Tracking Down Java Class File Dependencies

by Alan Oursland

I've been developing in Java for the past year and a half. I have found Java to be a wonderful language in most respects and horribly flawed in a few areas. This application addresses one of the problem areas I've found in Java: runtime linking. I will explain what I mean by that in a moment. If you need help on any of the Java topics I talk about, or if you need help in running the program, please read Marshall Brain's introduction to Java located on DevCentral here.

When you write and compile a program in C++, there is a link stage that matches all of your function calls to the appropriate calls in all of your different files. The linker then places everything into a package that is guaranteed to have all of your code (the executable). This compile time linking guarantees that all of your code is available in the executable (with the exception of DLLs, but I'm going to ignore that for now).

Java does a pseudo-link at compile time. If the functions you are calling in other classes do not exist, you will get a compile error. The compile process does not however package everything up into a nice little container. You are left to package all of your class files into a jar file yourself. If you somehow miss one of the class files, you will not know until the program attempts to use that class, at which time you will get a ClassNotFound runtime exception. If you are delivering the application to a customer, your testing needs to be absolutely complete to make sure that every single class is available (do you have any custom exceptions? Does your code exercise each error that exercises those exceptions?). When you use a jar file, you have no guarantee that all of your classes are available, and you will not know that a class is missing until it causes an error.

A similar problem is the case where the classpath does not include the location of a third party class used by your classes. If you were to receive a jar with poor documentation from someone, you would have no way of knowing what other jars are required.

The example tool with this article, JavaVizor-Link, helps to solve some of these problems. It will read a class file and find all of the classes used by that initial one. It will then repeat the process on all of the classes it finds. Lists of the classes that it could not find are recorded, and you are given a chance to tell it where to find those classes.

When you run this tool, you will see a fairly standard looking application with a menu and toolbar. There is also a tab pane with the following tabs: "Classes To Link", "Classpath", "Processed Classes", "Skipped Classes", and "Missing Classes". I am going to describe these in the order that you will probably want to use them.

First, look at the "Classpath" tab. This tab is initialized to your system classpath. The buttons on the right can be used to add new elements to your classpath, remove elements, and reorder the elements. The order determines the order in which the classes are found. If you have two classes with the same name, the one that is closer to the top of the list will be the one that is used. There is a checkbox next to each location. When the program finds a file in a location that has been checked, the program will find all of the classes used by that file and then insert that file in the "processed" list. If the file is in an unchecked location, the file is just added to the "skipped" list.

The "Classes to Link" tab has a list of classes and a list of jar files. The program will link each class in the class list. It will search for the class in the classpath. If the class is found, it is either placed into the "processed" list or the "skipped" list, depending on if the discovered location is marked for processing on the classpath tab. The program will do this for every class in each jar file that is listed. Note that unless the jar file is present in your classpath as well, none of the classes in the jar file will be found.

Now that you have your classpath and classes to link set up, press link. A progress dialog will appear and show you how many classes are remaining to process, how many classes were found, and how many classes are missing. When the progress dialog disappears, go to the "Processed Classes" tab.

"Processed Classes" shows you the files that were found in the classpath locations marked for processing. It lists the package, class name, and file location for each class. You can use this to make sure that it found each class in the location you expected. If a class exists in two places, and you happen to find the wrong one, you could get some bugs that would be pretty difficult to track down. This might happen if there are multiple version of the class on your machine (CORBA interfaces perhaps?).

"Skipped Classes" shows the classes that are used by your processed classes but that were not actually linked. These usually include the standard runtime classes in classes.zip, swingall.jar or rt.jar. If there are classes missing in those packages then we have bigger problems. By skipping these classes, we reduce the time it takes to link everything.

The last tab is "Missing Classes". This is probably the most useful tab. It shows the classes that are used but could not be located. Each line displays the class package, the name and location. The location is editable. If you realize the class is in a package that you forgot to add to your classpath, you can enter the location here. The relink button will verify the location of the class, add the location to your classpath, and move all of the missing classes at this location to the skipped class list. Once you have removed all of the extra classes, you will be left with a list of classes that are missing from your classpath. If no classes remain, then you know that the jar file is complete. This completeness is a guarantee, and you can pass that guarantee on to your customer. A side benefit of this process is that you also have a classpath that includes everything your program needs.

Additionally, the tool can save your class list, jar list, and classpath to a file. You can copy the classpath directly into a batch file that starts the tool (however, you will need to replace '\\' with '\' on Window's machines). The next time you need to link the program, you can load this file, and you won't have to deal with the entire process of entering everything.

I wrote this tool to solve a problem for me. I hope that you are able to solve similar problems for yourself.

Get the ZIP file containing JavaVizor-Link: file_depend.zip.

JavaVizor-Link runs under JRE1.2 or JRE1.3. Start the application using the class "JVLApp". A batch file is included in the ZIP file for your convenience. The batch file assumes that the JRE is installed in c:\jre1.3.1 and that the JVLApp jar is installed in c:\jvl.

Developed Under:
Java 2 Platform, v1.2