Appendix 1. Installation using install4j

This chapter uses install4j (http://www.ej-technologies.com/products/install4j/overview.html), a cross-platform tool for creating native installers for Java applications. install4j supports Windows, UNIX, Linux, and the Mac OS, although I'll only be creating installers for Windows. This is mainly to keep things simple, and because I only have regular access to Windows machines.

Any kind souls who have written example install4j installers for other platforms should contact me, and I'll include a link to their work in a future version of this appendix.

I'll develop installers for two examples from the book: BugRunner and Checkers3D. BugRunner comes from chapter 6 on 2D sprites; it uses the standard parts of J2SE *and* the J3DTimer class from Java 3D. Checkers3D, from chapter 8, is our first Java 3D example.

install4j can create an installer which includes a JRE, either as part of the .exe file or downloaded automatically from install4j's Web site when the installer first runs. However, I'll assume that Java is already installed.

It is altogether more tricky to create an installer for an application that requires parts of Java 3D, an extension which isn't included with J2SE or JRE.

The installers will be built with an evaluation copy of install4j Enterprise Edition v.2.0.7. It's fully functional, but adds several "this is not a registered copy" messages to the installation sequence.

Before starting, it's worthwhile to briefly compare install4j and Java Web Start (JWS), the subject of Appendix 2.

install4j creates a standalone installer for an application, which can be delivered to the user either on a CD or downloaded via a Web page link. A great advantage is the familiarity of the installation concept: double click on the .exe file, press a few "yes" buttons, and the application appears as a menu item and a desktop icon. The fact that the executable is coded in Java is irrelevant.

JWS is a network solution, which offers better protection from potentially renegade downloads, and supports application updates. JWS is typically utilized via the Java Web Start client that comes as part of the J2SE installation. The reliance on a network model seems overly restrictive, especially when applications are large. The possibility of being queried about security levels and updates is quite off-putting to novice computer users.

1. The Java 3D Components

In order to get BugRunner and Checkers3D to compile and run, we will need to include relevant bits from Java 3D with the applications. On Windows, the core elements of the OpenGL version of Java 3D consists of four JAR files and three

DLLs. The JAR files are j3dcore.jar, j3daudio.jar, vecmath.jar, and j3dutils.jar in <JRE_DIR>\lib\ext\. The DLLs are J3D.dll, j3daudio.dll, and J3DUtils.dll in <JRE_DIR>\bin\. <JRE_DIR> is the directory holding the JRE, typically C:\Program Files\Java\j2re1.4.2_02\.

If the Java 3D development kit is installed, the files will also be found below the J2SE home directory, which is usually C:\j2sdk1.4.2_02.

The OS level libraries will vary if the DirectX version of Java 3D is used, or the platform is Linux or the MacOS. The easiest way of finding out Java 3D's composition on your machine is to look through the Java 3D readme file, which is added to the J2SE home directory at installation time.

The BugRunner application only uses the J3DTimer, so which of the JAR and DLL files are required?

The J3DTimer class is part of the com.sun.j3d.utils.timer package, which is stored in j3dutils.jar (this can be confirmed by looking inside the JAR with a tool such as WinZip), as shown in Figure 1.

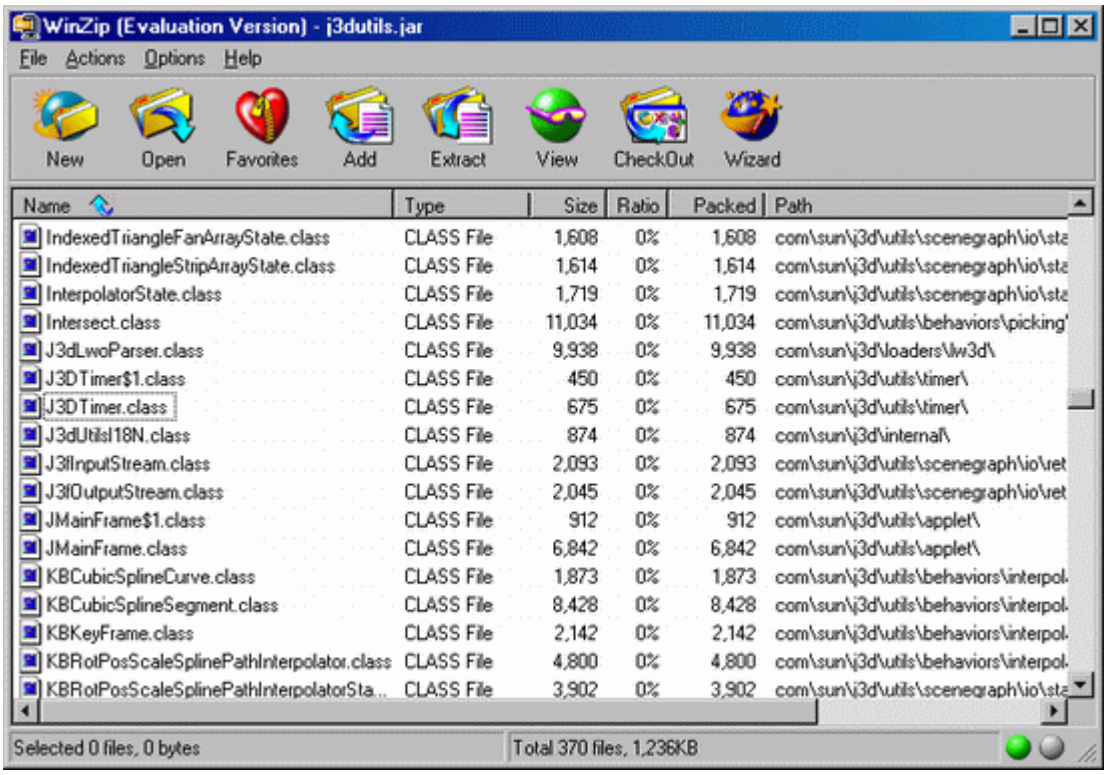

Figure 1. A WinZip View of j3dutils.jar

The J3DTimer class (and its inner class) account for about 1K out of the 1.2MB JAR!.

A look at the decompiled J3DTimer class, using software such as the DJ Java decompiler (http://members.fortunecity.com/neshkov/dj.html), shows that it's a very thin layer of Java over calls to J3DUtils.dll (see Figure 2).

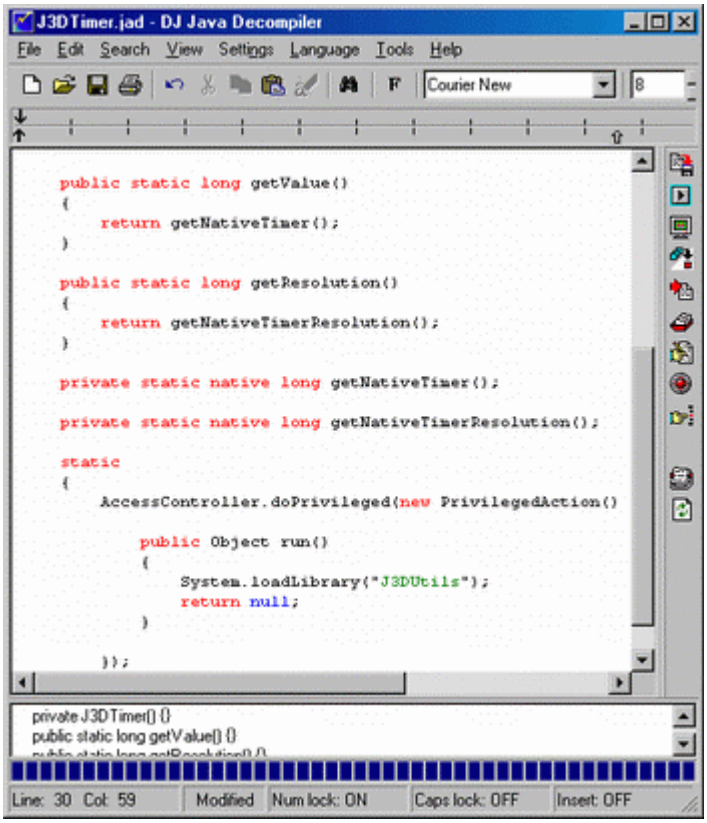

Figure 2. The DJ Java Decompiler View of J3DTimer.

For example, the Java method getValue() calls the J3DUtils.dll function getNativeTimer().

A decompiler isn't really needed at this stage, since the source code for the Java 3D classes is available for download.

J3DUtils.dll can be examined with HT Editor, a file editor/viewer/analyzer for Windows executables, available from http://hte.sourceforge.net/index.html (Figure 3).

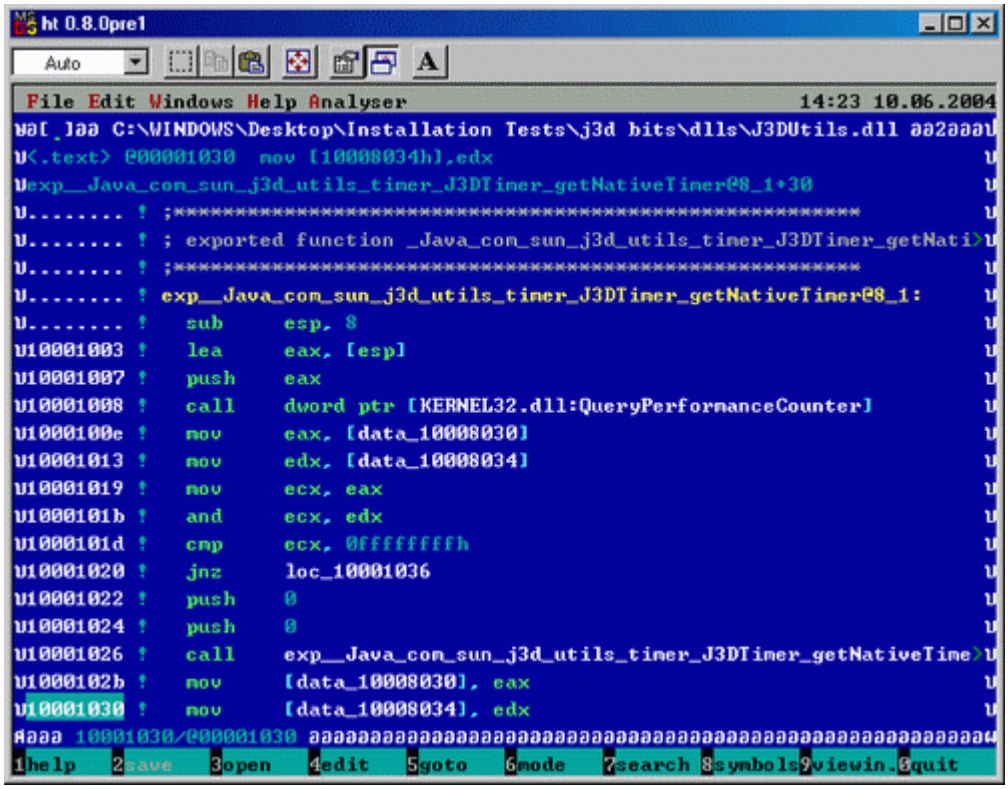

Figure 3. The HT Editor View of J3DUtils.dll

getNativeTimer() uses the Windows kernel32.dll functions QueryPerformanceCounter() and QueryPerformanceFrequency().

In summary, the calls to J3DTimer requires j3dutils.jar and J3DUtils.dll.

It may be worthwhile splitting off the timer code from j3dutils.jar into its own JAR, thereby saving about 1.2 Mb. The technique involves un-JARing j3dutils.jar using WinZip or similar. The result is two folders: com/ and meta-inf/. com/ holds the various classes in j3dutils, which can be deleted aside from the two timer classes in com/sun/j3d/utils/timer/. Meta-inf/ holds a manifest file, Manifest.mf, which should be pulled out of the directory and used in the reJARing:

```
jar cvmf Manifest.mf j3dutils.jar com
```

The size of the slimmed down j3dutils.jar is 2K.

The real drawback with this technique may be a legal one, since a Sun-created JAR is being dismembered. The status of Java 3D is about to change: at *Java One* at the end of this month (June 2004), Java 3D will be made open source, with some provisos, which may make it okay to release modifications.

In the rest of this appendix, we'll use the full version of j3dutils.jar.

2. The BugRunner Application

The BugRunner code is unchanged from the example in chapter 6, aside from the addition of one new method in the BugRunner class, which is explained below.

2.1. Preparing the JARs

Java 3D is not installed on the test machine, instead j3dutils.jar and J3DUtils.dll are placed in the BugRunner directory (as shown in figure 4).

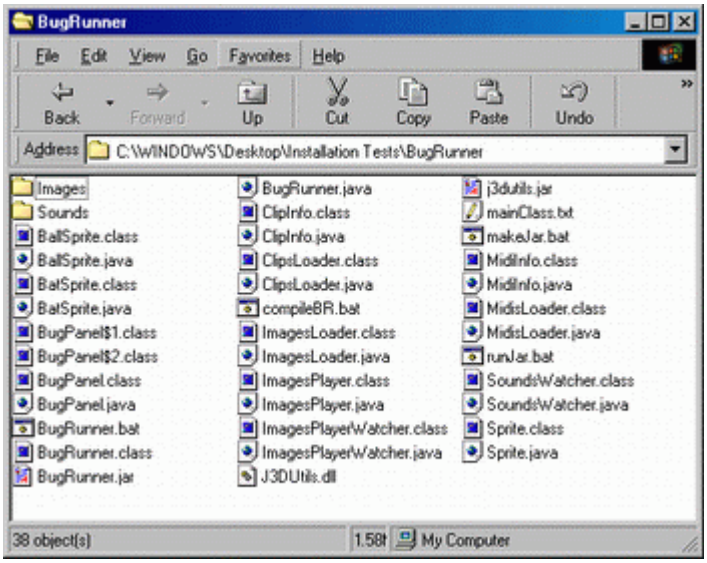

Figure 4. The BugRunner Directory.

Since Java 3D isn't installed in a standard location checked by javac and java, the calls to the compiler and JVM must include additional classpath information.

The compileBR.bat batch file contains:

```
javac -classpath "%CLASSPATH%;j3dutils.jar" *.java
```

The BugRunner.bat batch file contains:

```
java -cp "%CLASSPATH%;j3dutils.jar" BugRunner
```

There's no need to mention J3DUtils.dll, which will be found by the JAR so long as it's in the same directory.

Once the program has been fully tested, the classes and all other application resources must be packaged up as JARs prior to being passed to install4j.

The BugRunner application consists of various classes, and the two subdirectories Images/ and Sounds/. These should be thrown together into a single BugRunner.jar file, *along with any DLLs*. The makeJar.bat batch file contains the line:

```
jar cvmf mainClass.txt BugRunner.jar *.class *.dll Images Sounds
```

The manifest details in mainClass.txt are:

```
Main-Class: BugRunner
Class-Path: j3dutils.jar
```

The manifest specifies the class location of main(), and adds j3dutils.jar to the classpath used when the BugRunner.jar is executed. We assume it's in the same directory as BugRunner.jar.

The DLL is stored in the JAR because the installation is easier if install4j only has to deal with JARs. However, after the installation .exe file has been downloaded to the user's machine, the DLL must be removed from BugRunner.jar, and written to the new BugRunner directory.

This copying from the JAR to the local directory is achieved by the BugRunner class. main() in BugRunner calls a new installDLL() method.

```
public static void main(String args[])
{
  // DLLs used by Java 3D J3DTimer extension
  installDLL("J3DUtils.dll");

  long period = (long) 1000.0/DEFAULT_FPS;
  new BugRunner(period*1000000L);    // ms --> nanosecs
}


private static void installDLL(String dllFnm)
/* Installation of the DLL to the local directory
   from the JAR file containing BugRunner. */
{
  File f = new File(dllFnm);
  if (f.exists())
    System.out.println(dllFnm + " already installed");
  else {
    System.out.println("Installing " + dllFnm);
    // access the DLL inside this JAR
    InputStream in = ClassLoader.getSystemResourceAsStream(dllFnm);
    if (in == null) {
      System.out.println(dllFnm + " not found");
      System.exit(1);
    }
    try {   // write the DLL to a file
      FileOutputStream out = new FileOutputStream(dllFnm);

      // allocate a buffer for reading entry data.
      byte[] buffer = new byte[1024];
      int bytesRead;
      while ((bytesRead = in.read(buffer)) != -1)
        out.write(buffer, 0, bytesRead);

      in.close();
      out.flush();
      out.close();
    }
    catch (IOException e)
    { System.out.println("Problems installing " + dllFnm); }
  }
}  // end of installDLL()
```

installDLL() will be called every time that BugRunner is executed, and so installDLL () first checks whether the DLL is already present in the local directory. If not, it is

installed by being written out to a file. The stream from the DLL file inside the JAR is created with:

```
InputStream in = ClassLoader.getSystemResourceAsStream(dllFnm);
```

This works since the DLL is in the same JAR as BugRunner, and so the class loader for BugRunner can find it. Figure 5 illustrates the technique.

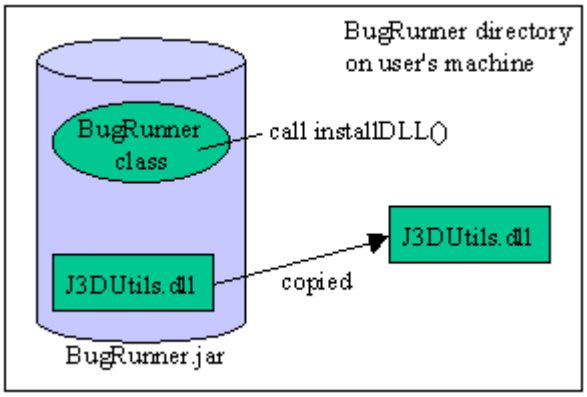

Figure 5. Installing a DLL Locally.

The result is that the BugRunner application installed on the user's machine will consist of two JARs, BugRunner.jar and j3dutils.jar. After the first execution of BugRunner, they will be joined by J3DUtils.dll.

The best testing environment is to move the two JARs to a different machine, one that doesn't have Java 3D installed (but J2SE or JRE must be present). Double click on BugRunner.jar, and the game should begin. The J3DUtils.dll will magically appear in the same directory.

Alternatively, the application can be tested with:

```
java -jar BugRunner.jar
```

Executing this in a command window, will allow stdout and stderr messages to be seen on screen, which can be useful for testing and debugging.

We could stop at this point, since the game is nicely wrapped up inside two JARs. However, an installer will allow a professional veneer to be added, including installation screens, splash windows, desktop and menu icons, and an uninstaller. These are essential elements for most users.

2.2. Creating the BugRunner Installer

This is not a book about install4j, so I'll only consider the more important points in creating the BugRunner installer. install4j has an extensive help facility, and links to a tutorial at its Web site (http://www.ej-technologies.com/products/install4j/overview.html). A good way of understanding things is to browse through the various screens of the BugRunner installation script, bugRun.install4j.

A crucial step is to define the *distribution tree*, the placement of files and directories in the BugRunner directory created on the user's machine at install time. Our approach is to include a Executables/ subdirectory to hold the two JAR files. The directory structure is shown in Figure 6.

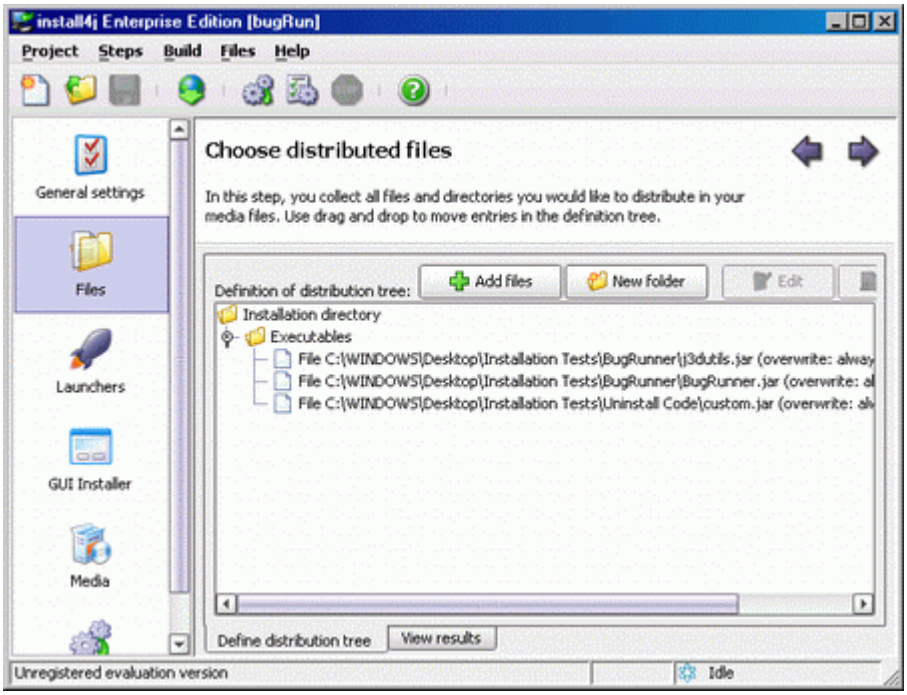

Figure 6. Distribution Tree for the BugRunner Installer.

A moments examination of Figure 6 will reveal *three* JARs inside Executables: BugRunner.jar, j3dutils.jar, *and* custom.jar. custom.jar contains Java code deployed by the uninstaller, which is explained below.

BugRunner.jar is not called directly, but via a .exe file, which is set up on the "Configure executable" screen during the "Launchers" stage (Figure 7).

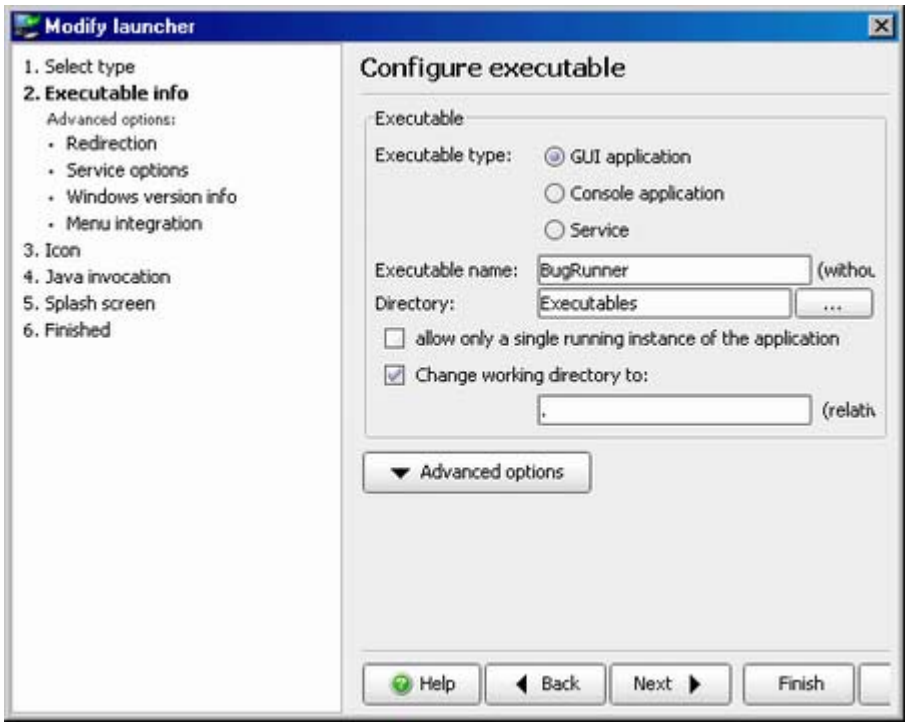

Figure 7. Configuring the BugRunner Executable.

The executable's name is BugRunner.exe, and will be placed in the Executables/ directory along with the JARs.

It is quite important to set the working directory to be ".", so that the JAR will search for resource in the correct place.

Under the "Advanced Options" button it is possible to redirect stdout and stderr into log files. This is generally a good idea for testing and debugging.

The .exe file must be told which JAR file holds the main() method for the application, and which JARs are involved in the application's execution. This is done through the "Configure Java Invocation" screen shown in Figure 8.

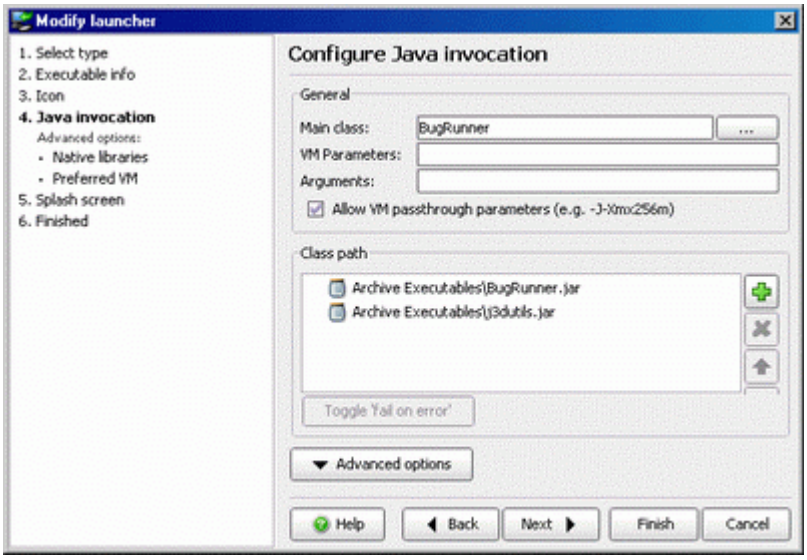

Figure 8. Configuring the Java Invocation.

Figure 8 shows that custom.jar (holding the uninstaller code) is not part of the application, since it isn't included in the "Class path" list.

The "GUI Installer" stage shown in Figure 9 allows the customization of various stages of the installation, such as the welcome screen, tasks to be done before installation, tasks after installation, and the finishing phase. The screen on the right of Figure 9 is for the installer actions which, rather confusing, also contain uninstallation actions.

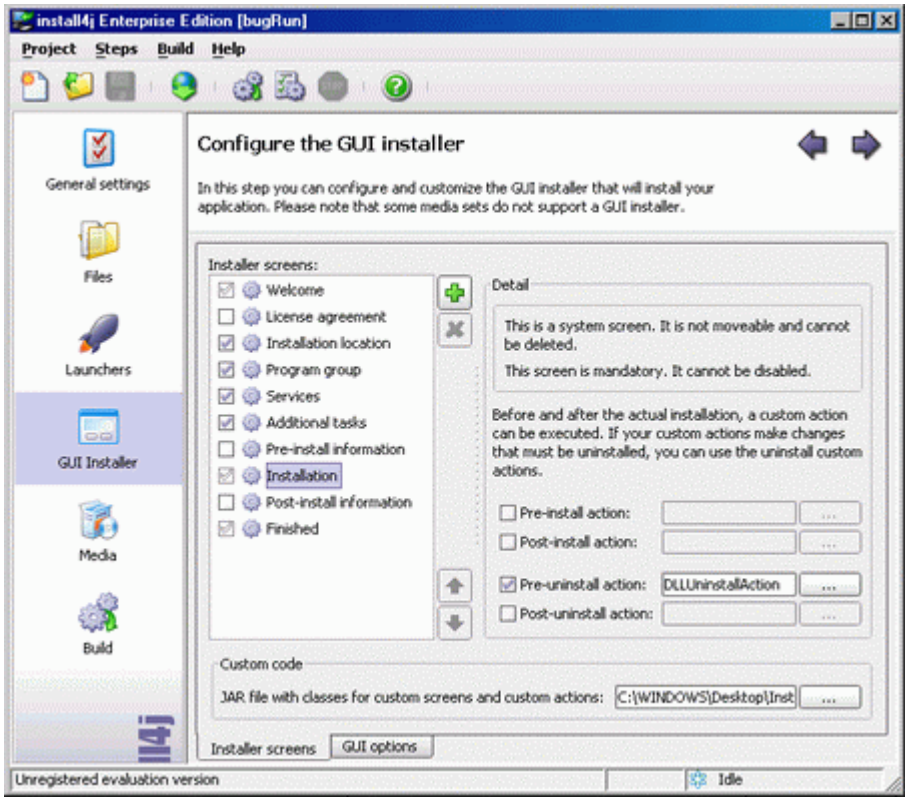

Figure 9. The Installer/Uninstaller Actions for the GUI Installer.

Perhaps the best advantage of using install4j is its close links to Java, most evident in the way that the installation (and uninstallation) process can be customized. install4j offers a Java API for implementing many tasks, and the install4j distribution comes with an example installer that uses the API's capabilities.

The pre-uninstall action is to call the DLLUninstallAction class in custom.jar

2.3. Uninstallation

The default install4j uninstaller will happily delete the JARs that it added to the Executables/ directory, and all the other files and directories it created at installation time. However, the installer didn't add J3DUtils.dll to Executables/; that task was carried out by the BugRunner class when it first ran.

install4j knows nothing about J3DUtils.dll, and so will not delete it, or the sub-directory which holds it. The outcome is that a basic BugRunner uninstaller will not remove the BugRunner/Executables/ directory or the J3DUtils.dll file inside it.

The solution is to define a pre-uninstall operation that removes the DLL, so the main uninstaller is able to delete everything else.

custom.jar contains the DLLUninstallAction class, which extends install4j's UninstallAction class. UninstallAction offers two abstract methods:

```
public abstract boolean performAction(Context context,
                                      ProgressInterface pi);
public abstract int getPercentOfTotalInstallation();
```

The uninstallation task (i.e. deleting the DLL) should be placed inside performAction (). User messages can be sent to the uninstallation screen via the ProgressInterface object. performAction() should return true if the task is successful, false to abort the uninstallation process.

getPercentOfTotalInstallation() sets the amount of the progress slider assigned to the task. The number should be between 0 and 100.

performAction() obtains a list of the DLLs in the Executables/ directory, then deletes each one. This approach is more flexible than just hardwiring "J3DUtils.dll" into the code, and means that the same DLLUninstallAction class can be employed with the Checkers3D installer.

```
private static final String PATH = "../Executables";
     // location of the DLLs relative to <PROG_DIR>/.install4j


public boolean performAction(Context context,
                             ProgressInterface progReport)
// called by install4j to do uninstallation tasks
{
  File delDir = new File(PATH);

  FilenameFilter dllFilter = new FilenameFilter() {
    public boolean accept(File dir, String name)
    { return name.endsWith("dll");  }
  };

  String[] fNms = delDir.list(dllFilter);  // list of dll filenames
  if (fNms.length == 0)
    System.out.println("Uninstallation: No DLLs found");
  else
    deleteDLLs(fNms, progReport);
  return true;
} // end of performAction()
```


The tricky aspect of the code is the use of PATH. The uninstaller is executed in the .install4j/ directory which is at the same level as Executables/. Both of them are located in the BugRunner/ directory installed on the user's machine (see Figure 10.)

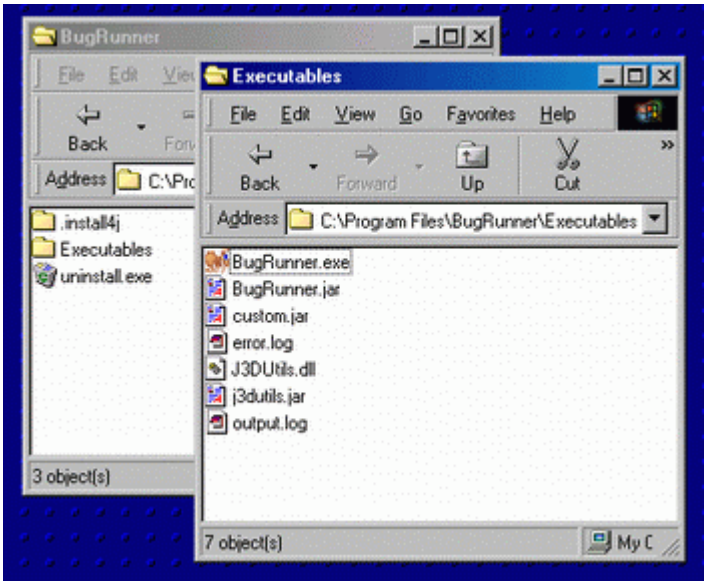

Figure 10. The Installed BugRunner Directories.

The PATH string redirects the File object to refer to Executables/. A list of filenames ending in ".dll" is collected by using a FileFilter anonymous class, and the list is passed to deleteDLLs().

```
private void deleteDLLs(String[] fNms,ProgressInterface progReport)
// delete each DLL file, and report the progress
{
  progReport.setStatusMessage("Deleting installed DLLs");

  int numFiles = fNms.length;
  String msg;
  for (int i=0; i < numFiles; i++) {
    msg = new String("" + (i+1) + "/" + numFiles + ": " +
                                          fNms[i] + "... ");
    deleteFile(fNms[i], progReport, msg);
    progReport.setPercentCompleted( ((i+1)*100)/numFiles );
    try {
      Thread.sleep(500);   // 0.5 sec to see something
    }
    catch (InterruptedException e) {}
  }
}
```

deleteDLLs() loops through the filenames, calling deleteFile() for each one. The ProgressInterface object informs the user of the progress of the deletions. deleteFile() creates a File object for the named file, then calls delete().

The DLLUninstallAction class must be compiled with the install4j API classes added to the classpath:


```
javac -classpath "%CLASSPATH%;d:\install4j\resource\i4jruntime.jar"
                                                DLLUninstallAction.java
```

The creation of the JAR file is standard:

```
jar cvf custom.jar *.class
```

2.4. The BugRunner Installer

The resulting installer, called BR_1.0.exe, takes a few seconds to generate, and is about 1.4 Mb large; the size drops to 950 Kb if the timer-specific version of j3dutils.jar is employed. A version bundled with JRE 1.4.2 comes in at 12 Mb.

3. Checkers3D

The Checkers3D code is unchanged from the example in chapter 8, aside from the addition of the installDLL() method in the Checkers3D class.

3.1. Preparing the JARs

Java 3D is not installed on the test machine, instead all of its JARs and DLLs (seven files) are copied to the Checkers3D directory (see Figure 11).

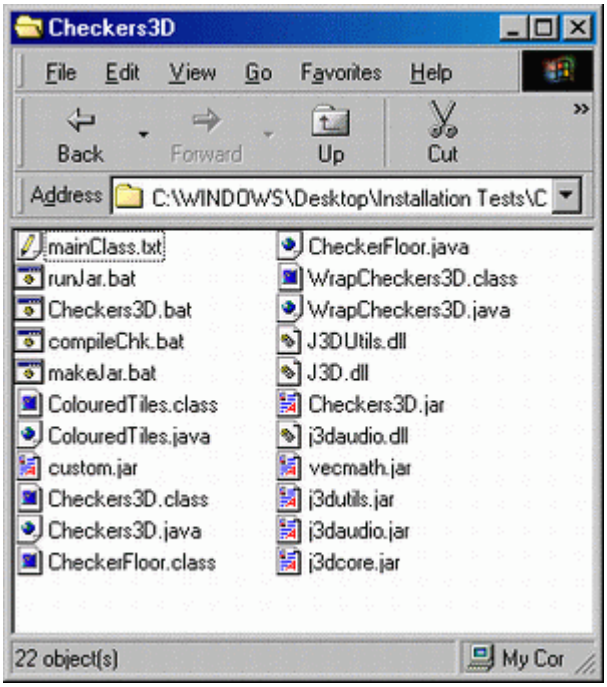

Figure 11. The Checkers3D Application Directory.

Since Java 3D isn't installed in the a standard location, the calls to the compiler and JVM must include additional classpath information.

```
javac -classpath "%CLASSPATH%;vecmath.jar;j3daudio.jar;
                              j3dcore.jar;j3dutils.jar" *.java
```



```
java -cp "%CLASSPATH%;vecmath.jar;j3daudio.jar;
                        j3dcore.jar;j3dutils.jar" Checkers3D
```

There's no need to mention the three DLLs (J3D.dll, j3daudio.dll, and J3DUtils.dll), which will be found by the JARs so long as they're in the same directory.

The Checkers3D classes should be collected into a single Checkers3D.jar file, *along with all the DLLs*:

```
jar cvmf mainClass.txt Checkers3D.jar *.class *.dll
```

The manifest information in mainClass.txt is:

```
Main-Class: Checkers3D
Class-Path: vecmath.jar j3daudio.jar j3dcore.jar j3dutils.jar
```

The manifest specifies the class location of main(), and adds all the Java 3D JARs to the classpath used by Checkers3D.jar.

Checkers3D contains the same installDLL() method as found in BugRunner, but calls it three times.

```
public static void main(String[] args)
{
  // DLLs used by Java 3D extensions
  installDLL("J3D.dll");
  installDLL("j3daudio.dll");
  installDLL("J3DUtils.dll");
  new Checkers3D();
}
```

The Checkers3D application installed on the user's machine will consist of five JARs, Checkers3D.jar, j3dcore.jar, j3daudio.jar, vecmath.jar, and j3dutils.jar. After the first execution, they'll be joined by the three DLLs, J3D.dll, j3daudio.dll, and J3DUtils.dll.

3.2. Creating the Checkers3D Installer

The distribution tree for the installer has the same shape as the one for BugRunner: a Executables/ subdirectory holds the JARs. The directory structure is shown in Figure 12.

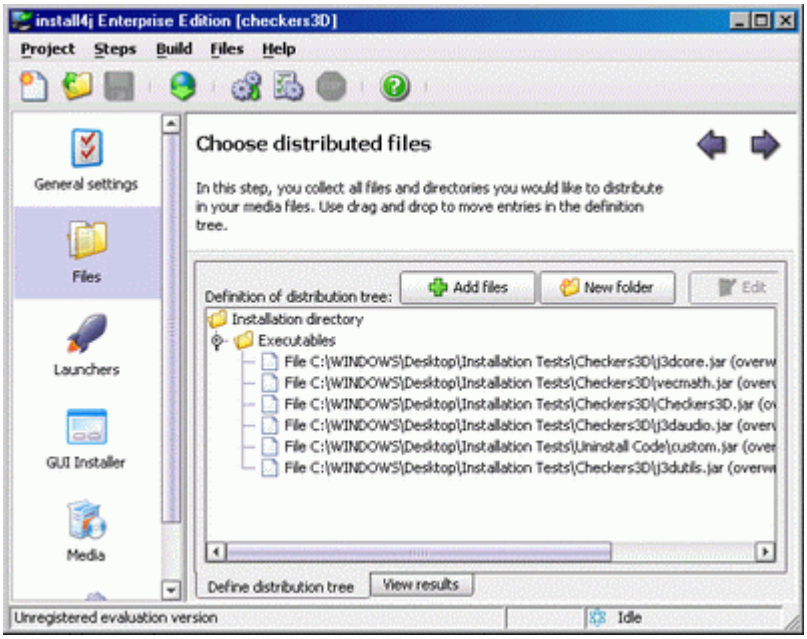

Figure 12. Distribution Tree for the Checkers3D Installer.

The five JARs required by the application are there, and custom.jar for uninstallation. It is the same one as used in BugRunner, no changes are necessary.

Most of the other installer configuration tasks are quite similar to those carried out for BugRunner, such as configuring the executable and Java invocation, and the definition of the pre-uninstall action using the DLLUninstallAction class in custom.jar.

3.3. The Checkers3D Installer

The resulting installer, called C3D_1.0.exe, takes around 8 seconds to generate, and is about 3.6 Mb large. A version bundled with JRE 1.4.2 comes in at 14.3 Mb.