

G

eography
Markup Lan-
guage (GML)

has emerged in the past several years as a primary means for the exchange, sharing and aggregation of geographic information. This is true increasingly for vector based data and we can anticipate will also be true for imaging data in the near future.

GML application schemas exist for most of the traditional GIS data formats such as S57, VPF, DAFIF, Tiger/Line. In addition, some GML application schemas have become standards in their own right such as cityGML, AIXM, XMMML, and GeoSciML.

With the increased understanding of spatial information infrastructures as a means of sharing geospatial information in real time or near real time, we anticipate increased use of GML for incremental updates to geospatial databases, achieved in most cases through Web Feature Service (WFS) interfaces.

This increased use of GML demands better tools for creating and editing GML, and for reading GML into in memory objects for data manipulation and analysis. Since XML Schema on which GML is based is not truly object oriented and makes use of constructs like choice groups and substitutions, working with GML (or even XML) may be foreign to some developers. This is made all the more complex, by the fact that GML is also a schema definition language, so that while there are many concrete object types defined in GML

gmlSDK

itself, users define entire new sets of objects in GML application schemas. For these reasons, GML may present a steep learning curve for the average developer. For these reasons Galdos developed the gmlSDK, a comprehensive and portable class library written in C++ for the creation of in-memory objects from GML and for writing GML from in memory representations of geographic objects.

To illustrate the use of the gmlSDK we consider a few simple tasks and how to use the library to accomplish them. We start with traversing a GML object.

TRAVERSING A GML OBJECT

A GML object for our purposes is an XML element whose content model is described by a schema component (XML Schema type declaration) in a suitable GML application schema. GML objects are always elements whose children are the properties of the object. We might for example have a road

object of the form:

```
<app:Road xmlns:app="http://sdkexample/road"
  xmlns:gml="http://www.opengis.org/gml">
  <gml:name>Cambie St.</gml:name>
  <app:numOfLanes>4</app:numOfLanes>
  <app:shape>
    <gml:LineString>
      <gml:posList>4800 1234
6678 9012</gml:posList>
    </gml:LineString>
  </app:shape>
</app:Road>
```

Here the properties are clearly gml:name, app:numOfLanes and app:shape.

To traverse this GML fragment, the gmlSDK provides a number of functions including

- getAllProperties() (returns all properties of the object),
- getProperty(propertyName, propertyNamespace),
- getPropertyList(propertyName, propertyNamespace) and
- getPropertyByValueType(objectTypeCode) and
- getPropertyListByValueType(objectTypeCode).



The later functions return the properties with specific names and with specific types.

The following code snippet returns all geometry properties of the road feature as shown above.

```
//get geometry property from the "Road" feature object
```

```
PropertyIterator* geometryPropertyIter = roadObject->getPropertyListByValueType(GMLObjectType::GEOMETRY_TYPECODE);
```

Note that the value of a GML property can be a simple value, a GML object, a list of GML objects, or even a list of arbitrary XML elements. It is thus necessary to provide additional functions (through the property interface) for accessing property value components such as illustrated by the following example.

```
// retrieve the value of the geometry property "shape"
```

```
GMLPropertyNode* shapeProperty = roadObject->getPropertyByValueType((GMLObjectType::GEOMETRY_TYPECODE);
```

```
GMLObjectNode* geometryObject = shapeProperty->getComplexValue();
```

GML schema developers build application schemas using the rules for application schemas and the set of primitives for geometry, topology, coordinate reference systems, coverages, etc that form the core of GML. We refer to these as core objects. The gmlSDK provides a set of accessor functions for getting the property values of these core objects since these are used very frequently in GML applications.

The SDK provides a set of these functions to make it easier and more convenient to deal with GML core objects.

Note that these access functions can be used even in the case where the user

derives an object from a GML core schema object's content model (by restriction or extension). For example, one might create a derived GML object AverageTemp from the GML core object RectifiedGridCoverage by restriction, and then use the built in accessor functions getRangeSet() and setRangeSet() on this derived AverageTemp object to get the values of these properties.

Application (user defined) or core domain objects can be constructed using the SDK. For GML core objects a set of built in object factories (e.g. gml:Polygon, gml:Grid, gml:RectifiedGrid, gml:Feature etc) are provided. For objects not yet supported in the SDK, you can use the provided constructors. Users can thus construct domain objects for all GML core objects and all objects derived from GML objects by restriction using the factories provided in the SDK. A variety of means are provided to simplify the task of object construction. To create an object declared with a type derived from a GML predefined object type, you simply reference the schema that defines the type as follows:

```
GMLObject* object = GMLObjectFactory::makeGMLObject(typeName, qualifiedName, namespaceURI, gmlSchema);
```

For data with types derived from GML predefined types by extension, the additional properties defined in the extended types can be accessed via the underlying GMLObjectNode interface which you can obtain using the toGML() method on the domain object. An alternative way to handle the user-defined data is to develop your own domain objects and object factories. The GMLObjectFactory class provides a method registerFactory(...) that allows clients to register their own object factories

so that their own objects can be created from the GMLObjectFactory in the way in which the domain objects are created from the GML object factory. Of course traversing GML is just the starting point, and the SDK can also be used to read, write and manipulate GML data. Let's look at reading a GML "document".

READING GML

We have seen how to traverse GML data fragments. Now we look at reading GML documents such as the content of GML files or the GML within WFS transactions or WFS data requests.

The SDK provides two approaches, one a tree builder, the other a combination of a tree builder and a streaming reader. Note that the SDK supports switch controllable validation of the GML instance being read relative to the application schema.

The tree builder is intended for fast in memory navigation of GML objects (the document root must be a GML object) and thus is suited only to small GML fragments such as would typically be found in WFS requests and some classes of transactions.

```
GMLObject* gmlDoc = docBuilder->build(dataSource);
```

```
GMLObjectNode* rootObject = gmlDoc->getRootObject();
```

The tree builder provides document constructors. Once the document is constructed you can then freely traverse the document using the accessor functions such as those discussed in Traversing GML.

For larger GML documents, a streaming reader, based on a pull parser that implements most of the StAX (Streaming API for XML) API, and which additionally provides direct GML object sup-

port is provided. This streaming reader overcomes many of the limitations of traditional streaming approaches in that it combines both the "read and discard" and the "build and navigate" approaches in a single API. This allows the programmer to optimize the reading process with respect to their particular application. In particular, the streaming reader enables the programmer to construct sub-trees for those GML objects of interest, and pull them from the parser without reading the whole document and constructing an entire tree in memory. Furthermore, since the pulled GML objects are represented in a tree structure (GMLObjectNode), you can access these with the navigational methods (as we have discussed). Finally, you can construct domain objects from the GML object nodes and access them with setter and getter methods supplied in the SDK.

All of this gives the developer greater freedom to control memory. You can keep the pulled GML objects in memory for later use, or discard them as you wish. To understand the use of the streaming reader in practice, consider its application to the extraction of features from a WFS response. The basic steps would then be:

First, you create a stream reader from the input data source.

Secondly, you write a loop to iterate through the input source and retrieve all features. Within each loop, you need three steps to get a feature object,

```
GMLStreamReader* reader = GMLReaderFactory::createGMLStreamReader(dataSource);
```

- Pull an event with next() method;
- Determine whether you encounter a GML object and particularly a feature object.
- Pull the feature object.

This is illustrated by the following code fragment:

```
while ( reader->hasNext() )
{
    // get the next token
    reader->next();
    // if the pulled token is a GML feature, then pull
    // the feature object and perform further operations.
    if ( reader->isStartObject() && reader->isGMLTypeOf(featureTypeCode) )
    {
        GMLObjectNode* featureNode = reader->pullObject();
        PropertyIterator* geometryProperties = featureNode->getPropertyByValueType(geometryTypeCode);
        // perform further operations on geometry properties.
        ...}
}
```

It should be clear that only a few lines of code are required, and that the code fragment can be applied to any feature collection regardless of the concrete feature types that might appear in that collection. Let's now look at writing some GML examples.

WRITING GML

The SDK provides a number of different ways to write GML from an in-memory representation, namely:

- Serialize a GMLObjectNode or a domain object to a string in memory;
- Serialize a GMLObjectNode to a file, standard output or memory buffer;
- Writing GML data directly to an output in a streaming manner

These different methods are provided to allow the developer the maximum flexibility and simplicity in dealing with GML for different applications.

GMLObjectNodes can be serialized to string XML representation by calling the toXMLString() method. Domain objects can be similarly serialized by first creating a GMLObjectNode using then the toGML() method and then invoking toXMLString(). Thus converting back and forth between XML string and object representations for both core

and user defined objects is a very simple process. For relatively small GML files the SDK supplied functions for file serialization can be used.

For larger files the streaming writer should be employed. It provides a variety of specific methods for writing an element, a GML object, an attribute, namespace declaration etc. The writer automatically escapes characters such as less than sign (<), greater than sign (>) and the ampersand (&). The writeEndElement() method selects the appropriate element to close, you don't even need to specify the element to be closed. More over, the writer takes care of the namespace declaration and can automatically produce a prefix if you did not provide the right one for a given namespace. The streaming writer imposes no limit on the size of GML data stream that can be written.

SUMMARY

The Galdos gmlSDK provides a comprehensive and very flexible approach to dealing with GML data that can provide good performance over a range of applications and a range of memory resources from small to large memory and CPU environments. With the SDK, handling GML is no more complex than other routine programming tasks, and the programmer is largely freed from having to have a detailed understanding of XML Schema or even GML. The SDK has been employed in a variety of applications including image processing (GML JP2), coordinate reference system handling, WFS transaction processing and data format conversion. ■



Ron Lake
Chairman & CEO,
Galdos Systems, Inc.
rlake@galdos.com