Compression Utility 2.0 Component Specification

Design Rationale

The Java 2 Platform already includes a flexible and extensible compression mechanism. This is exposed through the DeflaterOutputStream and InflaterInputStream classes with accompanying Deflater and Inflater classes.

This design seeks to extend and wrap that framework with the following goals in mind:

- Provide a high level interface for different input object types. (This is the goal explicitly stated in the Requirements Specification.)
- Create the notion of a matched compressor/decompressor "pair".
- Maintain backwards pluggability for the existing compression/decompression (Deflater/Inflater) classes and their descendants.

While the component includes the implementation of a "custom" compression/decompression module, the algorithm is an established one. Many years of research in the Computer Science community have been devoted to compression schemes, and this design aims to define a usable utility based on a tried and tested algorithm.

In order to maintain compatibility with the DeflaterOutputStream and InflaterInputStream, the Deflater and Inflater classes must be extended by custom schemes. While the creation of separate interfaces for encoders/decoders would allow slightly more flexibility, it would complicate matters in the more common expected usages, (1) making future custom schemes incompatible with DeflaterOutputStream and InflaterInputStream, and (2) making wrapper classes necessary (e.g. "public class DeflaterWrapperEncoder implements Encoder") to use the built in classes with the component. Furthermore the Deflater and Inflater class signatures are quite flexible, and allow for the implementation of single-pass stream processing—they do not require the entire input data set before output is produced. Of course, a compression scheme may intentionally cache the entire data set—by returning true for needsInput() until finish() is called—should the need arise (e.g. for constructing a dictionary).

The chosen Codec (an encoder/decoder pair) are specified by class name to a CompressionUtility object at runtime. This follows practices set forth in the Java XML API, another framework which allows pluggable implementation classes (http://xml.apache.org/~edwingo/jaxp-ri-1.2.0-fcs/docs/api/ —see the ParserFactory class for an example.)

Changes in Version 2.0

The major theme of changes for version 2.0 is support for archive files, and in particular, WinZIP-compatible ZIP files making use of the Deflate compression algorithm. ZIP files are a fundamentally different kind of entity from compressed files created by the CompressionUtility of version 1.0 of this component. ZIP files contain such compressed files within their content, but they also contain considerable global and per-file metadata, such as relative pathnames, CRC checksums, and comments. The generally-used version of the file format is in the public domain, but the later enhancements to it, especially by competitors PKWare (the original creator) and WinZip Computing, appear to be proprietary. A technical description of the file format is available at the PKWare web site: http://www.pkware.com/company/standards/appnote/. For the purposes of the Compression Utility 2.0, however, most of the details of the format are unnecessary because the component makes use of the Java platform library's standard support for creating and reading ZIP files.

Given that ZIP archives are such a fundamentally different thing from simple blocks of compressed data, a new collection of abstractions and a new utility class were created for

Compression Utility 2.0 to handle these and similar archives. Parallel to the original CompressionUtility class and its associated Codec interface and related classes, version 2.0 of the component defines a new ArchiveUtility utility class, with Archiver, ArchiveCreator, and ArchiveExtractor interfaces and ZIP-specific implementations. This provides a framework within which support for other archive formats such as tar and rar can be conveniently added in a future version of the component. The design of the ArchiveUtility and the general scheme for that class and its related interfaces are intentionally similar to those of the CompressionUtility class and other version 1.0 classes and interfaces, but the details of the ArchiveCreator, and ArchiveExtractor interfaces were chosen to be appropriate for their roles.

The requirements for version 2.0 also specified that support for the Deflate algorithm be added to the component. This algorithm was already supported by the component through its DefaultCodec class, but the class' name did not make that apparent. As approved in forum discussion during the design phase for this component update, the Deflate requirement is addressed by adding a new DeflateCodec class that does exactly the same thing as the existing DefaultCodec class, and deprecating DefaultCodec. DefaultCodec's name cannot simply be changed because it could break existing code that uses version 1.0 of the Compression Utility.

1. Design

1.1 Design Patterns

The Façade pattern is used for the CompressionUtility and ArchiveUtility classes. The client application does not need to be aware of the workings of the underlying Codec / Archiver or java.util.zip.* classes, though it does provide a class name to the CompressionUtility on construction.

The Codec and Archiver interfaces define Abstract Factories for Deflaters and Inflaters on the one hand, and for ArchiveCreators and ArchiveExtractors on the other. In this implementation, LZ77Codec is a concrete Factory and LZ77Encoder and LZ77Decoder are its Concrete Products; similarly with DeflateCodec and the standard java.util.zip.Deflater and java.util.zip.Inflater. On the archive side, ZipArchiver is a concrete Factory for ZipCreator and ZipExtractor onjects.

Use of a Factory (Codec or Archiver) as a key supporting object for the CompressionUtility and ArchiveUtility classes is similar to applying the Strategy pattern.

Designing the component this way enables us to extend the component in the future in a couple of ways. First, the Codec / Archiver implementation can be replaced with an alternative. DeflateCodec illustrates this as an alternative to LZ77Codec. Second, since the codecs are decoupled from the stream processing (calling read() and write() methods and looping) the CompressionUtility and ArchiveUtility classes could be replaced by alternatives; for instance, in place of CompressionUtility a compression front-end that supports asynchronous processing or non-stream data. The new class would only have to call Codec.createDeflater() and Codec.createInflater() to gain access to the setInput(), and deflate()/inflate() methods.

CompressionUtility also provides a public static method createCodec() (which it uses internally as well). A client application can use this to gain direct access to a Codec instance of a named class.

1.2 Industry Standards

The component utilizes the existing framework for inflating and deflating streams that exists in the java.util.zip package. It extends it by offering a runtime-pluggable mechanism for specifying Deflater/Inflater pairs through the Codec interface. It also

offers a simple façade, and easy access to streams, files, and string buffers as both input and output options.

In version 2.0, the component provides for creating, examining, and extracting standard ZIP archive files, also based on Java's built-in support.

1.3 Required Algorithms

1.3.1 LZ77 Compression

For this implementation of the component, compression and decompression are offered with the LZ77 Lempel-Ziv algorithm. This algorithm is legally unencumbered, and forms the basis for many other implementations (e.g. zlib, gzip) that purposefully choose patent-free algorithms. It should not be confused with the LZW (Lempel-Ziv Welch) algorithm owned by Unisys, which is used in the Compuserve GIF format.

The algorithm is outlined below with respect to the Deflater/Inflater interface methods:

LZ77Encoder, deflate():

- If the input buffer is empty, write nothing and return 0
- Maintain a window of the last 256 bytes read from the input
- Loop (while there are bytes to read and space to write):
    - Look for the longest match between the window and the upcoming characters
    - Write the position and length of the match, (0,0) for no match, (33,4) for a match that starts at position 33 (1-based) and extends 4 bytes
    - Write the next byte from the input
    - Increment the window

LZ77Decoder, inflate():

- If the input buffer is empty, write nothing and return 0
- Maintain a window of the last 256 bytes written to the output
- Loop (while there are bytes to read and space to write):
    - Read 3 bytes. The first two are position and length of a match. If they are (0,0) (this will always be the case at the beginning of an input) do nothing. If they are (22, 5) starting at position 22 in the window, copy 5 bytes to the output.
    - Just output the 3rd byte

More detail can be found at http://www.free2code.net/tutorials/other/LZ77 and many other locations on the Web.

1.3.2 ZIP Archives

Creating ZIP files is done via java.util.zip.ZipOutputStreams. To create a ZIP file, first open an output stream to the ZIP file and wrap it in a ZipOutputStream. For each regular file to add:

- convert its system-dependent relative pathname to a ZIP entry name via an intermediate File object with use of the static ZipCreator.createNameFromFile(File) method;
- construct a java.util.zip.ZipEntry object based on the entry name (nothing else need be done to it);
- pass the ZipEntry to the ZipOutputStream's putNextEntry(ZipEntry) method; and
- open an InputStream on the file to add and copy all the bytes from that stream to the ZipOutputStream until the end of the input is reached.

Although the ZIP format and Java library support it, for the broadest possible compatibility the Compression Utility does not place entries for directories into ZIP files it creates.

Examining and extracting the contents of a ZIP file is done via java.util.zip.ZipFile objects. (Despite its name, ZipFile is not a subclass of java.io.File. It is a class specific to the purpose of reading ZIP archives.) Among its features, ZipFile provides ZipEntry objects describing members of a ZIP archive, either specific to a particular member name (getEntry(String)) or in the form of an Enumeration of all the available ZipEntry objects for a particular ZIP. Listing ZIP contents principally involves extracting Files representing the entry names from the Enumeration (see ZipExtractor.createFileFromName(String)), but entries representing directories are ignored (see ZipEntry.isDirectory()). To extract a specific file, represented by relative File object *entryFile* from a ZIP to an existing target directory designated by a File object *dirFile*:

- obtain an entry name corresponding to entryFile by passing it to ZipCreator.createNameFromFile(File);
- obtain a ZipEntry corresponding to that name from ZipFile.getEntry(String); if that's not null,
- create a File representing the path to the target file via `new File(`*`dirFile`*`, `*`entryFile`*`)`, and obtain also a File for its parent directory (File.getParentFile());
- if the parent directory does not exist, (File.exists()) then create it and all necessary ancestors via File.mkdirs();
- open an OutputStream on the target file;
- open an input stream on the ZIP entry (ZipFile.getInputStream(ZipEntry));
- copy bytes from the input stream to the output stream until the end of the stream is reached.

Other procedural instructions regarding handling and conversion of the data streams are covered in the class and method documentation accompanying the UML diagrams.

1.4 Component Class Overview

ArchiveCreator:

Interface defining the API for objects that can create archive files of specific types.

ArchiveExtractor:

Interface defining the API for objects that can examine and extract files from archive files of specific types.

Archiver:

Interface to enable runtime-pluggable archive file manipulation via ArchiveCreator and ArchiveExtractor pairs specific to particular archive formats.

ArchiveUtility:

Utility class for manipulating archive files in various formats (as represented by concrete Archiver implementations)

CompressionUtility:

Utility class for instantiating a coder/decoder pair and processing data in various formats. Currently, files, streams, and string buffers are supported.

Codec:

Interface to enable runtime-pluggable coder/decoder pairs. The contract consists only of methods to provide Deflater and Inflater instances.

DefaultCodec:

Implementation of Codec that provides instances of java.util.zip.Deflater and java.util.zip.Inflater objects. This class is deprecated in version 2.0; DeflateCodec should be used instead.

DeflateCodec:

Implementation of Codec that provides instances of java.util.zip.Deflater and java.util.zip.Inflater objects. This class replaces DefaultCodec in version 2.0.

LZ77Codec:

Implementation of Codec that provides instances of LZ77Encoder and LZ77Decoder objects.

LZ77Encoder:

Subclass of java.util.zip.Deflater that implements the LZ77 encoding algorithm.

LZ77Decoder:

Subclass of java.util.zip.Inflater that implements the LZ77 decoding algorithm.

ZipArchiver:

An Archiver that provides ArchiveCreator and ArchiveExtractor implementations suitable for manipulating ZIP archives.

ZipCreator:

An ArchiveCreator that creates ZIP archives.

ZipExtractor:

An ArchiveCreator that examines and extracts files from ZIP archives.

FunctionalTests:

JUnit test suite for use cases involving files and string buffers. All stream methods are tested indirectly during the processing for either of the other two data formats.

1.5 Component Exception Definitions

ClassCastException:

An object was of a class incompatible with an attempted cast.

The ArchiveUtility, ArchiveCreator, ArchiveExtractor, ZipArchiveCreator, and ZipArchiveExtractor methods that accept List arguments throw this exception if the supplied List contains any element that is not a java.io.File.

ClassNotFoundException:

A class was not found.

The CompressionUtility constructor throws this when the Class.forName() call throws a ClassNotFoundException.

IllegalArgumentException:

The argument is illegal.

The CompressionUtility constructor throws this when the class name or the output object is null. Many methods on ArchiveUtility and on ArchiveCreator and ArchiveExtractor implementations throw this exception to signal File arguments that are illegal for various reasons (corresponding file does not exist; corresponding file is a directory, etc.)

IllegalStateException:

An object is not in an appropriate state for the manipulation attempted.

Most ArchiveUtility methods throw this exception if invoked when neither the configured archive file nor its parent directory exists.

IOException:

An I/O error occurred.

Any method that calls java.io methods that throw IOExceptions may throw them also. These are outlined in the Javadocs, and include compress(), decompress() (from CompressionUtility), read(), write(), close() (from the LZ77Encoder and LZ77Decoder), and the constructors for LZ77Codec, LZ77Encoder, LZ77Decoder, as well as all of the methods defined by the ArchiveCreator and ArchiveExtractor interfaces, and the corresponding Façade methods of ArchiveUtility.

LinkageError:

A linkage error occurred.

CompressionUtility.createCodec()throws this when the Class.forName() call throws a LinkageError. It may also be thrown by the CompressionUtility constructor if the error occurs during the call to createCodec().

NullPointerException:

A null reference was encountered where it was not expected.

Many methods of the ArchiveUtility and related classes throw this exception in response to null arguments.

1.6 Thread Safety

The component is not thread safe. Compression, decompression, and archive creation operations in particular are likely to produce garbled results if performed concurrently by two or more threads with use of the same CompressionUtility or ArchiveUtility instance. Deflaters and Inflaters are likewise not thread-safe. ArchiveCreators and ArchiveExtractors *can* be thread-safe relative to VM resources – the ZipArchiveCreator and ZipArchiveExtractor, in particular, maintain no instance state whatsoever, and therefore fall into that category – but because they manipulate external files, they are nevertheless not thread-safe.

2. Environment Requirements

2.1 TopCoder Software Components:

None

2.2 Third Party Components:

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.compression
com.topcoder.util.archiving

3.2 Configuration Parameters

Note that the CompressionUtility class is configurable at runtime.

Parameter	Description	Values
ClassName	Name of the Codec class	"LZ77Codec", "DefaultCodec" (not implemented)

3.3 Dependencies Configuration

None

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

- Include the Compression_Utility.jar file in your classpath.

4.3 Demo

4.3.1 Compression

```
CompressionUtility comp = new CompressionUtility(
"com.topcoder.util.compression.LZ77Codec", myBlankFile );

comp.compress( myInputFile );

comp.close();

// Refer to CompressionTests.java and FunctionalTests.java for
more functional tests
```

4.3.2 Archiving

```
/*
 * Create an ArchiveUtility for managing a ZIP file named
 * "important.zip" and located in directory zips/ relative to the
 * current user directory
 */
ArchiveUtility archive = new ArchiveUtility(
      new File("zips/important.zip"), new ZipArchiver());

/*
 * create a new archive containing all files in the subtree
 * rooted at ../project9, storing their paths relative to that
 * directory in the archive
 */
archive.createArchive(new File("../project9"));
```

```
// Which files were those, anyway?
List importantFiles = archive.listContents();

// Oops – where's that archive again?
File archiveFile = archive.getFile();

// Extract some of the files
List bossFileList = new ArrayList();
bossFileList.add(new File("project_summary.rtf"));
bossFileList.add(new File("project_status.txt"));
bossFileList.add(new File("images/screenshot1.jpg"));
archive.extractContents(
      new File("boss_stuff/project9"), bossFileList);
```

5. Future Enhancements

Additional compression Codecs (bzip2, for example) and Archivers (tar, rar) can be implemented. It might be appropriate to split out the archiving support into a separate component, as it is pretty much unrelated to the compression features.