Journal of Modern Technology and Engineering
Vol.3, No.3, 2018, pp.197-204

Jomard Publishing

PROJECT INITIUM AND THE MINIMAL CONFIGURATION PROBLEM

Douglas A. Lyon*

Fairfield University, Fairfield, USA

Abstract. We are given a fully-qualified class name that contains a *main* method for launching an application, in addition to a directory containing all the source files and a list of Jar files used in the larger project. We seek to find, isolate and minimize a set of Java source and Jar files that are necessary and sufficient for compilation, execution and dissemination of a Java system. Large projects are hard to compile and take a lot of computer and human resources. The minimal dissemination system is important for teaching, packaging and deployment. This paper describes a new feature of Project Initium. Initium is a Latin word that means: *at the start.*

Keywords: Java Deployment, Source Code Minimization, Packaging, Java Webstart, ClassPath Minimization, Dependency Analysis.
Corresponding author: Douglas, Lyon, Fairfield University, ECSE Department, 1074 N. Benson Rd, Fairfield, USA, Phone: 203-254-4000x3155, e-mail: *dlyon@fairfield.edu*
Received: 02 June 2018; Accepted: 12 September 2018; Published: 07 December 2018.

1 Introduction

Given a fully-qualified class name that contains a *main* method, a collection of Java source files and Jar files, organized in a directory tree structure, we seek to find a minimal file set that is both necessary and sufficient for compilation. We are subject to the constraint that the run-time requires no additional resources (including data or libraries) and that we can run the program without introducing errors. We call this the *Minimal Configuration Problem* and consider it the next logical step in what we have termed; Project Initium. Initium is a Latin word that means: *at the start.* Project Initium is a long and on-going project that deals with the deployment of Java programs into a properly configured environment.

2 Motivation

We seek a minimal configuration deployment of a minimal set of Java source files, Jar files and data files for the purpose of creating small self-contained programs. Most programs that work on one machine and fail to port easily to another machine seem to fail because of the run-time environment (i.e., the configuration is not correct). The minimization of source and Jar resources encourages encapsulation of inter-object associations, makes explicit dependencies on resources, speeds compilation, reduces maintenance costs and encourages creation of facade design patterns. From an efficiency point-of-view, the start-up time is reduced as the class path is minimized and thus easing the work of the class loader. One aspect of teaching Java includes dissemination of source code to a heterogeneous group of students, many of whom get lost when attempting to compile large projects. By giving them simpler systems to compile, they can focus their attention on a smaller subset of code with a cleaner architecture.

3 Historic review

In the past, project Initium has addressed several deployment problems. For example, there was the problem of resigning already signed Jar files (Lyon, 2008), the problem of deploying a screen saver for CPU scavenging, with mac, windows and Unix screen savers (Lyon & Castellanos, 2007), (Lyon & Castellanos, 2006a), (Lyon & Castellanos, 2006b), (Lyon & Castellanos, 2006c).

We have yet to find anything in the literature that addresses the minimal configuration problem, though there are class-file minimization tools (Lyon, 2004).

4 The static dependency analysis approach

Static dependency analysis makes use of compiler output by forensic examination of the byte codes in Java class files. The Byte Code Engineering Library (BCEL) enables the identification of imports and packages that are present and referred to by a class that contains a main method (Whaley et al., 2002). This class is selected using the Initium interface, as shown in Figure 1.

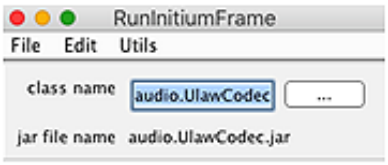

Figure 1: The Initium Interface

The interface allows us to select a class name using an open file dialog box to navigate to the class file that contains the main method for the application. The BCEL allows us to obtain the class name and this is populated in the text field of the graphic user interface. Recursive exploration of the inter-class associations by the BCEL enables the packing of all the classes in a single Jar file. The minimized Jar file contains a manifest and it is executable by a Java runtime.

The minimal source file set is produced in a compressed file called *src.jar* by comparing the source code tree with the classes in the executable Jar file. Code entities that refer to inner reference types are excluded from the *src.jar* file as these are contained in the outer class files source code.

The minimal Jar library set is produced via a class path analysis using a customized version of the JDeps tool (Sharan, 2014). In this case, we include only those jars that are referred to by the class files for which no source code exists. JDeps customization enabled its' integration with the Initium system, as shown in Figure 2.

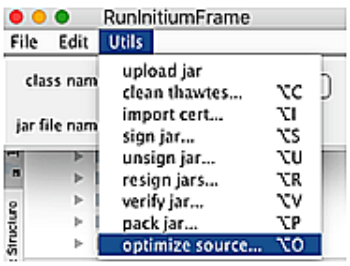

Figure 2: The Optimize Source Menu Item

The final step is to emit a *build.xml* ant file that enables the compilation. The Jar file set is minimized based on static dependency analysis.

5 Results and problems with static dependency analysis

For some cases, the system produces good results. For example, in one case, upon completion, we obtain three files, *build.xml, jars.jar* and *src.jar*. For a simple audio playback application, the src.jar contained 4 Java source files and the *jars*.jar file contained two Jar files. The original project had 8,346 Java files and 219 jar files. This represents a 2,089 times reduction in source files and a 100 times reduction in the number of jar files. The build is clean and we find that the outcome is typically positive (faster compilation, ease of maintenance, faster start-up times, etc.).

Static dependency analysis can trigger issues of improper configuration that can yield a compile failure. Consider the following code:

```
   File f = new File(
       "/Users/lyon/current/java/data/audio/bong.au");
System.out.println(f); UlawCodec ulc = new UlawCodec(f); ulc.play();
```

Static dependency analysis fails to identify dependency on the audio file "*bong.au*". Data files located via an absolute path name pose tricky configuration issues during deployment. Code data dependencies are a fruitful source of dissemination bugs and various run-time failures (i.e., *FileNotFoundException*). As this is a run-time failure, only dynamic analysis can detect the problem.

Another use-case for failure includes dynamic class loading. Consider:

```
Class.forName("org.sqlite.JDBC");
```

The dependency on the JDBC driver cannot be detected at compile time by any normal compilation process and thus a *ClassNotFoundException* will be thrown at run-time.
Other use-cases for failure include reflection based coding that makes use of the annotation API, as described in (Lyon, 2010). For example:

```
Annotation a[] = c.getAnnotations();
```

6 Dynamic dependency analysis

The afore mentioned configuration issues demonstrate that static dependency analysis is impoverished when faced with dynamically loaded classes or resources. During the deployment phase, missing resources will typically halt with an exception (i.e., *ClassNotFoundException*, or some variant thereof). Using a Java Agent (Rodriguez-Prieto et al., 2018) we were able to obtain a list of classes that are needed at run-time. The Java Agent instruments the JVM and makes a list of what it is loading. The *Instrumentation* interface enables a listing of all loaded classes

One of the frameworks that we relied upon is the EA Agent Loader (EA). The Agent Loader enables the writing and testing of Java agents that use dynamic agent loading without making use of the –java agent jvm parameter (and thus attaching to a live JVM). The basic idea is that a class with the signature of:

```
public class HelloAgentWorld{
    public static class HelloAgent {
        public static void agentmain(
        String agentArgs, Instrumentation inst)
```

can be invoked with:

```
public static void main(String[] args){
       AgentLoader.loadAgentClass(
       HelloAgent.class.getName(), "Hello!");
}
```

The *loadAgentClass* method loads the Java agent dynamically, thus dealing with the problem of finding the proper JVM class. In practice, we optimize our project (including the Jar file set) using the following code:

```
final String mainClassName = "audio.UlawCodec"; final String
sourceRootDirectory
       = "/Users/lyon/current/java/j4pCode/src";
final String sourceOutputDirectory
       = "/Users/lyon/attachments/ulawProject/src/";
final String classFileDirectory
       = "/Users/lyon/current/java/j4pCode/out/production" +
         "/j4p";
final String jarRootDirectory
       = "/Users/lyon/current/java/j4pCode/jars/";
final String jarOutputDirectory
       = "/Users/lyon/attachments/ulawProject/jars/";


AgentLoader.loadAgentClass(ddaa.class.getName(), "");

File srdf = new File(sourceRootDirectory); File sodf = new
File(sourceOutputDirectory); File jodf = new
File(jarOutputDirectory); final List<URL> urls = DynamicAnalyzerUtil
       .populateLibrariesAndSourceFiles(
              new File(jarRootDirectory), new File
                     (classFileDirectory));

final DynamicDependencyAnalyzer ddaa
       = new DynamicDependencyAnalyzer(mainClassName,
                                  srdf, sodf, jodf,
                                  urls);
```

An instrument is passed the *classLoaderContext* by the run-time:

```
public class DDAInstrument {
   private static Instrumentation instrumentation=null;

      public static void agentmain(final String agentArgument,
                           final Instrumentation inst) {
       instrumentation = inst;
   }

   public static Instrumentation getInstrumentation(Class cls)

         throws NoSuchMethodException, InvocationTargetException,
               IllegalAccessException {
       final Method meth = cls.getMethod("main", String[].class);
       final String[] params = null;
       meth.invoke(null, (Object) params);
       return instrumentation;
   }
}
```

The *DDA Instrument* instance uses the current class loader context to retrieve a list of all the loaded classes and Jars, if and only if these things are loaded when the main is executed:

```
public File[] analyzeDynamicDependency(boolean hasSourceFile)
       throws IOException, URISyntaxException,
             InvocationTargetException, IllegalAccessException,
             NoSuchMethodException, ClassNotFoundException {
    ClassLoader cl= Thread.currentThread().getContextClassLoader();

    final Instrumentation inst = DDAInstrument
           .getInstrumentation(cl.loadClass(mainClassName));
    Class[] initiatedClasses = inst.getInitiatedClasses(cl);
    return processLoadedClasses(Arrays.asList(initiatedClasses),
                               hasSourceFile);

}
```

The *processLoadedClasses* contains the details of copying the needed source code and Jar libraries into the target directories:

```
private File[] processLoadedClasses(final List<Class<?>> classes,
                                    boolean sourceFile)
       throws IOException {
    // We don't need duplicate files especially jar files
    final Set<File> usedSourceFiles = new HashSet<>();
    final Set<File> usedJarFiles = new HashSet<>();

    for (Class<?> clazz : classes) {
        if (clazz.getName().contains("$")) continue;
        final ProtectionDomain
               protectionDomain =
               clazz.getProtectionDomain();
        final CodeSource
               codeSource =
               protectionDomain.getCodeSource();
        if (codeSource != null) {
            final URL jarFileUrl = codeSource.getLocation();
            if (jarFileUrl.getFile().endsWith(".jar")) {
                File file = new File(jarFileUrl.getFile());
                String outputFile = String.format("%s%s%s",
                                                  jarOutputDirectory,
                                                  File.separator,
                                                  file.getName()
                                                 );

                copyFiles(file, new File(outputFile));
                usedJarFiles.add(file);
            } else {
                String canonicalName = clazz.getCanonicalName();
                if (canonicalName == null) continue;
                String filePathUrl = mapClassToSource(
                       canonicalName);

                String replace = canonicalName.replace(".",
                                                       File.separator
                                                      );
                String outputFile = String.format("%s%s%s%s",
```

```
                                                sourceOutputDirectory,
                                                File.separator,
                                                replace, ".java"
                                               );

                copyFiles(new File(filePathUrl),
                          new File(outputFile)
                         );
                usedSourceFiles.add(new File(filePathUrl));
            }
        }

    }

    return sourceFile ? usedSourceFiles.toArray(new File[0]) :
            usedJarFiles.toArray(new File[0]);
}
```

Inner classes have names that contain a "$" and these are filtered out from our source file set, as the inner classes are contained in the source files that hold the outer classes.

7 Problems with dynamic dependency analysis

We are given a main method that dynamically loads data and classes by making use of the *Class.forName* and an external file reference:

```
public static void main(String[] args)
        throws ClassNotFoundException {
    Class.forName("org.sqlite.JDBC");
    File f = new File(
            "/Users/lyon/current/java/data/audio/bong.au");
    System.out.println(f);
    UlawCodec ulc = new UlawCodec(f);
    ulc.play();
}
```

Our dynamic dependency analysis produced output that included:

```
/jars/sqlite-jdbc-3.16.1.jar
```

This indicated that we have addressed one of the major points of static dependency analysis; given an execution of all paths that dynamically load external classes, we can be reasonably assured that we will have the libraries that we need to run the code. The program has minimized the source file set:

```
./src/audio/SimpleAudioPlayer.java
./src/audio/UlawCodec.java
```

Both programs are critical to execution. However, the downside is we have still not addressed the data that we need to execute. The file:

```
"/Users/lyon/current/java/data/audio/bong.au"
```

will be missed by the dynamic examination of the *classloader*. There are several approaches to solve such a problem. For example, we could bundle the resource into the source code (Lyon,

2005). Such an approach UUEncodes the data into the source file. This approach has the advantage of making sure the data needed by the object is geographically co-located in the byte code and is unlikely to be lost. However, the approach is invasive, increases compilation time, increases byte code size and, worse, may not scale well to large data since compilers often crash when presented with source code files that are too large.

Another approach is to use a URL to make reference to the data file. This has the advantage of enabling the code to work anywhere. However, the drawback is that such a change is invasive (requiring a code change) and requires that the Internet be available. Moreover, we may not always want to put our data on the Internet, where the world can see it. Another way we can load our resource is through the class loader:

```
url = classLoader.getResource(resource);
```

Such an approach results in an invasive change to the code, to wit:

```
ClassLoader classLoader =
       Thread.currentThread().getContextClassLoader();
URL url = classLoader.getResource("resources/bong.au"); File f = new
File(url.toURI()); System.out.println(f); UlawCodec ulc = new
UlawCodec(f);
```

Even worse, the instrument class loader keeps tracks of classes that it is loading, but it is not keeping track of resources that it is loading.

Another solution is to employ a *ResourceManager* able to transfer over compressed data when it is out of data, if the Internet is available, and otherwise checks the local disk for the data. This too is both invasive and requires an Internet connection, when the resources are not locally available.

8 Conclusion

Static dependency analysis provides a fast solution to creating a minimal set of Jar libraries and Java source code, however, it fails for the use-case of dynamically loaded resources and classes. Dynamic dependency analysis is able to detect the run-time loading of classes and provides a good intermediate solution. However, it requires complete exploration of all execution paths. Additionally, dynamic dependency analysis fails to find data resources with absolute pathnames. Adding a class loader to Instrumentation in Java is unable to detect dynamically loaded resources without some sort of a custom class loader.

The use of a *ResourceManager* to handle data may be a very good way to approach the problem. The question of how this should be implemented, exactly, remains open. Ad-hoc retrieval of individual files does not seem like sound software engineering, as testing may fail to detect a missing resource.

In summary, static dependency analysis will miss reflection-loaded classes and data referred to via local disk store. Instrumentation will detect reflection-loaded classes, but it too will miss data files stored on local disk and referred to via absolute path names. Instrumentation will also miss relative class loader references, as in:

```
ClassLoader classLoader =
       Thread.currentThread().getContextClassLoader();
URL url = classLoader.getResource("resources/bong.au"); File f = new
File(url.toURI()); System.out.println(f); UlawCodec ulc = new
UlawCodec(f);
```

The question of how to best detect the use of file-based resources at run-time remains a topic of current research. Possible candidate approaches include a custom class loader or a resource manager.

References

Sharan, K. (2014). *Beginning java 8 fundamentals language syntax, arrays, data types, objects, and regular expressions*, Berkeley: CA Apress. Retrieved from http://dx.doi.org.libdb.fairfield.edu/10.1007/978-1-4302-6653-2

'EA Agent Loader' by Electronic Arts. https://github.com/electronicarts/ea-agent-loader (last accessed Jun 22, 2018).

Lyon, D.A. (2010). Semantic Annotation for Java. *Journal of Object Technology*, *9*(3), 19-29. http://www.docjava.com/pub/document/jot/v9n3.pdf

Lyon, D.A. (2008). I Resign! Resigning Jar Files with Initium. *Journal of Object Technology*, *7*(4), 9-27. http://www.docjava.com/pub/document/jot/v7n4.pdf

Lyon, D.A., Castellanos F. (2007). The Saverbeans Screensaver and Initium RJS System Integration: Part 5. *Journal of Object Technology*, *6*(1), 35-57. http://www.docjava.com/pub/document/jot/v6n1.pdf

Lyon, D.A., Castellanos F. (2006a). The Initium RJS Screensaver: Part 4, Automatic Deployment. *Journal of Object Technology*, *5*(8), 31-40. http://www.docjava.com/pub/document/jot/v5n8.pdf

Lyon, D.A., Krepsztul P. & Castellanos F. (2006). Macintosh Screensaver in Java: Part 3. *Journal of Object Technology*, *5*(7), 9-17 http://www.docjava.com/pub/document/jot/v5n7.pdf

Lyon, D.A., Castellanos F. (2006b). The Initium RJS Screensaver: Part 2, UNIX. *Journal of Object Technology*, *5*(5), 7-15 http://www.docjava.com/pub/document/jot/v5n5.pdf

Lyon, D.A., Castellanos F. (2006c). The Initium RJS Screensaver: Part1, MS Windows. *Journal of Object Technology*, *5*(4), 7-16 http://www.docjava.com/pub/document/jot/v5n4.pdf

Lyon, D.A. (2005). Resource Bundling for Distributed Computing. *Journal of Object Technology*, *4*(1), 45-58. http://www.docjava.com/pub/document/jot/resource.pdf

Lyon, D.A. (2004). Project Initium: Programmatic Deployment. *Journal of Object Technology*, *3*(8), 55-69.

Rodriguez-Prieto, O., Ortin, F., O'Shea, D. (2018). Efficient runtime aspect weaving for java applications. *Information and Software Technology*, 100, 73-86. http://www.sciencedirect.com/science/article/pii/S0950584918300521. Accessed Jun 20, 2018.

Whaley, J., Martin, M.C., & Lam, M.S. (2002, July). Automatic extraction of object-oriented component interfaces. In *ACM SIGSOFT Software Engineering Notes*, *27*(4), 218-228.