GISCore User Guide

The GIScore package provides the capability to perform streaming input and output of data from GIS representation file formats, such as ESRI Shapefiles or geo-databases (GDB), Google Earth KML/KMZ, GeoRSS as well as CSV formats. GIScore also provides the mediation between these file formats by converting each format to an internal normalized form.

It was conceived to overcome problems perceived in other prior projects related to in-memory representations of data. In-memory models have many advantages in terms of ease of use and speed, but lack the ability to deal with large data sets. GIScore tries to straddle both of these worlds by providing good performance with modest data set sizes while addressing the ability to deal with arbitrary data set sizes.

The ideal for GIScore was that the object model would be agnostic of the underlying file formats. This is at best unrealistic and factually seemed impossible. GIScore chooses to make the library representative of the richest set of the underlying libraries and ignore features of the object model when a library cannot represent a given feature for a specific implementation. This is one of a number of choices, but one that seemed best to the author, and better than choosing only the common features of each.

# Getting Started

GIScore is built around a factory pattern. The user of a particular stream is meant to be ignorant of the implementation of the stream class that one is using. The caller unfortunately does need to know about extra arguments each document type requires, but not extra arguments based on the underlying class per se.

## Reading a Shapefile

Here is the code to instantiate a shapefile input stream taken from some actual code:

IGISInputStream sis = GISFactory.getInputStream(DocumentType.Shapefile,  
 ngr.getShapefile().getInputStream());

Some things to note. The user passes an input stream that contains a zip input stream holding a directory of shapefiles (which may have just a single shapefile). To switch to a FileGDB the type changes the contents of the zip input stream changes to a GDB directory. KMZ is a zip holding a KML file. There are some other factory methods available as well.

Here's a simple processing loop, taken from the same code. This particular loop looks for rings and polygons to pull out the rings to determine and area of interest to query.

IGISObject obj = sis.read();

while(obj != null)

{

if (obj instanceof Feature) {

Geometry geo = ((Feature) obj).getGeometry();

if (geo instanceof LinearRing) {

processRing(ngr, ((LinearRing) geo).getPoints());

} else if (geo instanceof Polygon) {

processRing(ngr, ((Polygon) geo).getOuterRing().getPoints());

} else if (geo instanceof MultiLinearRings) {

MultiLinearRings mlr = (MultiLinearRings) geo;

for(LinearRing ring : mlr.getLinearRings()) {

processRing(ngr, ring.getPoints());

}

} else if (geo instanceof MultiPolygons) {

MultiPolygons mlp = (MultiPolygons) geo;

for(Polygon poly : mlp.getPolygons()) {

processRing(ngr, poly.getOuterRing().getPoints());

}

} else if (geo instanceof Line) {

processRing(ngr, ((Line) geo).getPoints());

}

}

obj = sis.read();

}

The loop terminates when the read method returns a null value. This particular loop ignores the schema, but yours may want to look at data associated with the features. that's up to you.

Finally, the processing should terminate by closing the input stream, which cleans everything up:

sis.close();

### Writing a Shapefile

Writing any GIS output file is going to be more well determined then reading in one. Why? Because when you're reading one in you don't really need to be concerned with what's required and structure. When you're writing one out you do. Ideally we'd hide all the details from you but we really aren't as clever as we'd all like to be.

That said, you will generally be safe writing a GIS output file if you remember to first write the following elements:

* Schema
* DocumentStart(documenttype)
* ContainerStart("Folder")

Then write your features

Then don't forget to finish by writing out:

* ContainerEnd

To start writing a shapefile you use the output factory:

IGISOutputStream shpos = GISFactory.getOutputStream(DocumentType.Shapefile, zos, outDir);

In addition to the zip output stream we have an output directory specified. This output directory is a scratch directory used to create the actual shapefiles before writing them out to the stream. Ideally we would be able to write the files directly as entries to the zip, but that doesn't work in practice, so we need to write them in the file system first in order to create the zip stream.

Writing to the stream is simple enough, we create the various events and write them to the stream. Here's a test method that writes a simple point geometry:

private static final File shapeOutputDir = new File("testOutput/shptest");

static {

shapeOutputDir.mkdirs();

}

@Test public void testWriteReferencePointOutput() throws Exception {

FileOutputStream zip = new FileOutputStream(new File(shapeOutputDir, "reference.zip"));

ZipOutputStream zos = new ZipOutputStream(zip);

File outDir = new File("testOutput/shptest/buf");

outDir.mkdirs();

IGISOutputStream shpos = GISFactory.getOutputStream(DocumentType.Shapefile, zos, outDir);

Schema schema = new Schema(new URI("urn:test"));

SimpleField id = new SimpleField("testid");

id.setLength(10);

schema.put(id);

DocumentStart ds = new DocumentStart(DocumentType.Shapefile);

shpos.write(ds);

ContainerStart cs = new ContainerStart("Folder");

cs.setName("aaa");

shpos.write(cs);

shpos.write(schema);

for(int i = 0; i < 5; i++) {

Feature f = new Feature();

f.putData(id, "id " + i);

f.setSchema(schema.getId());

double lat = 40.0 + (5.0 \* RandomUtils.nextDouble());

double lon = 40.0 + (5.0 \* RandomUtils.nextDouble());

Point point = new Point(lat, lon);

f.setGeometry(point);

shpos.write(f);

}

shpos.close();

zos.flush();

zos.close();

}

## KML

A *KmlReader* class transparently handles [KML](http://mitrepedia.mitre.org/index.php/KML) and compressed KMZ files by file or URL along with fetching all NetworkLinks. Likewise, a *KmlWriter* class handles creation of KML or KMZ files and optionally allow adding other files as entries in the KMZ (ZIP) file. Most importantly *KmlReader* rewrites all the relative URLs such that they can be traced back to the correct URL and the appropriate resource fetched which would normally be tricky for nested KML/KMZ files and resources referenced within a KMZ file. All versions of KML specifications are imported so don't need to convert older 2.0 and 2.1 KML documents into the latest OGC KML 2.2 spec. The API takes care of most dirty details. The *KmlReader* and *KmlWriter* classes are wrappers for *KmlInputStream* and *KmlOutputStream*, respectively.

While some deprecated and deleted features from older KML specs are implemented to support importing of legacy KML data, KML output conforms to KML 2.2 schema so some deprecated/deleted features may not be preserved on output without some manual intervention. For example having the 'parent' element or attribute appear in the Schema element is correctly handled in import in that aliased elements are converted to Placemarks to be valid KML but the metadata is not auto-converted to *ExtendedData* fields. While some fine-grain customization may be lost in a few cases (e.g. KML regions, models, balloon styles, etc.), the core geospatial and temporal data is preserved in a common representation. The feature structure can be exported to these standard formats once the common representation is created programmically or imported from an existing source.

## Reference

For better or worse the giscore factory methods use variable argument lists. This makes it hard to understand the calling sequences. The following explain the calling sequence for each document type. For each of the following, the call is assumed to be of the form:

GISFactory.getInputStream(DocumentType.<TYPE>,<inputStream>,arguments...)

for the Input cases below or

GISFactory.getOutputStream(DocumentType.<TYPE>,<outputStream>,arguments...)

for the Output cases below

Where <TYPE> is one the types in the list below, and <inputStream> is an appropriate input stream as documented or <outputStream> is an appropriate outputStream as documented.

The FileGDB implementation and the Shapefile input implementation use the ESRI library for support and only work on Windows at this point. The other implementations are written in Java and will work on any platform as far as we know.

Arguments are marked (\*) for required, or unmarked if not required.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Document Type | Input Stream | Input Arguments | Output Stream | Output Arguments |
| Shapefile | ZipInputStream | IAcceptSchema | ZipOutputStream | File(\*) IContainerNameStrategy PointShapeMapper |
| FileGDB | ZipInputStream | IAcceptSchema | ZipOutputStream | File(\*) IContainerNameStrategy |
| CSV | InputStream | Schema  String Character Character | OutputStream | String Character Character Boolean |
| KML/KMZ | InputStream |  | OutputStream |  |
| XMLGDB | n/a | n/a |  |  |

There are also input factory methods to read some document types as files for both convenience and efficiency reasons. It hardly makes sense to zip up a shapefile, filegdb or other GIS file just to read it in. The general outline looks like this:

IGISInputStream stream = GISFactory.getInputStream(DocumentType.<TYPE>,   
 File <dirOrFile>, <varargs>);

The following table describes the file argument and varargs taken for each supported type:

|  |  |  |
| --- | --- | --- |
| DocumentType | File | Arguments |
| KML | KML File |  |
| Shapefile | Directory holding shapefiles | IAcceptSchema |
| FileGDB | Directory holding GDB | IAcceptSchema |
| CSV | CSV File | Schema String Character Character |

Note that *KmlReader/KmlWriter* are advanced helper classes not accessible via the *GISFactory* and must explicitly be instantiated. Likewise, GeoRSS output writer is also not yet added to the Factory method described above.