to find at run time a library identified in the Java code by "iraf" (as in the System.loadLibrary call).