OGR Architecture

This document is intended to document the OGR classes. The OGR classes are intended to be generic (not specific to OLE DB or COM or Windows) but are used as a foundation for implementing OLE DB Provider support, as well as client side support for SFCOM. It is intended that these same OGR classes could be used by an implementation of SFCORBA for instance or used directly by C++ programs wanting to use an OpenGIS simple features inspired API.

Because OGR is modelled on the OpenGIS simple features data model, it is very helpful to review the SFCOM, or other simple features interface specifications which can be retrieved from the Open Geospatial Consortium web site. Data types, and method names are modelled on those from the interface specifications.

Class Overview

- **Geometry** (ogr_geometry.h): The geometry classes (**OGRGeometry**, etc) encapsulate the OpenGIS model vector data as well as providing some geometry operations, and translation to/from well known binary and text format. A geometry includes a spatial reference system (projection).
- **Spatial Reference** (ogr_spatialref.h): An **OGRSpatialReference** encapsulates the definition of a projection and datum.
- **Feature** (ogr_feature.h): The **OGRFeature** encapsulates the definition of a whole feature, that is a geometry and a set of attributes.
- **Feature Class Definition** (ogr_feature.h): The **OGRFeatureDefn** class captures the schema (set of field definitions) for a group of related features (normally a whole layer).
- **Layer** (ogrsf_frmts.h): **OGRLayer** is an abstract base class represent a layer of features in an **GDALDataset**.
- **Dataset** (gdal_priv.h): A **GDALDataset** is an abstract base class representing a file or database containing one or more **OGRLayer** objects.
- **Drivers** (gdal_priv.h): A **GDALDriver** represents a translator for a specific format, opening **GDALDataset** objects. All available drivers are managed by the **GDALDriverManager**.

Geometry

The geometry classes are represent various kinds of vector geometry. All the geometry classes derived from **OGRGeometry** which defines the common services of all geometries. Types of geometry include **OGRPoint**, **OGRLineString**, **OGRPolygon**, **OGRGeometryCollection**, **OGRMultiPolygon**, **OGRMultiPoint**, and **OGRMultiLineString**.

GDAL 2.0 extends those geometry type with non-linear geometries with the **OGRCircularString**, **OGRCompoundCurve**, **OGRCurvePolygon**, **OGRMultiCurve** and **OGRMultiSurface** classes.

Additional intermediate abstract base classes contain functionality that could eventually be implemented by other geometry types. These include **OGRCurve** (base class for **OGRLineString**) and **OGRSurface** (base class for **OGRPolygon**). Some intermediate interfaces modelled in the simple features abstract model and SFCOM are not modelled in OGR at this time. In most cases the methods are aggregated into other classes.

The **OGRGeometryFactory** is used to convert well known text, and well known binary format data into geometries. These are predefined ASCII and binary formats for representing all the types of simple features geometries.

In a manner based on the geometry object in SFCOM, the **OGRGeometry** includes a reference to an **OGRSpatialReference** object, defining the spatial reference system of that geometry. This is normally a reference to a shared spatial reference object with reference counting for each of the **OGRGeometry** objects

using it.

Many of the spatial analysis methods (such as computing overlaps and so forth) are not implemented at this time for **OGRGeometry**.

While it is theoretically possible to derive other or more specific geometry classes from the existing **OGRGeometry** classes, this isn't an aspect that has been well thought out. In particular, it would be possible to create specialized classes using the **OGRGeometryFactory** without modifying it.

Compatibility issues with GDAL 2.0 non-linear geometries

Generic mechanisms have been introduced so that creating or modifying a feature with a non-linear geometry in a layer of a driver that does not support it will transform that geometry in the closest matching linear geometry.

On the other side, when retrieving data from the OGR C API, the **OGRSetNonLinearGeometriesEnabledFlag()** function can be used, so that geometries and layer geometry type returned are also converted to their linear approximation if necessary.

Spatial Reference

The **OGRSpatialReference** class is intended to store an OpenGIS Spatial Reference System definition. Currently local, geographic and projected coordinate systems are supported. Vertical coordinate systems, geocentric coordinate systems, and compound (horizontal + vertical) coordinate systems are as well supported in recent GDAL versions.

The spatial coordinate system data model is inherited from the OpenGIS **Well Known Text** format. A simple form of this is defined in the Simple Features specifications. A more sophisticated form is found in the Coordinate Transformation specification. The **OGRSpatialReference** is built on the features of the Coordinate Transformation specification but is intended to be compatible with the earlier simple features form.

There is also an associated **OGRCoordinateTransformation** class that encapsulates use of PROJ.4 for converting between different coordinate systems. There is a tutorial available describing how to use the **OGRSpatialReference** class.

Feature / Feature Definition

The **OGRGeometry** captures the geometry of a vector feature ... the spatial position/region of a feature. The **OGRFeature** contains this geometry, and adds feature attributes, feature id, and a feature class identifier. Starting with OGR 1.11, several geometries can be associated to a **OGRFeature**.

The set of attributes, their types, names and so forth is represented via the **OGRFeatureDefn** class. One **OGRFeatureDefn** normally exists for a layer of features. The same definition is shared in a reference counted manner by the feature of that type (or feature class).

The feature id (FID) of a feature is intended to be a unique identifier for the feature within the layer it is a member of. Freestanding features, or features not yet written to a layer may have a null (OGRNullFID) feature id. The feature ids are modelled in OGR as a 64-bit integer (GDAL 2.0 or later); however, this is not sufficiently expressive to model the natural feature ids in some formats. For instance, the GML feature id is a string.

The feature class also contains an indicator of the types of geometry allowed for that feature class (returned as an OGRwkbGeometryType from **OGRFeatureDefn::GetGeomType()**). If this is wkbUnknown then any type of geometry is allowed. This implies that features in a given layer can potentially be of different geometry types though they will always share a common attribute schema.

Starting with OGR 1.11, several geometry fields can be associated to a feature class. Each geometry field has its own indicator of geometry type allowed, returned by **OGRGeomFieldDefn::GetType()**, and its spatial reference system, returned by **OGRGeomFieldDefn::GetSpatialRef()**.

The **OGRFeatureDefn** also contains a feature class name (normally used as a layer name).

Layer

An **OGRLayer** represents a layer of features within a data source. All features in an **OGRLayer** share a common schema and are of the same **OGRFeatureDefn**. An **OGRLayer** class also contains methods for

reading features from the data source. The **OGRLayer** can be thought of as a gateway for reading and writing features from an underlying data source, normally a file format. In SFCOM and other table based simple features implementation an **OGRLayer** represents a spatial table.

The **OGRLayer** includes methods for sequential and random reading and writing. Read access (via the **OGRLayer::GetNextFeature()** method) normally reads all features, one at a time sequentially; however, it can be limited to return features intersecting a particular geographic region by installing a spatial filter on the **OGRLayer** (via the **OGRLayer::SetSpatialFilter()** method).

One flaw in the current OGR architecture is that the spatial filter is set directly on the **OGRLayer** which is intended to be the only representative of a given layer in a data source. This means it isn't possible to have multiple read operations active at one time with different spatial filters on each. This aspect may be revised in the future to introduce an OGRLayerView class or something similar.

Another question that might arise is why the **OGRLayer** and **OGRFeatureDefn** classes are distinct. An **OGRLayer** always has a one-to-one relationship to an **OGRFeatureDefn**, so why not amalgamate the classes. There are two reasons:

1. As defined now **OGRFeature** and **OGRFeatureDefn** don't depend on **OGRLayer**, so they can exist independently in memory without regard to a particular layer in a data store.

2. The SF CORBA model does not have a concept of a layer with a single fixed schema the way that the SFCOM and SFSQL models do. The fact that features belong to a feature collection that is potentially not directly related to their current feature grouping may be important to implementing SFCORBA support using OGR.

The **OGRLayer** class is an abstract base class. An implementation is expected to be subclassed for each file format driver implemented. OGRLayers are normally owned directly by their **GDALDataset**, and aren't instantiated or destroyed directly.

Dataset

A **GDALDataset** represents a set of **OGRLayer** objects. This usually represents a single file, set of files, database or gateway. A **GDALDataset** has a list of OGRLayers which it owns but can return references to.

GDALDataset is an abstract base class. An implementation is expected to be subclassed for each file format driver implemented. **GDALDataset** objects are not normally instantiated directly but rather with the assistance of an **GDALDriver**. Deleting an **GDALDataset** closes access to the underlying persistent data source, but does not normally result in deletion of that file.

A **GDALDataset** has a name (usually a filename) that can be used to reopen the data source with a **GDALDriver**.

The **GDALDataset** also has support for executing a datasource specific command, normally a form of SQL. This is accomplished via the **GDALDataset::ExecuteSQL()** method. While some datasources (such as PostGIS and Oracle) pass the SQL through to an underlying database, OGR also includes support for evaluating a subset of the SQL SELECT statement against any datasource.

Drivers

A **GDALDriver** object is instantiated for each file format supported. The **GDALDriver** objects are registered with the **GDALDriverManager**, a singleton class that is normally used to open new datasets.

It is intended that a new **GDALDriver** object is instanciated and define function pointers for operations like Identify(), Open() for each file format to be supported (along with a file format specific **GDALDataset**, and **OGRLayer** classes).

On application startup registration functions are normally called for each desired file format. These functions instantiate the appropriate **GDALDriver** objects, and register them with the **GDALDriverManager**. When a dataset is to be opened, the driver manager will normally try each **GDALDataset** in turn, until one succeeds, returning a **GDALDataset** object.