**Jim McGill Brain Dump**

**Introduction**

This is a complete a core dump of what I worked on while employed by Joe Felsenstein. It’s organized by topic rather than historically since I bounced around between topics as needed. Most of this work was done in Java and the primary development tool I used was Eclipse. I used the Indigo release of Eclipse and did all development on a Mac using the Lion operating system. It is generally assumed in this document that you will look at the Java and C++ source code for details and are reasonably savvy with Java. Software topics I mention here are either not obvious or painfully gained.

**PHYLIP 3.695**

The PHYLIP 3.695 release had been held up for >8 years when Mac dropped support for Carbon which meant that Drawgram and Drawtree no longer worked and Joe had no software help to solve the problem.

When I joined his group, the first project was to get Drawgram’s and Drawtree’s plotting working again so 3.695 could ship. It quickly became clear that the previous solution, which was running different plotting software based on the operating system being run, was no longer viable because so much had changed in the 10+ years since the software had been written. A bit of investigation on the web pointed me to Java and the Java Native Access / Interface (JNA / JNI) package that was written by Sun to link Java interfaces to existing C/C++/Fortran software. It was developed by Sun to allow legacy code to receive a modern GUI interface without having to be rewritten entirely in Java. I will discuss the details of the Java development cycle below under “**Building a PHYLIP Interface**”.

Once I got the Drawgram and Drawtree interfaces running on a Mac I ported them to a PC and to an Ubuntu Linux machine. Only real problem I found was that the PC definition of how boxes are plotted is slightly different than the Linux version, so all buttons, data entry boxes, etc. expanded 1 pixel in all directions. This made them collide in my first version so I went back to my Mac and moved everything apart.

Once Drawgram and Drawtree were working on all 3 operating systems, I modified the Make files for all 3 systems (Makefile.osx, Makefile.unx, Makefile.cyg) to build PHYLIP correctly, we bundled all the executables, and made the release. There is a document in the phylip3.6 CVS called “**Phylip3\_695Release.docx“** that covers the details of the release process. The html help documentation was also updated to include screen shots of the Drawgram and Drawtree interfaces.

**PHYLIP 4.0 Interfaces**

We were happy enough with the Drawgram and Drawtree interfaces in 3.695 that once it was released, I started putting Java interfaces on all the rest of the PHYLIP programs. Most of them now have working interfaces, but a few (Contrast, Dnamove, Dolmove, Move, and Restml) do not because they were either still in flux or needed extensive redevelopment. All that have working interfaces are also documented in the html help documentation and screen shots of their interfaces are in the /doc/images directory.

All of the PHYLIP programs except Retree follow the pattern described below in “**Building a PHYLIP Interface**”. Retree is the first PHYLIP program that was completely rewritten in Java. This is because it is so interactive that it made sense to do everything in the Java code rather than passing off computations to the C++ code. As such it will be the prototype for the move part of the more interactive PHYLIP Move family of software (Dnamove, Dolmove, and Move).

**Current PHYLIP 4.0 Development**

Here is a list of the currently active PHYLIP development projects:

1. PHYLIP super program

We added a super Java program called Phylip which can call all of the existing PHYLIP Java interfaces. Currently it has an alphabetical set of buttons for each PHYLIP Java interface which, when picked, starts up that interface and passes the calling routine name (phylip) to the called interface. This allows the interface to know it should shut down its interface but not exit when the user tells it to quit.

The future plan is to add a tab interface to the PHYLIP interface and put common PHYLIP pathways on other tabs so the user can step through them in a logical order.

1. PHYLIP chains

These will allow the user to chain multiple PHYLIP routines together and will support the PHYLIP pathways discussed in “PHYLIP super program” above.

The prototype is found in Restml. It has links it to Dnaml, Drawgram, and Drawtree, the latter two purely to allow testing of the trees that Restml outputs.

Restml can output multiple trees to an outtree file. These can then be read by Dnaml, DNA data from an infile can be applied to the trees, and the best tree discovered. Much more could be done, but that’s where we are currently.

1. Java interface memory between runs

In order to support the PHYLIP chaining methods, each PHYLIP program needs to save and restore its current settings when a run is started. The naming convention for the settings file is [nameofprogram]Init.txt. The prototype for this is found in Dnaml. Two buttons were added to the bottom left corner of the interface: “ Reset to Defaults” and “Read Init file”. Four functions were added to the Java code:

* resetDefaults() which contains all the initial values for the interface (these were transferred from the initialize() function).
* DoInit() which is called after initialize() in the application creation.
* getStoredSettings() which reads the stored “dnamlInit.txt” file and sets the interface values. If it does not find “dnamlInit.txt” it calls “resetDefaults()”
* saveSettings() which writes the “dnamlInit.txt” as a flat text file so it is easily read and edited. **One crucial warning!** The variable name and the variable value must be separated by the exact string “ : “ (that’s a colon with spaces on both sides). If you don’t use that, the parser in getStoredSettings() breaks. Yes, it should be more fault tolerant, but this is a prototype so it’s fragile.

Ultimately both getStoredSettings() and saveSettings() should be part of the utilities package and be driven with lists of labels and data, but it didn’t happen yet.

**PHYLIP Pure Java Programs**

There is currently only one of these, Retree. Probably easier to look at the source than for me to describe it.

**Building a PHYLIP Interface**

I’ve left this for last because it is a messy process and will give you headaches, whether writing, extending, or debugging it. Sorry about that. We are essentially grafting two incompatible software paradigms together, so the result is a bit gruesome. I’ll take things in the order they have to be done, ignoring the fact that it is a highly iterative and very buggy process. General approach is “copy and conquer”. I tend to use Clique as my prototype because it is small and covers most of the basics, both on the C++ and the Java side. But if, for example, you need to pass an array (something that JNA does not do well) you’ll need to look in Dnaml to see how that is done.

**General caveat**

PHYLIP is ancient code. Its roots go back to late 1970’s Pascal on batch mainframes (actually it’s origins are mid-1970’s Fortran, but that has all disappeared). Some of the arrays still reflect the limitations of IBM 80 column cards and 120 column line printers. Some indexing still reflects the auto translation from Pascal to C sometime in the mid-1980’s.

PHYLIP has been continuously supported, expanded, patched, and repurposed since then by at least 20 different programmers of varying skills and experience. So it is not uncommon to find 2 or 3 different ways of doing the same thing in the same code. Keep an open mind and be absolutely sure you know what something is doing before you modify it. Some of my most difficult to find bugs have been the result of “fixing” some obvious problem, only to discover that something downstream critically depended on a side effect of that “problem”.

There are a mind boggling number of things passed around as global variables, many with uninformative names like “ival”. Furthermore many globals are redefined locally (whether intentionally or not is often difficult to say) so moving code around can have wildly unexpected consequences.

Remember, you are just the latest in a long line of programmers to work on this code and there will be people after you. Comment things thoroughly as you go so if you get run over by a truck, someone can fix the code.

**Analysis of the C++ command line interface**

The first thing you have to do is to completely understand what the C++ command line interface can do. This is fairly difficult because options come and go within the command line run depending upon what options have been previously chosen. My approach is to print lots of screen shots covering every possible option and then walk through the C++ interface source and check that every possible pathway through the interface has been found. There are a couple things here that will drive you nuts. One is there are some command options that come from routines that are called from the interface (particularly those associated with reading or writing files, but also some math decisions). There are also hunks of dead code that can no longer be accessed. Do the best you can and assume there will be things to fix later.

**Building the Java interface**

Now that you have an analysis of what the command line interface can do, you need to build a Java GUI interface that covers all possible combinations. This will include a lot of enabling and disabling of options and associated text, depending of settings of other variables.

First you need to group functions together so they may visual sense. Generally I put all input and output files at the top, followed by various control values and put the standard Booleans like printing the input values at the end. If the list gets too long, I go to two columns and try to break at a logical place. Look at the existing interfaces to get the general look and feel.

Eclipse makes creating the actual interface fairly easy since you can drag and drop functional pieces. I found the MigLayout Grid structure to be immensely helpful because it automatically lines things up and spaces things correctly so they look good on PC, Mac, and Linux displays. Naming conventions are critical! The one I have used is [what an item is][informative name]. So, for example, btnInputFile is a jButton that kicks off the Input File dialog, and txtInputFile is the name of the current input file. Be verbose and refactor names as soon as they become uninformative. There are a huge number of properties that need to key off these names, so minimize your confusion up front by making it clear what they are.

Once you have the interface created, you need to pass the user entered data through to the C++ library that contains the C++ code you want to execute (look in the Makefile.osx to see how to create the libraries). You can pass as many arguments as you want through a JNA interface via a Native.loadLibrary call to the underlying C++ code (see one of my interfaces for the details). Be aware they all have to be the same type and in the same order as the C++ code expects. This is the basis of a lot of ugly crashes in JNA because it does no error checking. Only thing to do is printf everything out on the C++ end and make sure it is exactly what you expected.

One wrinkle is that in order to display the progress of the C++ program (which is written to a file called “progress.txt” as it goes) you must do the loadLibrary call on a separate thread. You’ll find working examples of that in all the PHYLIP programs that have Java interfaces, but it’s a bit tricky to get running if you aren’t already a Java hacker.

**Building the C++ end of the JNA**

JNA is a powerful package but there are limits to what it can do. It passes through the data created in the previous section and maps fairly sensibly, String to char\* for example and Boolean to int. But Java and C++ do not agree on what types mean. Float is problematic, always use double, but the worst one is Booleans where true and false are 1 or 0 in C++, but not 0 and 0 in Java. This means that a true in Java usually comes through as 0xFFFF in C++.

The first thing to do is create a new entry point in the C++ code (I name it the name of the PHYLIP function, so “void dnaml(…)”, for example. It has as arguments every variable you passed from Java, auto translated to a few C++ types. Don’t get creative, JNA is pretty limited. For example, passing an array requires you to write your own parser. It’s much easier to unwrap the array, pass individual variables, and then rebuild the array on the C++ end.

The debugging of this interface will occupy an inordinate amount of time. Part of the problem is you cannot interactively debug the C++ library code. Only thing you can do is put in printfs. I have found it necessary to printf every variable that comes from the Java code and, in parallel, printf every variable that comes from the command line interface, print them out and hand compare them. It is way to easy to misread things on the screen, paper is the only way to be sure you have things right.

Beyond that, look at something I’ve written. You’ll find all sorts of commented out printf statements. Pretty much anywhere things can go wrong, they will, and working at a remove with only printfs can be maddening. If you find something funky in my existing interface code, it’s probably there for a reason. Figure it out before you mess with it.

**Getting data back to Java**

You can return one value from the C++ code. Given the problems of translation between C++ and Java, that is usually not much use. I resorted to the primitive method of writing out text files with generic names like “progress.txt” or “JavaPreview.txt” and reading them in the Java code on a different thread or after the C++ code exits. Works fine but is a bit clunky. Luckily most of the data returned from the C++ code is pretty small.

**PopG**

PopG is an entirely independent piece of software that Joe uses for teaching. The Java version was initially written as a summer project by Ben Zawadzki under my supervision. He got most of the basic pieces running. I then finished it and got it packaged for release. It is a complete Java rewrite of the old C++ program. Should be pretty solid at them moment, but if something breaks, I tried to leave good comments in it.