Java Archive Fingerprinting Application

Jeff Regis
Carleton University
134 Baseline Road
Ottawa, ON
(613) 324-0632
jjcregis@gmail.com

ABSTRACT
In this paper, a content-based approach to gauging the similarity of two Java archives (jar files) is discussed. The jar files are examined for similar functions and methods and any similarities found are reported to the user.

Programming Languages
The application was programmed using Java.

Keywords
Application: The Java archive fingerprinter that this report discusses

Jar file: Java archive

1. INTRODUCTION

The purpose of the application is to be able to examine two jar files and give an indication of how similar they are. This application was developed to satisfy the requirements for completing the term project of COMP 4900/5900 offered by Carleton University in the Fall 2010 semester.

There are many different strategies for examining jar files for similarities. The main two approaches are examining the structure of the archive and examining the contents of the files in the archive. The application takes the content-based approach.

The application examines all Java source files contained in each jar file. The application examines each of these files for function or method definitions that adhere to a predetermined structure. Several properties of functions that are found are stored. When both jar files have been completely examined, the functions found in their respective archives are compared against each other and a percentage value is given based on their similarity.

2. CLASSES

The application uses several classes to perform its function.

It is required that the project implement a few interface classes. IFingerprint is implemented by Fingerprint which stores a string encoding of a jar file. IFingerprintGeneratorDetector is implemented by FingerprintGeneratorDetector which generates a fingerprint from a jar file and generates a comparison result from two jar files. IFingerprintResult is implemented by FingerprintResult which stores the similarity percentage of the two jar files and a textual comment conveying information about the analysis.

The application also utilizes a few custom classes not required by the project requirements. Function stores information about each function found in the source files, including the function's protection, return type, name and the number of arguments it takes. FingerprintProgram contains the main procedure.

A number of classes from the Java SDK are used as well. BufferedReader, InputStream and InputStreamReader are used for reading the contents of a file in an archive and are found in the package java.io (IOException is also included due to being a requirement for some of these classes to work). Jar files are handled using the JarFile class and its contents are represented with JarEntry objects, both from java.util.jar. Finally, three classes from java.util are utilized in the application. ArrayList is used for storing objects that need to be examined at a later time, StringTokenizer is used for analyzing the textual content of the JarEntries and Enumeration is used as a wrapper object for easy iteration over a collection of JarEntries.

3. PROCESSES

3.1 Analysis

3.1.1 Setup

Most of the application logic is located in the FingerprintGeneratorDetector class. All of the logic responsible for analyzing the contents of jar files is located in the method named generateFingerpringFrom(). When invoked, this method will produce a Fingerprint object that stores the fingerprint data for a jar file with the archive's name provided as an argument.

The jar file analysis process begins by opening an existing jar file given its name. The archive is stored as a JarFile object. The contents of the JarFile, JarEntry objects, are wrapped by an Enumeration object into a list of entries.

The list of JarEntries is then examined for Java source files. The application will know that a file is a source file if its filename ends with the substring ".java". Every file in the JarFile that has this extension will be added to a list (using an ArrayList object) of files to be examined further.

A BufferedReader is used for examining the contents of each source file. The BufferedReader is fed an InputStreamReader as an argument which is itself set up by the InputStream obtained from each JarEntry. One at a time, the BufferedReader examines a source file stored in the list. One line of the current source file is examined at a time.

All brackets, curly braces and commas that may exist in each line are spaced into their own tokens. That is, these characters are

replaced by the same character with a space on either side. This is done in order to make processing easier which will be clear shortly. When these characters have been spaced into individual tokens, the current line is added to a large string that is used for processing.

3.1.2 Processing

After the above steps are completed, the large string containing all of the text from all Java source files in the jar file is tokenized and examined for possible function and method definitions. The application will search the string for a function having the following format:

public|private|protected returnType funcName (argType1 argName1 , ...) { … }

The application will consider any sequence of tokens to be a possible function if the token is the string “public”, “private” or “protected”. Upon reading one of these strings, the string will be added to a list to store the details of a Function object's properties if an entire function definition ends up being found.

The next two expected strings in the sequence are the function's return type and the function's name. Unlike the function's protection which was previously detected, the return type may not be a predefined word in Java since it could be a user-defined type. This is especially obvious for the function name since the user has the freedom to name their function whatever they want. The application adds these two details to the list.

Next, the application expects an opening bracket which signifies that there are arguments that are to be read in. Beginning here is where the spacing of brackets, braces and commas becomes important. Had this not been done before, a token of the format “funcName(argType1” might exist. The processing of these tokens is greatly simplified if spaces are used as the only delimiters between the tokens. The types or names of the function's arguments are not taken into account, only how many arguments the function accepts.

After the closing bracket is read, the next token expected is an opening curly brace which signifies the beginning of the function's body. Analyzing the body in a meaningful way would be a difficult endeavor so its contents are disregarded. Once a closing curly brace is found, the application creates a new Function object and stores the function's protection, return type, name and its number of arguments. The object is then added to a list of other Function objects to be encoded later.

3.1.3 Encoding

After all source content has been analyzed and processed, a list of functions found in the jar file will available and will need to be encoded into a string fingerprint. The encoding process is simple: for each function in the list we have been keeping, the function's protection, return type, name and its number of arguments are appended to the string, separated by spaces.

3.2 Comparison

Comparison between two Fingerprint objects is done in the gemerateFingerprintResultFrom() function in the FingerprintGeneratorDetector class. A similarity value is maintained and increased by varying, semi-arbitrary amounts depending on how many of the properties of the functions being compared are the same.

For each fingerprint, its encoding is dismantled and a list of Function objects corresponding to their respective jar files is created. The functions in each list are compared first by return type, then by the number of arguments, then by name, then by protection. If two functions have the same return type, the similarity value increases by 5% since it is unlikely (yet still somewhat likely) that two functions with the same return type will be the same function. If the number of arguments match as well, the similarity rises to 25%. If the names of the functions match too, the similarity becomes 95%. This large jump is due to the fact that little information can be gathered from the return type and number of arguments alone, but if the names match then it is likely that the functions have the same purpose. The similarity becomes 100% if these three properties match in addition to the protection since the protection of a function is a minor detail. During this process, comments regarding the similarity of the two functions being compared are stored in a FingerprintResult so the user may better interpret the results and perform further investigation if they desire.

After all of the functions have been compared, the certainty of the two jar files being the same is calculated with the formula:

similarity / ((numFuncInjar1 + numFuncInjar2) / 2)

This value should be used in conjunction with the comments in order to determine if a certain function could exist in both jar files being examined.

4. RESULTS

The formula used to compare the certainty that the two jar files being compared will yield 100% certainty if a jar file is compared against it self. The more the jar files differ, the smaller the certainty value becomes. This approach of comparing similar functions found within two jar files works relatively well with smaller archives but the certainty value quickly becomes less relevant as the files become larger in size and contain more functions.

The comments produced by the application are a useful aid for the user when determining if the same function is found in the two files. Since comments are only added once functions with the same names are detected, they still remain useful when comparing large jar files as opposed to the certainty value which may become obsolete.