Feature Data Objects (FDO)

Developer's Guide

Autodesk®

February 2006

Published By: Autodesk, Inc.
111 McInnis Parkway
San Rafael, CA 94903, USA
Government Use

1 2 3 4 5 6 7 8 9 10

Contents

About This Guide

1

The *FDO Developer's Guide* introduces the Feature Data Objects (FDO) application programming interface (API) and explains how to use its customization and development features.

NOTE For detailed information about installing the FDO SDK and getting started using the FDO API, see *The Essential FDO* (FET_TheEssentialFDO.pdf).

In this chapter

- Audience and Purpose
- How This Guide Is Organized
- What's New

Audience and Purpose

This guide is intended to be used by developers of FDO applications. It introduces the FDO API, explains the role of a feature provider, and provides detailed information and examples about how to code your application.

How This Guide Is Organized

This guide consists of the following chapters and appendixes:

- Introduction (page 7), provides an overview of the FDO API and the function of FDO feature providers.
- FDO Concepts (page 15), describes the key data and operational concepts upon which FDO is constructed.
- Development Practices (page 25), discusses the best practices to follow when using FDO for application development.
- Establishing a Connection (page 31), describes how to establish a connection to an FDO provider.
- Capabilities (page 45), discusses the Capabilities API, which is used to determine the capabilities of a particular provider.
- Schema Management (page 59), describes how to create and work with schemas and presents the issues related to schema management.
- Data Maintenance (page 93), provides information about using the FDO API to maintain the data.
- Performing Queries (page 103), describes how to create and perform queries.
- Long Transaction Processing (page 111), discusses long transactions (LT) and how to implement LT processing in your application.
- Filter and Expression Languages (page 115), discusses the use of filter expressions to specify to an FDO provider how to identify a subset of the objects of an FDO data store.
- The Geometry API (page 129), discusses the various Geometry types and formats and describes how to work with the Geometry API to develop FDO-based applications.

- Autodesk FDO Provider for Oracle (page 141), discusses development issues that apply when using FDO Provider for Oracle.
- OSGeo FDO Provider for ArcSDE (page 171), discusses development issues that apply when using FDO Provider for ESRI® ArcSDE®.
- OSGeo FDO Provider for MySQL (page 197), discusses development issues that apply when using FDO Provider for MySQL.
- OSGeo FDO Provider for ODBC (page 211), discusses development issues that apply when using FDO Provider for ODBC.
- Autodesk FDO Provider for Raster (page 225), discusses development issues that apply when using FDO Provider for Raster.
- OSGeo FDO Provider for SDF (page 241), discusses development issues that apply when using FDO Provider for SDF.
- OSGeo FDO Provider for SHP (page 255), discusses development issues that apply when using FDO Provider for SHP (Shape).
- Autodesk FDO Provider for SQL Server (page 269), discusses development issues that apply when using FDO Provider for SQL Server.
- OSGeo FDO Provider for WFS (page 283), discusses development issues that apply when using FDO Provider for WFS.
- OSGeo FDO Provider for WMS (page 297), discusses development issues that apply when using FDO Provider for WMS.

What's New

This section summarizes the changes and enhancements you will find in this version of FDO.

Support for Additional FDO Providers

The following Autodesk and OSGeo providers are now supported:

- Autodesk FDO Provider for Oracle
- Autodesk FDO Provider for Raster
- Autodesk FDO Provider for SQL Server
- OSGeo FDO Provider for ArcSDE

- OSGeo FDO Provider for MySQL
- OSGeo FDO Provider for ODBC
- OSGeo FDO Provider for SDF
- OSGeo FDO Provider for SHP
- OSGeo FDO Provider for WFS
- OSGeo FDO Provider for WMS

NOTE For more information about the Open Source Geospatial Foundation (OSGeo), see www.OSGeo.org.

Physical Schema Overrides and XML File Format

A large number of FDO interface changes are introduced for physical schema overrides, or mappings, due to the new providers. The new providers also require updates to the FDO Schema XML file format.

Non-Physical Mapping FDO Interface Changes

The non-physical mapping FDO interface changes are in the following areas:

- **Property Contraints.** Constraints now affect the schema-related classes.
- **XML Serialization.** Support FDO *data* in GML format, as opposed to only the previously supported *schema* in GML format, using a number of enhancements. Specifically, the Web Feature Service (WFS) capabilities for the FDO Provider for WFS is now supported.
- **Long Transactions and Locking.** Now supports the ability to return lock conflicts from long transaction commit and rollback commands and also supports class-level settings to determine whether the class is long transaction version-enabled and persistent locking-enabled.

Bulk Copy

Use the Bulk Copy Utility API to copy data from one FDO data store to another FDO data store. You can either copy a subset or the complete source data store. The data to be copied can be filtered based on:

- Schema (for example, Acad)
- Class (for example, AcDbPolyline)

- Property (for example, layer)
- Filter (for example, layer = “road”)

The Bulk Copy API has the following characteristics:

- A schema is automatically created, if it does not already exist in the target data store.
- Default values can also be set for a target property if the source property does not exist.
- Schema names, class names, and property names from the source data store do not need to match the names in the target data store.
- An XML mapping file is used to setup the mapping between the source and target data store.
- Methods to serialize and deserialize the mapping to an XML mapping file are available.

RDBMS Provider Common Architecture

All API changes are internal. No FDO interface changes are required.

Introduction

2

You can use the APIs in the FDO API to manipulate, define, and analyze geospatial information.

This chapter introduces application development with the FDO API and explains the role of a feature provider.

In this chapter

What Is the FDO API?

From the Perspective of the Client Application User

The FDO API is a set of APIs used for creating, managing, and examining information, enabling Autodesk GIS products to seamlessly share spatial and non-spatial information, with minimal effort.

FDO is intended to provide consistent access to feature data, whether it comes from a CAD-based data source, or from a relational data store that supports rich classification. To achieve this, FDO supports a model that can readily support the capabilities of each data source, allowing consumer applications functionality to be tailored to match that of the data source. For example, some data sources may support spatial queries, while others do not. Also, a flexible metadata model is required in FDO, allowing clients to adapt to the underlying feature schema exposed by each data source.

From the Perspective of the Client Application Engineer

The FDO API provides a common, general purpose abstraction layer for accessing geospatial data from a variety of data sources. The API is, in part, an interface specification of the abstraction layer. A provider, such as Autodesk FDO Provider for Oracle, is an implementation of the interface for a specific type of data source (for example, for an Oracle relational database). The API supports the standard data store manipulation operations, such as querying, updating, versioning, locking, and others. It also supports analysis.

The API includes an extensive set of methods that return information about the capabilities of the underlying data source. For example, one method indicates whether the data source supports the creation of multiple schemas, and another indicates whether the data source supports schema modification.

A core set of services for providers is also available in the API, such as provider registration, schema management, filter and expression construction, and XML serialization and deserialization.

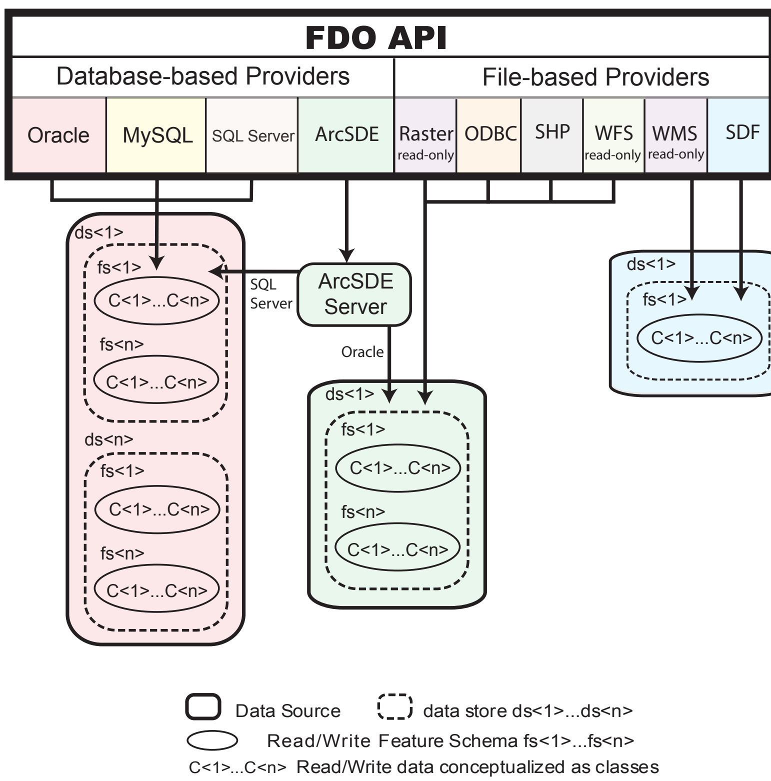
The API uses an object-oriented model for the construction of feature schema. A feature is a class, and its attributes, including its geometry, are a property of the class. The instantiation of a feature class, a Feature Data Object (FDO), can contain other FDOs.

Getting Started

For detailed information to help you install and get started using Feature Data Objects (FDO), see *The Essential FDO*. It provides details about connecting to and configuring providers, data store management (create/delete), user IDs (create, grant permissions), and spatial context.

FDO Architecture and Providers

The following diagram shows the high-level overview architecture of the FDO API and included FDO providers. For clarity, only the underlying data source details for the Autodesk FDO Provider for Oracle, OSGeo FDO Provider for ArcSDE, and OSGeo FDO Provider for SDF are shown as examples. Similar data store, schema, and data connection information is available for the other providers.



FDO Packages

FDO is assembled in conceptual packages of similar functionality. This conceptual packaging is reflected in the substructure of the FDO SDK “includes” folder. For more information about the structure, see *The Essential FDO*.

FDO commands, provider-specific commands, and connections and capabilities provide access to native data stores through each different FDO provider. Schema management (through XML), client services, and filters and expressions are provider-independent packages that tie into the FDO API. Each of these are explained in more detail in subsequent sections.

The FDO API consists of classes grouped within the following packages:

- **Commands package.** Contains a collection of classes that provide the commands allowing the application to select and update features, define new types of feature classes, lock features, and perform analysis on features. Each Command object executes a specific type of command against the underlying data store. In addition, FDO providers expose one or more Command objects.

- **Connections/Capabilities.** Contains a collection of classes that establish and manage the connection to the underlying data store. Connection objects implement the FdoIConnection interface. Capabilities API provides the code for retrieving the various FDO provider capability categories, such as connection or schema capabilities. You can use this this API to determine the capabilities of a particular provider.

- **Filters and Expression package.** Contains a collection of classes that define filters and expression in FDO, which are used to identify a subset of objects of an FDO data store.

- **Client Services package.** Contains a collection of classes that define the client services in FDO that, for example, enable support for dynamic creation of connection objects given a provider name.

- **Schema package and FDO XML.** Contains a collection of classes that provides a logical mechanism for specifying how to represent geospatial features. The FDO feature schema is based somewhat on a subset of the OpenGIS and ISO feature models. FDO feature schemas can be written to an XML file. The FdoFeatureSchema and FdoFeatureSchemaCollection classes support the FdoXmlSerializable interface.

In addition, FDO is integrated with the Geometry API, which includes the classes that support specific Autodesk applications and APIs, including FDO.

For more information about each of the FDO packages, see *FDO API Reference Help* (FDO_API.chm) and subsequent chapters is this guide.

Provider API(s) complete the FDO API configuration. Each provider has a separate API reference Help (for example, SDF_Provider_API.chm).

What Is a Provider?

A provider is a specific implementation of the FDO API. It is the software component that provides access to data in a particular data store.

For this release, the providers that are included are as follows:

NOTE Autodesk FDO Provider for Oracle and FDO Provider for ArcSDE are listed first because they were included in previous releases. The remaining providers are new to this release and are in alphabetical order. Providers referenced in this document with "Autodesk" as part of their name are included only with Autodesk software. Other providers are open source. For more information, see the Open Source Geospatial Foundation website at www.OSGeo.org.

- **Autodesk FDO Provider for Oracle.** Read/write access to feature data in an Oracle-based data store. Supports spatial indexing, long transactions, and persistent locking. A custom API can gather provider information, transmit client services exceptions, list data stores, and create connection objects.
- **OsGeo FDO Provider for ArcSDE.** Read/write access to feature data in an ESRI ArcSDE-based data store (that is, with an underlying Oracle or SQL Server database). Supports describing schema, and inserting, selecting, updating, and deleting feature data in existing schemas; does not support creating or deleting schemas.
- **OsGeo FDO Provider for MySQL.** Read/write access to feature data in a MySQL-based data store. Supports spatial data types and spatial query operations. A custom API can gather information, transmit exceptions, list data stores, and create connection objects. MySQL architecture supports various storage engines, characteristics, and capabilities.
- **OsGeo FDO Provider for ODBC.** Read/write access to feature data in an ODBC-based data store. Supports XYZ feature objects and can define feature classes for any relational database table with X, Y, and optionally Z

columns; does not support creating or deleting schema. Object locations are stored in separate properties in the object definition.

- **Autodesk FDO Provider for Raster.** Read-only access to feature data in raster-based file format. Supports various image and GIS data formats (for example, JPEG, PNG, MrSID, and others). Supports georeferenced file-based raster images and file-based grid coverages. Pixel-based images, such as satellite images, are useful underneath vector data.

- **OsGeo FDO Provider for SDF.** Read-write access to feature data in an SDF-based data store. Autodesk's geospatial file format, SDF, supports multiple features/attributes, provides high performance for large data sets and interoperability with other Autodesk products, and spatial indexing. The SDF provider a valid alternative to database storage. Note that this release of the SDF provider supports version 3.0 of the SDF file format.

- **OsGeo FDO Provider for SHP.** Read/write access to existing spatial and attribute data in an ESRI SHP-based data store, which consists of separate shape files for geometry, index, and attributes. Each SHP and its associated DBF file is treated as a feature class with a single geometry property. This is a valid alternative to database storage but does not support locking.

- **Autodesk FDO Provider for SQL Server.** Read/write access to feature data in a Microsoft SQL Server-based data store. A custom API supports schema read/write access, and geospatial and non-geospatial data read/write access.

- **OsGeo FDO Provider for WFS.** Read-only access to feature data in an OGC WFS-based data store. Supports client/server environment and retrieves geospatial data encoded in GML from one or more Web Feature Services sites. Client/server communication is encoded in XML with the exception of feature geometries, which are encoded in GML. Note that there is no public API documentation for this provider; all functionality is accessible via the base FDO API.

- **OsGeo FDO Provider for WMS.** Read-only access to feature data in an OGC WMS-based data store. Web Map Service (WMS) produces maps of spatially referenced data dynamically from geographic information, which are generally rendered in PNG, GIF, or JPEG, or as vector-based Scalable Vector Graphics (SVG) or Web Computer Graphics Metafile (WebCGM) formats.

FDO supports retrieval and update of spatial and non-spatial GIS feature data through a rich classification model that is based on OpenGIS and ISO standards.

An overview of the relationships between providers, data sources, data stores, and schemas is presented in the FDO Architecture and Providers (page 9) graphic.

For more detailed information about the providers, see the appropriate appendix in this document. Data sources and data stores are discussed in the Establishing a Connection (page 31) chapter. Schema concepts are discussed in the Schema Management (page 59) chapter.

Developing Applications

You will need to perform several major tasks in using the FDO API to develop a custom application. Each of these tasks breaks down into a number of more detailed coding issues.

The major development tasks are:

- Working with the Build Environment
- Establishing a Connection
- Schema Management
- Data Maintenance
- Creating Queries
- Using Custom Commands (Provider-Specific)

These tasks are explored in detail in the chapters that follow.

FDO Concepts

3

Before you can work properly with the FDO API, you need to have a good understanding of its basic, underlying concepts. This chapter defines the essential constructs and dynamics that comprise the FDO API. The definitions of these constructs and dynamics are grouped into two interdependent categories:

- **Data Concepts.** Definitions of the data constructs that comprise the FDO API
- **Operational Concepts.** Definitions of the operations that are used to manage and manipulate the data.

In this chapter

Data Concepts

All concepts that are defined in this section relate to the data that FDO is designed to manage and manipulate.

What Is a Feature?

A feature is an abstraction of a natural or man-made real world object. It is related directly or indirectly to geographic locations. A spatial feature has one or more geometric properties. For example, a road feature might be represented by a line, and a hydrant might be represented by a point. A non-spatial feature does not have geometry, but can be related to a spatial feature which does. For example, a road feature may contain a sidewalk feature that is defined as not containing a geometry.

What Is a Schema?

A schema is a logical description of the data types used to model real-world objects. A schema is not the actual data instances (that is, not a particular road or land parcel), rather it is metadata. A schema is a model of the types of data that would be found in a data store. For example, a schema which models the layout of city streets has a class called Road, and this class has a property called Name. The definition of Road and its associated classes constitute the schema.

For information about foreign schemas, see FDO Provider for Oracle and Foreign Schemas (page 143).

What Is a Schema Override?

A schema override comprises instructions to override the default schema mappings. For example, an RDBMS-type FDO provider could map a feature class to a table of the same name by default. A schema override might map the class to a differently named table, for example, by mapping the "pole" class to the "telco_pole" table.

What is a Schema Mapping

A Schema Mapping is a correspondence between a Schema Element and a physical object in a data store. For example, FDO Provider for Oracle maps each Feature Class onto a table in the Oracle database where the data store resides. The physical structure of data stores for each FDO provider can vary greatly, so the types of Schema Mappings can also vary between providers.

Each provider defines a set of default schema mappings. For example, FDO Provider for Oracle maps a class to a table of the same name by default. These defaults can be overridden by specifying Schema Overrides.

What Are Elements of a Schema?

A schema consists of a collection of schema elements. In the FDO API, schema elements are related to one another by derivation and by aggregation. An element of a schema defines a particular type of data, such as a feature class or a property, or an association. For example, a feature class definition for a road includes the class name (for example, Road), and the class properties (for example, Name, NumberLanes, PavementType, and Geometry).

What Is a Class Type?

A class type is a specialization of the base FDO class definition (FdoClassDefinition). It is used to represent the complex properties of spatial and non-spatial features.

What is a Feature Class?

A feature class is a schema element that describes a type of real-world object. It includes a class name and property definitions, including zero or more geometric properties. It describes the type of data that would be included in object instances of that type.

What Is a Property?

A property is a single attribute of a class and a class is defined by one or more property definitions. For example, a Road feature class may have properties called Name, NumberLanes, or Location. A property has a particular type, which can be a simple type, such as a string or number, or a complex type defined by a class, such as an Address type, which itself is defined by a set of properties, such as StreetNumber, StreetName, or StreetType.

There are five kinds of properties: association properties, data properties, geometric properties, object properties, and raster properties.

Individual properties are defined in the following sections.

What Is an Association Property?

The FdoAssociationPropertyDefinition class is used to model a peer-to-peer relationship between two classes. This relationship is defined at schema creation time and instantiated at object creation time. The association property supports various cardinality settings, cascading locks, and differing delete rules. An FDO filter can be based on association properties and FdoIFeatureReader can handle associated objects through the GetObject() method.

What Is a Data Property?

A data property is a non-spatial property. An instance of a data property contains a value whose type is either boolean, byte, date/time, decimal, single, double, Int16, Int32, Int64, string, binary large object, or character large object.

What Is Dimensionality?

Dimensionality, and the concept of dimension, has two different meanings in the discussion of geometry and geometric property.

The first is called shape dimensionality, and it is defined by the FdoGeometricType enumeration. The four shapes are point (0 dimensions), curve (1 dimensions), surface (2 dimensions), and solid (3 dimensions).

The other is called ordinate dimensionality, and it is defined by the GisDimensionality enumeration. There are four ordinate dimensions: XY, XYZ, XYM, and XYZM. M stands for measure.

What Is a Geometric Property?

An instance of a geometric property contains an object that represents a geometry value. The definition of the geometric property may restrict an object to represent a geometry that always has the same shape, such as a point, or it could allow different object instances to have different dimensions. For example, one geometric property object could represent a point and another could represent a line. Any combination of shapes is permissible in the specification of the geometric types that a geometry property definition permits. The default geometric property specifies that an object could represent a geometry that is any one of the four shapes.

With respect to ordinate dimensionality, all instances of a geometric property must have the same ordinate dimension. The default is XY.

Geometric property definitions have two attributes regarding ordinate dimensionality: HasElevation for Z and HasMeasure for M.

What is a Geometry?

A geometry is represented using geometric constructs either defined as lists of one or more XY or XYZ points or defined parametrically, for example, as a circular arc. While geometry typically is two- or three-dimensional, it may also contain the measurement dimension (M) to provide the basis for dynamic segments.

The geometry types are denoted by the GisGeometryType enumeration and describe the following:

- Point
- LineString (one or more connected line segments, defined by positions at the vertices)
- CurveString (a collection of connected circular arc segments and linear segments)
- Polygon (a surface bound by one outer ring and zero or more interior rings; the rings are closed, connected line segments, defined by positions at the vertices)
- CurvePolygon (a surface bound by one outer ring and zero or more interior rings; the rings are closed, connected curve segments)
- MultiPoint (multiple points, which may be disjoint)
- MultiLineString (multiple LineStrings, which may be disjoint)
- MultiCurveString (multiple CurveStrings, which may be disjoint)
- MultiPolygon (multiple Polygons, which may be disjoint)
- MultiCurvePolygon (multiple CurvePolygons, which may be disjoint)
- MultiGeometry (a heterogenous collection of geometries, which may be disjoint)

Most geometry types are defined using either curve segments or a series of connected line segments. Curve segments are used where non-linear curves may appear. The following curve segment types are supported:

- CircularArcSegment (circular arc defined by three positions on the arc)
- LineStringSegment (a series of connected line segments, defined by positions are the vertices)

There are currently no geometries of type “solid” (3D shape dimensionality) supported.

The FdoIConnection::GetGeometryCapabilities() method can be used to query which geometry types and ordinate dimensionalities are supported by a particular provider.

What Is an Object Property?

An object property is a complex property type that can be used within a class, and an object property, itself, is defined by a class definition. For example, the Address type example described previously in the Property definition. An object property may define a single instance for each class object instance (for example, an address property of a land parcel), or may represent a list of instances of that class type per instance of the owning class (for example, inspection records as a complex property of an electrical device feature class).

What is a Raster Property?

A raster property defines the information needed to process a raster image, for example, the number of bits of information per pixel, the size in pixels of the X dimension, and the size in pixels of the Y dimension, needed to process a raster image.

What Is a Spatial Context?

A spatial context describes the general metadata or parameters within which geometry for a collection of features resides. In particular, the spatial context includes the definition of the coordinate system, spheroid parameters, units, spatial extents, and so on for a collection of geometries owned by features.

Spatial context can be described as the “coordinate system plus identity.” Any geometries that are to be spatially related must be in a common spatial context.

The identity component is required in order to support separate workspaces, such as schematic diagrams, which are non-georeferenced. Also, it supports georeferenced cases. For example, two users might create drawings using some default spatial parameters (for example, rectangular and 10,000x10,000), although each drawing had nothing to do with the other. If the drawings were put into a common database, the users could preserve not only the spatial parameters, but also the container aspect of their data, using spatial context.

For more information about spatial context, see Spatial Context (page 138).

What is a Data Store?

A data store is a repository of an integrated set of objects. The objects in a data store are modeled either by classes or feature classes defined within one or more schemas. For example, a data store may contain data for both a LandUse schema and a TelcoOutsidePlant schema. Some data stores can represent data in only one schema, while other data stores can represent data in many schemas (for example, RDBMS-based data stores, such as Oracle).

Operational Concepts

The concepts that are defined in this section relate to the FDO operations used to manage and manipulate data.

What Is a Command?

In FDO, the application uses a command to select and update features, define new types of feature classes, lock features, version features, and perform some analysis of features. Each Command object executes a specific type of command against the underlying data store. Interfaces define the semantics of each command, allowing them to be well-defined and strongly typed. Because FDO uses a standard set of commands, providers can extend existing commands and add new commands, specific to that provider. Feature commands execute against a particular connection and may execute within the scope of a transaction.

An FDO command is a particular FDO interface that is used by the application to invoke an operation against a data store. A command may retrieve data from a data store (for example, a Select command), may update data in a data store (for example, an Update or Delete command), may perform some analysis (for example, an Activate Spatial Context command), or may cause some other change in a data store or session (for example, a Begin Transaction command).

What Is an Expression?

An expression is a construct that an application can use to build up a filter. An expression is a clause of a filter or larger expression. For example, “Lanes >=4 and PavementType = 'Asphalt'” takes two expressions and combines them to create a filter.

For more information about using expressions with FDO, see Filter and Expression Languages (page 115).

What Is a Filter?

A filter is a construct that an application specifies to an FDO provider to identify a subset of objects of an FDO data store. For example, a filter may be used to identify all Road type features that have 2 lanes and that are within 200 metres of a particular location. Many FDO commands use filter parameters to specify the objects to which the command applies. For example, a Select command uses a filter to identify the objects that the application wants to retrieve. Similarly, a Delete command uses a filter to identify the objects that the application wants to delete from the data store.

For more information about using filters with FDO, see Filter and Expression Languages (page 115).

What Is Locking?

A user can use locking to gain update control of an object in the data store to the exclusion of other users. There are two general types of locks—transaction locks and persistent locks. Transaction locks are temporary and endure only for the duration of the transaction (see What Is a Transaction? (page 22)).

Persistent locks applied to objects by a user remain with the object until either that user removes those locks or the locks are removed by another user with the appropriate authority.

What Is a Transaction?

A transaction changes the data store in some way. The way these changes affect the data store is determined by the transaction's properties. For example, the Atomic property specifies that either all changes happen or non happen. In transaction processing the data store treats a series of commands as a single atomic unit of change to that data store. Either all changes generated by the commands are successful or the whole set is cancelled. A transaction is a single atomic unit of changes to a data store. The application terminates a transaction with either a "commit," which applies the set of changes, or a "rollback," which cancels the set of changes. Further, the data store may automatically roll back a transaction if it detects a severe error in any of the commands within the transaction. A transaction has the following properties:

- **Atomic.** Either all changes generated by the commands within a transaction happen or none happen.
- **Consistent.** The transaction leaves the data store in a consistent state regarding any constraints or other data integrity rules.

- **Isolated.** Changes being made within a transaction by one user are not visible to other users until after that transaction is committed.
- **Durable.** After a transaction is completed successfully, the changes are persistent in the data store on disk and cannot be lost if the program or processor fails.

What Is a Long Transaction?

A long transaction (LT) is an administration unit used to group conditional changes to objects. Depending on the situation, such a unit might contain conditional changes to one or to many objects. Long transactions are used to modify as-built data in the database without permanently changing the as-built data. Long transactions can be used to apply revisions or alternates to an object.

NOTE For this release, the providers that support long transaction processing are Autodesk FDO Provider for Oracle and OSGeo FDO Provider for ArcSDE.

What Is a Root Long Transaction?

A root long transaction is a long transaction that represents permanent data. Any long transaction has a root long transaction as an ancestor in its long transaction dependency graph.

Development Practices

4

This chapter explains several practices to follow when working with the FDO API and provides examples of how to follow these practices.

In this chapter

Memory Management

Some FDO functions (for example, the Create methods) allocate memory when they are called. This memory needs to be freed to prevent memory leaks. All destructors on FDO classes are protected, so you must call a Release() function to destroy them (thus freeing their allocated memory). Each class inherits from the GisIDisposable class, which defines the AddRef() and Release() functions.

In addition, these classes are reference counted, and the count is increased when you retrieve them through a Get function. After finishing with the object, you need to release it (just as with COM objects). The object is destroyed only when the reference count hits 0. For example:

```
FdoFeatureClass* pBase = myClass->GetBaseClass();
...
// Must release reference added by GetBaseClass when done.
GIS_SAFE_RELEASE(pBase);
```

GIS_SAFE_RELEASE (*ptr)

If the “*ptr” argument is not null, GIS_SAFE_RELEASE calls the release() method of the object pointed to by the “*ptr” argument.

GisPtr

A GisPtr smart pointer is provided to help manage memory. You wrap an FDO object in a GisPtr. The object is then released automatically when the GisPtr goes out of scope. The following code illustrates how to use GisPtr:

```
GisPtr<FdoFeatureClass> pBase = myClass->GetBaseClass();
...
// No need to release. Automatically happens when pBase
// is destroyed.
```

Exception Handling

In the FDO API, FdoCommandException class is the exception type thrown from classes in the Commands package, and FdoConnectionException class is the exception type thrown from classes in the Connections package. Both of these exception types derive from a language-level exception class that is environment-specific.

All exceptions are derived from the GisException class. To catch and process specific exception types, nest catch statements as in the following example:

```
try {
... code
}
  catch (FdoCommandException *ex){
    .. process message
    }
  catch (GisException *ex){
    .. process message
    }
```

In some cases, underneath an FDO command, the GIS level throws a GisException. The FDO command then traps the GisException and wraps it in an FdoCommandException (or FdoSchemaException for a schema command). In this case, several messages are returned by one exception. The following example shows how to process multiple messages from one exception:

```
catch ( FdoSchemaException* ex )  {
  // Loop through all the schema messages
  GisException* currE = ex;
  while ( currE ) {
    CW2A msg(currE->GetExceptionMessage());
    acutPrintf ("FdoConnectionException: %s\n", msg);
    currE = currE->GetCause();
```

An application function may need to catch and then re-throw exceptions in order to clean up memory. However, the need to do this can be eliminated by using GisPtr. The following example cleans up memory on error:

```
FdoFeatureClass* pBase = NULL;
try {
pBase = myClass->GetBaseClass();
...
}
catch (...) {
GIS_SAFE_RELEASE(pBase);
throw;
}
// Must release reference added by GetBaseClass when done.
GIS_SAFE_RELEASE(pBase);
```

The catch and rethrow is unnecessary when GisPtr is used:

```
GisPtr<FdoFeatureClass> pBase = myClass->GetBaseClass();
...
```

Managing GisPtr Behaviors

The topics in this section describe several ways that you can manager GisPtr behavior. For more information about managing GisPtr behavior, see the related topics “GisPtr <T> Class Template Reference” and “GisIDisposable Class Reference” in the *FDO Reference Help* and *The Essential FDO*.

Chain Calls

Do not chain calls. If you do, returned pointers will not be released. For example, given an FdoClassDefinition* pclassDef:

```
psz = pclassDef ->GetProperties()->GetItem(0)->GetName())
```

The above code would result in two memory leaks. Instead use:

```
FdoPropertyDefinitionCollection* pprops = pclassDef -> GetProper
ties();
  FdoPropertyDefinition* ppropDef = pprops->GetItem(0);
  psz = propDef->GetName();
  ppropDef->Release();
  pprops->Release();
```

or (with FdoPtr):

```
FdoPtr<FdoPropertyDefinitionCollection> pprops = pclassDef-> Get
Properties();
  FdoPtr<FdoPropertyDefinition> ppropDef = pprops-> GetItem(0);
psz = propDef->GetName();
```

or (also with FdoPtr):

```
psz = FdoPtr <FdoPropertyDefinition> (FdoPtr <FdoPropertyDefini
tionCollection>(pclassDef->GetProperties())-> GetItem(0))->Get
Name();
```

Assigning Return Pointer of an FDO Function Call to a Non-Smart Pointer

If you are assigning the return pointer of an FDO function call to a non-smart pointer, then you should assign that pointer to a GisPtr. For example:

```
GisLineString* P = gf.CreateLineString(...);
GisPtr <GisLineString> p2 = GIS_SAFE_ADDREF(p);
```

5

Establishing a Connection

This chapter explains how to establish a connection to an FDO provider and provides a connection example.

In this chapter

Connection Semantics

Data Sources and Data Stores

The FDO API uses connection semantics to implement access to feature schema data. The term data store is used to refer to a collection of zero or more objects, which instantiate class definitions belonging to one or more FDO feature schema. The connection is to a data store because that is where data objects are stored. The underlying data source technologies used to hold data stores can be relational databases, such as, an Oracle database, or a file-based solution, such as an SDF file.

The mapping of a data store to data source technology can be one-to-one or many-to-one.

- One-to-one when the connection is made by way of FDO Provider for ArcSDE and the ArcSDE server is using an Oracle database.
- Many-to-one when the data source is an Oracle database and the connection is made by way of FDO Provider for Oracle (in this case, the data store is like a container within a container).

When many-to-one mapping is possible, a connection can be made in one or two steps. For more information, see Establishing a Connection (page 34) and *The Essential FDO*.

The underlying data source technologies differ in the connection parameters used for connecting to a particular provider. The values for these parameters are generated during the installation and configuration of the container technologies. For more information about these values and the process of installing and configuring the associated data source technologies, see the appropriate appendix in this document and *The Essential FDO*.

Providers

You connect to a data store by way of a provider. The FDO SDK includes the following providers:

Database-based

- FDO Provider for Oracle
- FDO Provider for ArcSDE
- FDO Provider for SQL Server

- FDO Provider for MySQL

File-based

- FDO Provider for ODBC
- FDO Provider for Raster
- FDO Provider for SDF
- FDO Provider for SHP
- FDO Provider for WFS
- FDO Provider for WMS

The FDO API contains a registry interface that you can use to register or deregister a provider. Sample code for registering and deregistering a provider is located in Connection Example (page 37).

The providers are registered during the initialization of the FDO SDK. In order to connect to a provider, you will need the name of the provider in a particular format: <Company/Foundation/Originator>.<Provider>.<Version>. The <Company/Foundation/Originator> and <Provider> values are invariable. For specific values, see *The Essential FDO*.

In order to connect, you will need the full name including the <Version> value. You can retrieve the full name from the registry and display the set of provider names in a connection menu list. If, for whatever reason, you deregister a provider, save the registry information for that provider in case you want to reregister it again. The provider object returned by the registry has a Set() method to allow you to change values. However, the only value you can safely change is the display name. Sample code for retrieving the provider registry information is located in Connection Example (page 37).

The registry contains the following information about a provider:

- **Name.** The unique name of the feature provider. This name should be of the form <Company/Foundation/Originator>.<Provider>.<Version>, for example, Autodesk.Oracle.3.0 or OSGeo.MySQL.3.0.
- **DisplayName.** A user-friendly display name of the feature provider. The initial values of this property for the pre-registered providers are “Autodesk FDO Provider for Oracle”, “OSGeo FDO Provider for SDF”, etc., or the equivalent in the language of the country where the application is being used.

- **Description.** A brief description of the feature provider. For example, the the OsGeo FDO Provider for SDF description is “Read/write access to Autodesk's spatial database format, a file-based personal geodatabase that supports multiple features/attributes, spatial indexing, and file-locking.”
- **Version.** The version of the feature provider. The version number string has the form <VersionMajor>.<VersionMinor>.<BuildMajor>.<BuildMinor>, for example, 3.0.0.0.
- **FDOVersion.** The version of the feature data objects specification to which the feature provider conforms. The version number string has the form <VersionMajor>.<VersionMinor>.<BuildMajor>.<BuildMinor>, for example, 3.0.1.0.
- **libraryPath.** The FULL library path including the library name of the provider, for example, <FDO SDK Install Location>/bin/FdoRdbms.dll.
- **isManaged.** A flag indicating whether the provider is a managed or unmanaged .NET provider.

Establishing a Connection

As mentioned in a previous section, Connection Semantics (page 32), the FDO API uses a provider to connect to a data store and its underlying data source technology. These data source technologies must be installed and configured. Certain values generated during data source installation and configuration are used as arguments during the connection process. Because the FDO API does not provide any methods to automate the collection and presentation of these configuration values, either the application developer must request the user to input these configuration values during the connection process, or the application developer can provide an application configuration interface, which would populate the application with the container configuration values and thus allow the user to choose them from lists.

NOTE For more information about connecting, see *The Essential FDO*.

A connection can be made in either one or two steps:

- **One-step connection.** If the user sets the required connection properties and calls the connection object’s Open() method once, the returned state is FdoConnectionState_Open, no additional information is needed.

- **Two-step connection.** If the user sets the required connection properties and calls the connection object's Open() method, the returned state is FdoConnectionState_Pending, additional information is needed to complete the connection. In this case, the first call to Open() has resulted in the retrieval of a list of values for a property that becomes a required property for the second call to the Open() method. After the user has selected one of the values in the list, the second call to Open() should result in FdoConnectionState_Open.

Connecting to a data store by way of the Oracle or the ArcSDE provider, for example, can be done in either one or two steps. In the first step, the data store parameter is not required. If the user does not give the data store parameter a value, the FDO will retrieve the list of data store values from the data source so that the user can choose from them during the second step. Otherwise the user can give the data store a value in the first step, and assuming that the value is valid, the connection will be completed in one step.

The following steps are preliminary to establishing a connection:

1. Get the display names for all of the providers in the registry.
2. Use the display names to create a menu list, which the user will select from when making a connection.

After the user initiates a connection attempt, do the following:

1. Loop through the providers in the registry until you match the display name selected by the user from the connection menu with a provider display name in the registry and retrieve the internal name for that provider.
2. Get an instance of the connection manager.
3. Call the manager's CreateConnection() method using the provider internal name as an argument to obtain a connection object.
4. Obtain a connection info object by calling the connection object's GetConnectionInfo() method.
5. Obtain a connection property dictionary object by calling the connection info object's GetConnection Properties() method and use this dictionary to construct a dialog box requesting connection information from the user.

6 Get a list of connection property names by calling the dictionary's GetPropertyNames() method and loop through the list constructing a data entry line in the dialog box for each name in the list.

7 Use the GetLocalizedName method to obtain the label for the data entry line in the dialog.

8 Use the IsPropertyRequired method to determine whether to mark the line as either required or optional; the dialog box handler should not permit the user to click OK in the dialog box unless a required field has a value.

9 Use the IsPropertyProtected method to determine whether the dialog box handler should process the field value as protected data.

10 Use the IsPropertyEnumerable method to determine whether to call the EnumeratePropertyValues method to get a list of valid values. Then,

If IsPropertyEnumerable returns True and EnumeratePropertyValues sets the updates count parameter to 0, then grey (make unavailable) this line in the dialog;

If the count is 1, set the line value to the value in the returned list;

If the count is greater than 1, then set up a spin box for this line containing the list of values and then call the GetProperty method;

If this method returns a value that is in the list, set the exposed spin-box value in the dialog box line to this value;

If this method returns the empty string, call the GetPropertyDefault method and if this returns a value that is in the list, and set the exposed spin-box value in the dialog box line to this value;

Otherwise, set the exposed spin-box value to some value in the returned list.

1 If the property is not enumerable, call the GetProperty method to determine whether you get a non-empty return value, set the value for that line in the dialog box to the return value.

2 If the property is not enumerable and does not yet have a value, call the GetPropertyDefault method and set the value for that line in the dialog box to the return value.

3 After processing each property in the dictionary, expose the dialog.

4 After the user has okayed the connection dialog box and the dialog box handler has determined that all of the required information has been

filled in, the dialog box handler uses the dictionary's SetProperty() method to update the dictionary with the values specified by the user.

5 Call the connection object's Open() method and check the returned connection state value; if the value is FdoConnectionState_Pending, then reconstruct the connection dialog box and present it to the user for further input.

6 If the return value is FdoConnectionState_Open, the connection process is complete.

Connection Example

The following example demonstrates how to establish a connection. The connection is contained within one class, which has the following four public methods:

- void PopulateConnectionMenu(Menu * connectMenu);
- GisString * MapProviderMenuNameToInternalName(GisString * menuName);
- int GetConnectionPropertyValues(FdoIConnectionPropertyDictionary *dictionary, Dialog * connectDialog);
- int ConnectToProvider(GisString * providerMenuName);

This class also has the following three private data members:

- GisPtr<IProviderRegistry> registry;
- GisPtr<IConnectionManager> connectionManager;
- GisPtr<FdoIConnection> connection;

The registry and connectionManager variables are initialized during object creation. The connection variable is given a value by the connection operation.

```
//get the display names for all of the providers in the registry
//and build a connection menu
void
ExerciseFdoUtilities::PopulateConnectionMenu(Menu * connectMenu)
{
  const FdoProviderCollection * providers;
  GisPtr<FdoProvider> provider;
  try {
    providers = registry->GetProviders ();
    GisInt32 providerCount = providers->GetCount();
    GisString * providerDisplayName = NULL;
    for (int i = 0; i < providerCount; i++) {
      provider = providers->GetItem (i);
      providerDisplayName = provider->GetDisplayName();
      // add providerDisplayName to menu
      connectMenu->Add(providerDisplayName);
    }
  }
  catch (GisException* ex) {
  // exception handling
  ex->Release();
  }
}
// user selects a provider from the connection menu
// loop through the registry to match the provider name selected
// by the user with the display names in the registry
// once you get a match, get the provider internal name
GisString *
ExerciseFdoUtilities::MapProviderMenuNameToInternalName(
  GisString * menuName) {
  try {
    const FdoProviderCollection * providers =
registry->GetProviders();
    GisPtr<FdoProvider> provider;
    GisString * providerInternalName = NULL;
    GisInt32 providerCount = providers->GetCount();
    for(int i = 0; i < providerCount; i++) {
      provider = providers->GetItem(i);
      if (wcsicmp(menuName,
        provider->GetDisplayName()) == 0) {
        providerInternalName = provider->GetName();
        break;
      }
```

```
    }
    if (providerInternalName == NULL) {
      // error handling
      return NULL;
    } else {
      return providerInternalName;
    }
  }
  catch (GisException* ex) {
    // exception handling
    ex->Release();
    return NULL;
  }
}
// map the provider menu name to an internal name
// use the connection manager to make a connection object using
// the provider internal name
// get the connection property dictionary from the connection
// object use the dictionary to construct a dialog, which asks
// the user to input values for connection properties specific
// to the provider (see the comments in the
// GetConnectionProperties method)
// use the values given by the user to set the properties in the
// dictionary
// open the connection
// if the connection state returned by the open operation is
// pending, then ask the user for additional connection property
// values and call open again
int
ExerciseFdoUtilities::ConnectToProvider(GisString * providerMe
nuName) {
  GisString * providerInternalName = MapProviderMenuNameToInternal
Name(providerMenuName);
  if (providerInternalName == NULL) {
    return 1;
  }
  GisPtr<FdoIConnectionInfo> connectionInfo;
  GisPtr<FdoIConnectionPropertyDictionary> connectionPropertyDic
tionary;
  Dialog * connectDialog = new Dialog();
  FdoConnectionState connectState;
  int retval = 0;
  try {
```

```
    connection = connectionManager->CreateConnection(providerInter
nalName);
    connectionInfo = connection->GetConnectionInfo();
    connectionPropertyDictionary = connectionInfo->GetConnection
Properties();
    retval = GetConnectionPropertyValues(connectionPropertyDic
tionary, connectDialog);
    if (retval == 0) {
      connectState = connection->Open();
      switch (connectState) {
        case FdoConnectionState_Busy: break;
        case FdoConnectionState_Closed: break;
        case FdoConnectionState_Open : break;
        case FdoConnectionState_Pending :
          retval = GetConnectionPropertyValues(connectionProperty
Dictionary, connectDialog);
          if (retval == 0) {
            connectState = connection->Open();
          }
          break;
        default :
          GisException * ex = GisException::Create(L"connection-
>Open() returned unknown connection state");
          throw ex;
      }
    }
  }
  catch (GisException * ex){
    // error handling
    ex->Release();
    if (connection) {
      connection->Close();
    }
    return 1;
  }
  if (connectState != FdoConnectionState_Open) {
    // error handling
    return 1;
  }
  return 0;
}
// this method constructs the dialog the user fills in with
// values for the connection properties
```

```
// if the user fills in all the required fields and does not
// cancel, the method sets the property values in the property
// dictionary once that is done, the connection can be opened
int
ExerciseFdoUtilities::GetConnectionPropertyValues(
FdoIConnectionPropertyDictionary
    *dictionary, Dialog * dialog) {
  int retval = 0;
  // get the list of property names in the dictionary
  GisString ** propertyNames = NULL;
  GisInt32 nameCount = 0;
  propertyNames = dictionary->GetPropertyNames(nameCount);
  GisString * propertyName = NULL;
  // loop through the property names adding each property to the
  // dialog
  for(int i = 0; i < nameCount; i++) {
    // get the property name
    propertyName = propertyNames[i];
    // get the label to be used for the property input line
    // in the dialog
    GisString * propertyLabel = dictionary->
      GetLocalizedName(propertyName);
    // determine whether to make the entry line required
    // or optional
    bool IsRequired = dictionary->
      IsPropertyRequired(propertyName);
    // determine whether the user input has to be handled in a
    // secure way
    bool IsProtected = dictionary->
      IsPropertyProtected(propertyName);
    // get the actual and default values for the property
    // these could be the empty string, that is, (GisString *)””
    GisString * actualValue = dictionary->
      GetProperty(propertyName);
    GisString * defaultValue = dictionary->
      GetPropertyDefault(propertyName);
    // determine whether the property values are enumerable
    bool IsEnumerable = dictionary->
      IsPropertyEnumerable(propertyName);
    GisString ** EnumeratedValues = NULL;
    GisInt32 numValues = 0;
    if (IsEnumerable) {
      // get the list of valid values
```

```
    EnumeratedValues = dictionary->EnumeratePropertyValues
      (propertyName, numValues);
  }
  // the dictionary entry for this property could possibly be
  // populatetd by a subsequent call to the Open() method
  bool greyOut = false;
  if (IsEnumerable && numValues == 0) {
    greyOut = true;
  }
  // the values are enumerable and there is only one
  else if (IsEnumerable && numValues == 1) {
    // add the line to the dialog,
    // setting the spin box value to EnumeratedValues[0]
  }
  // the values are enumerable and there is more than one
  else if (IsEnumerable && numValues > 1) {
    // add the line to the dialog,
    // setting the spin box value to the actualValue if
    // not empty, or setting it to the defaultValue if
    // valid and not empty, or setting it to one of the
    // enumerated values
  }
  // set the field to the actual value if not empty
  else if ( wcscmp(actualValue, L"") != 0) {
    // add line to dialog
  }
  // set the field to whatever is the default value
  else {
    // add line to dialog
  }
}
// blocks until user clicks ok or cancel in dialog
// returns 0 if user clicks ok and all required input is
// there and valid if user doesn't fill in required fields,
// dialog persists until user does so or presses cancel
// returns positive if user cancels
retval = dialog->expose();
if (retval == 0) {
  GisString * value = NULL;
  for(int i = 0; i < nameCount; i++) {
    // get the property name
    propertyName = propertyNames[i];
```

```
      // get the value input by the user for this property
      value = dialog->GetValue(propertyName);
      dictionary->SetProperty(propertyName, value);
    }
  }
  return retval;
}
```

6

Capabilities

This chapter explains the Capabilities API and provides the code for retrieving the various FDO provider capability categories, such as connection or schema capabilities. You can use this this API to determine the capabilities of a particular provider.

In this chapter

What Is the Capabilities API?

You can use this API and its various capability categories to determine the capabilities of a particular provider, for example, FDO Provider for Oracle. The capabilities methods can be used to execute code conditionally, depending on which provider is being used and which capability is being exercised.

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Command
- Expression
- Filter
- Geometry
- Raster
- Topology

NOTE Topology-related samples are provided for informational use only. There is no interface or support provided. Autodesk reserves the right to change the software related to the content herein.

The capabilities are retrieved by using methods belonging to an FdoIConnection object. First, you connect to the provider. Then, you query its capabilities.

The sections in this chapter describe how to retrieve the capabilities for each of the categories. In each section, the code fragment assumes that you have connected to the provider and declared the following connection object:

```
#include <fdo.h>
FdoIConnection * connection;
```

Connection Capabilities

Code

Declarations

The object and variable declarations are the following:

```
FdoIConnectionCapabilities * connectionCapabilities;
// FdoThreadCapability is an enumerated type
FdoThreadCapability threadCapability;
// FdoSpatialContextExtentType is an enumerated type
FdoSpatialContextExtentType * spatialContextExtentTypes;
GisInt32 numSpatialContexts;
// FdoLockType is an enumerated type
FdoLockType * lockTypes;
GisInt32 numLockTypes;
bool supportsLocking;
bool supportsTimeout;
bool supportsTransactions;
bool supportsLongTransactions;
bool supportsSQL;
bool supportsConfiguration;
```

Method calls

The method calls are the following:

```
connectionCapabilities = connection->GetConnectionCapabilities();
// GetThreadCapability() returns a single value
threadCapability = connectionCapabilities->GetThreadCapability();
// GetSpatialContextTypes() returns a list
spatialContextExtentTypes = connectionCapabilities->
  GetSpatialContextTypes(numSpatialContexts);
// loop through the spatialContextExtentTypes
supportsLocking = connectionCapabilities->SupportsLocking();
// GetLockTypes() returns a list
lockTypes = connectionCapabilities->GetLockTypes(numLockTypes);
// loop through the lockTypes
supportsTimeout = connectionCapabilities->SupportsTimeout();
supportsTransactions = connectionCapabilities->
  SupportsTransactions();
supportsLongTransactions = connectionCapabilities->
  SupportsLongTransactions();
supportsSQL = connectionCapabilities->SupportsSQL();
supportsConfiguration = connectionCapabilities->
  SupportsConfiguration();
```

Reference

For more information, see these *FDO API Reference Help* topics:

- class FdoIConnectionCapabilities
- enum FdoLockType
- enum FdoSpatialContextExtentType
- enum FdoThreadCapability

Schema Capabilities

Code

Declarations

The object and variable declarations are the following:

```
FdoISchemaCapabilities * schemaCapabilities;
// FdoClassType is an enumerated type
FdoClassType * classTypes;
// FdoDataType is an enumerated type
FdoDataType * dataTypes;
bool supportsInheritance;
bool supportsMultipleSchemas;
bool supportsObjectProperties;
bool supportsAssociationProperties;
bool supportsSchemaOverrides;
bool supportsNetworkModel;
bool supportsAutoIdGeneration;
bool supportsDataStoreScopeUniqueIdGeneration;
FdoDataType * autoGeneratedTypes;
bool supportsSchemaModification;
```

Method Calls

The method calls are the following:

```
schemaCapabilities = connection->GetSchemaCapabilities();
// this returns a list of FdoClassType
classTypes = schemaCapabilities->GetClassTypes();
// loop through the classTypes// this returns a list of
FdoDataType
dataTypes = schemaCapabilities->GetDataTypes();
// loop through the dataTypes
supportsInheritance = schemaCapabilities->SupportsInheritance();
supportsMultipleSchemas = schemaCapabilities->
  SupportsMultipleSchemas();
supportsObjectProperties = schemaCapabilities->
  SupportsObjectProperties();
supportsAssociationProperties = schemaCapabilities->
  SupportsAssociationProperties();
supportsSchemaOverrides = schemaCapabilities->
  SupportsSchemaOverrides();
supportsNetworkModel = schemaCapabilities->SupportsNetworkModel();
supportsAutoIdGeneration = schemaCapabilities->
  SupportsAutoIdGeneration();
supportsDataStoreScopeUniqueIdGeneration = schemaCapabilities->
  SupportsDataStoreScopeUniqueIdGeneration();
// this returns a list of FdoDataType
autoGeneratedTypes = schemaCapabilities->
  GetSupportedAutoGeneratedTypes();
supportsSchemaModification = schemaCapabilities->
  SupportsSchemaModification();
```

References

For more information, see these *FDO API Reference Help* topics:

- class FdoISchemaCapabilities
- enum FdoClassType
- enum FdoDataType

Command Capabilities

Code

Declarations

The object and variable declarations are the following:

```
FdoICommandCapabilities * commandCapabilities;
// this will contain values of type FdoCommandType and//
possibly values of type FdoRdbmsCommandType, which are//
provider-specific commands
GisInt32 * commandTypes;
bool supportsParameters;
bool supportsTimeout;
bool supportsSelectExpressions;
bool supportsSelectFunctions;
bool supportsSelectDistinct;
bool supportsSelectOrdering;
bool supportsSelectGrouping;
```

Method Calls

The method calls are the following:

```
commandCapabilities = connection->GetCommandCapabilities();
// this returns a list of command types
commandTypes = commandCapabilities->GetCommands();
// loop through the commandTypes
supportsParameters = commandCapabilities->SupportsParameters();
supportsTimeout = commandCapabilities->SupportsTimeout();
supportsSelectExpressions = commandCapabilities->
  SupportsSelectExpressions();
supportsSelectFunctions = commandCapabilities->
  SupportsSelectFunctions();
supportsSelectDistinct = commandCapabilities->
  SupportsSelectDistinct();
supportsSelectOrdering = commandCapabilities->
  SupportsSelectOrdering();
supportsSelectGrouping = commandCapabilities->
  SupportsSelectGrouping();
```

References

For more information, see these *FDO API Reference Help* topics:

- class FdoICommandCapabilities
- enum FdoCommandType
- enum FdoRdbmsCommandType

Expression Capabilities

Code

Declarations

The object and variable declarations are the following:

```
FdoIExpressionCapabilities * expressionCapabilities;
GisInt32 numExpressionTypes = 0;
// this is an enumerated type
FdoExpressionType * expressionTypes;
GisInt32 numFunctionDefinitions = 0;
FdoFunctionDefinitionCollection * functions;
FdoFunctionDefinition * functionDefinition;
GisString * functionName;
GisString * functionDescription;
GisInt32 numArgumentDefinitions = 0;
FdoReadOnlyArgumentDefinitionCollection * arguments;
FdoArgumentDefinition * argumentDefinition;
GisString * argumentName;
GisString * argumentDescription;
FdoDataType argumentType;
```

Method Calls

The method calls are the following:

```
expressionCapabilities = connection->GetExpressionCapabilities();
// this returns a list of expression types
expressionTypes = expressionCapabilities->GetExpressionTypes();
// loop through the expression Types
functions = expressionCapabilities->GetFunctions();
numFunctionDefinitions = functions->GetCount();
for (int i = 0; i < numFunctionDefinitions; i++) {
  functionDefinition = functions->GetItem(i);
  functionName = functionDefinition->GetName();
  functionDescription = functionDefinition->GetDescription();
  arguments = functionDefinition->GetArguments();
  numArgumentDefinitions = arguments->GetCount();
  for ( int j = 0; j < numArgumentDefinitions; j++) {
    argumentDefinition = arguments->GetItem(j);
    argumentName = argumentDefinition->GetName();
    argumentDescription = argumentDefinition->GetDescription();
    argumentType = argumentDefinition->GetDataType();
  }
}
```

Filter Capabilities

Code

Declarations

The object and variable declarations are the following:

```
FdoIFilterCapabilities * filterCapabilities;
GisInt32 numConditionTypes = 0;
FdoConditionType * conditionTypes;
GisInt32 numSpatialOperations = 0;
FdoSpatialOperations * spatialOperations;
GisInt32 numDistanceOperations = 0;
FdoDistanceOperations * distanceOperations;
bool supportsGeodesicDistance;
bool supportsNonLiteralGeometricOperations;
```

Method Calls

The method calls are the following:

```
filterCapabilities = connection->GetFilterCapabilities();
conditionTypes = filterCapabilities->
  GetConditionTypes(numConditionTypes);
// loop through conditionTypes
spatialOperations = filterCapabilities->
  GetSpatialOperations(numSpatialOperations);
// loop through spatialOperations
distanceOperations = filterCapabilities->
  GetDistanceOperations(numSpatialOperations);
// loop through distanceOperations
supportsGeodesicDistance = filterCapabilities->
  SupportsGeodesicDistance();
supportsNonLiteralGeometricOperations = filterCapabilities->
  SupportsNonLiteralGeometricOperations();
```

Geometry Capabilities

Code

Declarations

The object and variable declarations are the following:

```
FdoIGeometryCapabilities * geometryCapabilities;
GisInt32 numGeometryTypes = 0;
GisGeometryType * geometryTypes;
GisInt32 numGeometryComponnentTypes = 0;
GisGeometryComponentType * geometryComponentTypes;
GisInt32 dimensionalities = 0;
```

Method calls

The method calls are the following:

```
geometryCapabilities = connection->GetGeometryCapabilities();
geometryTypes = geometryCapabilities->
  GetGeometryTypes(numGeometryTypes);
// loop through geometryTypes
geometryComponentTypes = geometryCapabilities->
  GetGeometryComponentTypes(numGeometryComponentTypes);
// loop through geometryComponentTypes
dimensionalities = geometryCapabilities->GetDimensionalities();
// GisDimensinality_XY is 0 and so is always a given
if (dimensionalities & GisDimensionality_Z) {
  // do whatever
}
if (dimensionalities & GisDimensionality_M) {
  // do whatever
}
```

Raster Capabilities

Code

Declarations

The object and variable declarations are the following:

```
FdoIRasterCapabilities * rasterCapabilities;
bool supportsRaster;
bool supportsStitching;
bool supportsSubsampling;
bool supportsDataModel;
FdoRasterDataModel * rgbRasterDataModel;
```

Method calls

The method calls are the following:

```
rasterCapabilities = connection->GetRasterCapabilities();
supportsRaster = rasterCapabilities->SupportsRaster();
if (supportsRaster) {
  supportsStitching = rasterCapabilities->SupportsStitching();
  supportsSubsampling = rasterCapabilities->SupportsSubsampling();
  rgbRasterDataModel = FdoRasterDataModel::Create();
  rgbRasterDataModel->
    SetDataModelType(FdoRasterDataModelType_RGB);
  rgbRasterDataModel->SetBitsPerPixel(64);
  rgbRasterDataModel->
    SetOrganization(FdoRasterDataOrganization_Image);
  rgbRasterDataModel->SetTileSizeX(64);
  rgbRasterDataModel->SetTileSizeY(128);
  supportsDataModel = rasterCapabilities->
    SupportsDataModel(rgbRasterDataModel);
}
```

Topology Capabilities

Code

Declarations

The object and variable declarations are the following:

```
FdoITopologyCapabilities * topologyCapabilities;
bool supportsTopology;
bool supportsToplogicalHierarchy;
bool breakCurveCrossingsAutomatically;
bool activatesTopologyByArea;
bool constrainsFeatureMovements;
```

Method Calls

The method calls are the following:

```
topologyCapabilities = connection->GetTopologyCapabilities();
supportsTopology = topologyCapabilities->SupportsTopology();
if (supportsTopology) {
  supportsTopologyicalHierarchy = topologyCapabilities->
  SupportsTopologicalHierarchy();
  breaksCurveCrossingsAutomatically = topologyCapabilities->
  BreaksCurveCrossingsAutomatically();
  activatesTopologyByArea = topologyCapabilities->
    ActivatesTopologyByArea();
  constrainsFeatureMovements = topologyCapabilities->
    ConstrainsFeatureMovements();
}
```

NOTE Topology-related samples are provided for informational use only. There is no interface or support provided. Autodesk reserves the right to change the software related to the content herein.

7

Schema Management

This chapter describes how to create and work with schemas and explains some issues related to schema management. For example, you can use the FDO feature schema to specify how to represent geospatial features.

In this chapter

Schema Package

The FDO feature schema provides a logical mechanism for specifying how to represent geospatial features. FDO providers are responsible for mapping the feature schema to some underlying physical data store. The FDO feature schema is based somewhat on a subset of the OpenGIS and ISO feature models. It supports both non-spatial features and spatial features.

The Schema package contains a collection of classes that define the logical feature schema. These classes can be used to set up a feature schema and to interrogate the metadata from a provider using an object-oriented structure. The logical feature schema provides a logical view of geospatial feature data that is fully independent from the underlying storage schema. All data operations in FDO are performed against the classes and relationships defined by the logical feature schema. For example, different class types in the feature schema are used to describe different types of geospatial objects and spatial features.

Base Properties

All classes in the feature schema support the concept of base properties, which are properties that are pre-defined either by the FDO API or by a specific FDO feature provider. For example, all classes in the schema have two base properties: ClassName and SchemaName. These properties can be used to query across an inheritance hierarchy or to process the results of heterogeneous queries. FDO feature providers can also predefine base properties. The following base properties are predefined by the FDO API:

Property Name	Required	Description
SchemaName	Y	Name of the schema to which objects of the class belong; read-only string.
ClassName	Y	Name of the class that defines the object; read-only string.
RevisionNumber	N	Revision number of the object; read-only 64-bit integer. **NOTE** Some providers may use this property to support optimistic locking.

Cross-Schema References

Some FDO feature providers may support multiple schemas. For these providers, the feature schema supports the concept of cross-schema references for classes. This means that a class in one schema may derive from a class in another schema, relate to a class in another schema, or contain an object property definition that is based on a class in another schema.

Parenting in the Schema Classes

The feature schema object model defined in the FDO API supports full navigation through parenting. That is, once a schema element is added to an FdoFeatureSchema class, it can navigate the object hierarchy upward to the root FdoFeatureSchema and, from there, to any other element in the feature schema. This parenting support is fully defined in the FdoSchemaElement abstract base class.

When inserting features that have object collections, the parent object instance must be identified when inserting the child objects (for example, a parent class “Road” has an object property called “sidewalks” of type “Sidewalk”). For more information, see Data Maintenance (page 93).

Physical Mappings

Each feature provider maps the logical feature schema to an underlying physical data store. Some feature providers may provide some level of control over how the logical schema gets mapped to the underlying physical storage. For example, an RDBMS-based feature provider may allow table and column names to be specified for classes and properties. Since this is entirely provider-dependent, the FDO API simply provides abstract classes for passing physical schema and class mappings to the provider (FdoPhysicalSchemaMapping, FdoPhysicalClassMapping, FdoPhysicalPropertyMapping, and FdoPhysicalElementMapping, respectively). The implementation of these abstract classes is up to each feature provider.

Schema Overrides

Using schema overrides, FDO applications can customize the mappings between Feature (logical) Schemas and the Physical Schema of the provider data store.

Schema overrides are provider-specific because different FDO providers support FDO data stores with widely different physical formats. Therefore, the types of schema mappings in these overrides also vary between providers. For example, an RDBMS-type provider might provide a mapping to index a set of columns in a class table. However, other providers would not necessarily be able to work with the concept of an index. For information about schema overrides support by a specific provider, see the appropriate appendix in this document and *The Essential FDO*.

NOTE Some providers support only default schema mappings.

Working with Schemas

There are three primary operations involved with schema management:

- Creating a schema
- Describing a schema
- Modifying a schema

Creating a Schema

The following basic steps are required to create a schema (some steps are optional; some may be done in an alternate order to achieve the same result):

- Use the FdoFeatureSchema::Create("SchemaName", "FeatureSchema Description") method to create a schema.
- Use the FdoFeatureSchema::GetClasses() method to return a class collection.
- Use the FdoClass::Create("className", "classDescription") or FdoFeatureClass::Create("className", "classDescription") method to create FdoClass or FdoFeatureClass type objects.
- Use the FdoClassCollection::Add(class) method to add FdoClass or FdoFeatureClass objects to the class collection.
- Use the FdoGeometricPropertyDefinition::Create("name", "Description") method to create FdoGeometryProperty.
- Use the FdoDataPropertyDefinition::Create("name", "Description") method to create FdoDataProperty.
- Use the FdoObjectPropertyDefinition::Create("name", "Description") method to create FdoObjectProperty.

- Use the FdoClassDefinition::GetProperties() and Add(property) methods to add property to class definition.
- Use the FdoIApplySchemaCommand::SetFeatureSchema(feature schema) method to set the schema object for the IFdoApplySchemaCommand.
- Use the FdoAssociationPropertydefinition class to represent the association between two classes. The class of the associated class must already be defined in the feature schema and cannot be abstract.
- Use the FdoIApplySchemaCommand::Execute() method to execute changes to the feature schema.

For an example of schema creation, see Example: Creating a Feature Schema (page 85).

Use the FdoClassDefinition::GetIdentityProperties() and Add(Property Object) methods to set the property as FdoClass or FdoFeatureClass Identifier. FDO allows multiple Identifiers for both types of classes, although Identifiers have slight differences in both cases.

FDOFeatureClass

FdoFeatureClass is a class that defines features. In the case of GIS, they would often be spatial features, having some sort of geometry associated with them. In most providers, FdoFeatureClass requires a unique identifier to distinguish the features.

However, there are identifiers only if no base class exists. If the base class has an identifier, the child class does not have one. You cannot set an identifier to the child class. Any class definition that has a base class cannot also have any identity properties because it inherits from the base class.

Therefore, you cannot send an identifier when a feature class is a child since it always inherits the identifier from the base class.

FDOClass

This class is used for non-spatial data. It can act as a stand-alone class, where it would have no association with any other class, or if the FdoClass is being used as an ObjectProperty, it can be used to define properties of some other FdoClass or FdoFeatureClass.

ObjectProperty Types

ObjectProperties have the following types:

- Value
- Collection
- OrderedCollection

The Value ObjectProperty type has a relationship of one-to-one, providing a single value for each property.

The Collection and OrderedCollection ObjectProperty types have a one-to-many relationship, where many ObjectProperties may be associated with one property. Ordered Collections can be stored in an ascending or descending order

At least one Identifier will be required if the FdoClass is to be used as a stand-alone Class.

- All Identifiers for FdoDataType_Int64 must not be Read-Only, since none of these will be an auto-generated property value.
- If creating multiple Identifiers, all Identifiers must be set to NOT NULL.

Non-Feature Class Issues

A non-feature class in FDO can be created as a stand-alone class, a contained class, or both. As a contained class, it defines a property of another class or feature class (see FdoFeatureClass and FdoClassType Global Enum). How this non-feature class is created affects the way the data is inserted, queried, and updated.

Stand-alone Class

This type of class stores non-feature data (for example, manufacturers). The FdoClassType_Class must be created with one or more identity properties (see FdoObjectPropertyDefinition), which is required in order that the class has a physical container (that is, a table in the RDBMS) associated with it. If the class is created without specifying an IdentityProperty, only the definition is stored in the metadata, which prevents any direct data inserts.

Contained Class

This type of class stores non-feature data that defines a property of another class or feature class (for example, Sidewalk could be a property of a Road feature class; the Sidewalk class defines the Road.Sidewalk property). In this case, the FdoClassType_Class does not need to be created with one or more identity properties, although it can be.

Class With IdentityProperty Used as ObjectProperty

This type of class reacts like a stand-alone class; however, with this type, it is possible to do direct data inserts. It can also be populated through a container class (for example, Road.Sidewalk) since it defines an object property (see FdoObjectPropertyDefinition). If this class is queried directly, only the data inserted into the class as a stand-alone is returned. The data associated with the ObjectProperty can only be queried through the container class (for example, Road.Sidewalk).

Class Without a Defined IdentityProperty Used as ObjectProperty

Because this class has no defined IdentityProperty, it can only be populated through the container class (for example, Road.Sidewalk) since it defines ObjectProperty. This class cannot be queried directly. The data associated with the object property can only be queried through the container class (for example, Road.Sidewalk). As an object property, it is defined as one of the following:

- **Value type property.** Does not need any identifier since it has a one-to-one relationship with the container class.
- **Collection type property.** Requires a local identifier, which is an identifier defined when creating the ObjectProperty object.
- **Ordered Collection type property.** Requires a local identifier, which is an identifier defined when creating the ObjectProperty object.

When defining either a Collection or Ordered Collection type ObjectProperty, you must set an IdentityProperty attribute for that object property. This ObjectClass.IdentityProperty acts only as a local identifier compared to the IdentityProperty set at the class level. As a local identifier, it acts to uniquely identify each item within each collection (for example, if the local identifier for Road.Sidewalk is Side, there can be multiple sidewalks with Side=”Left” but only one per Road).

Describing a Schema

Use the FdoIDescribeSchema::Execute function to retrieve an FdoFeatureSchemaCollection in order to obtain any information about existing schema(s). The FdoFeatureSchemaCollection consists of all FdoFeatureSchemas in the data store and can be used to obtain information about any schema object, including FdoFeatureSchema, FdoClass, FdoFeatureClass, and their respective properties. The following functions return the main collections required to obtain information about all schema objects:

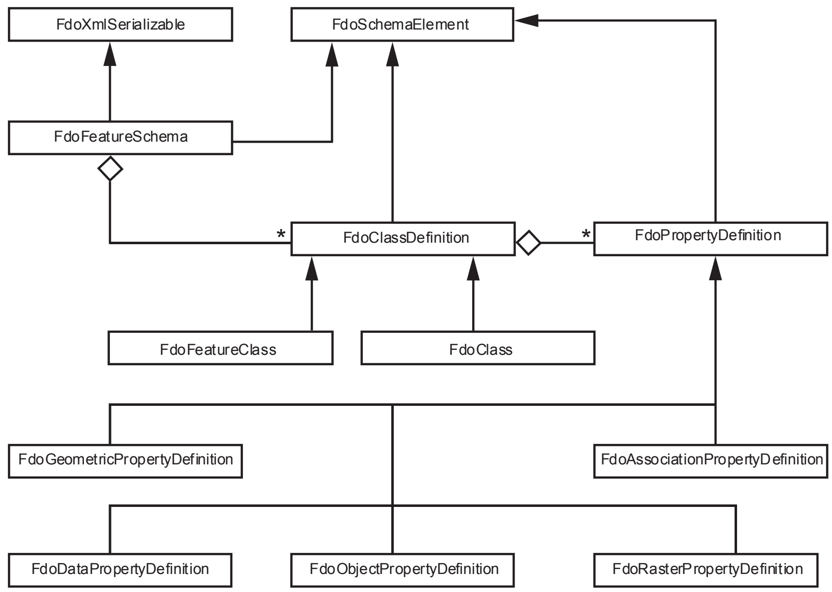
- FdoFeatureSchema::GetClasses method obtains FdoClass and FdoFeatureClasses.
- FdoClassDefinition::GetProperties method obtains a FdoPropertyDefinitionCollection.
- FdoClassDefinition::GetBaseProperties method obtains a FdoPropertyDefinitionCollection of the properties inherited from the base classes.

NOTE Even if your schema has no base classes (inheritance), all classes will inherit some properties from system classes.

Use these functions throughout the application to obtain any information about schema objects. For example, in order to insert data into a class, you must use these functions to determine what data type is required. Description of the data is separate from actions.

The example in the following link is a simple function that shows how to use FdoIDescribeSchema and loop through the schema and class containers to search for duplicate class names. It searches all schemas to ensure that the class name does not exist in any schema in the data store. Class names must be unique across the entire FDO database.

For a schema description example, see Example: Describing a Schema and Writing It to an XML File (page 88).



FDO Schema Element Class Diagram

Modifying Models

Add schema elements to a model by inserting them into the appropriate collection.

Elements are removed from the model by using either of the following methods:

- Call the FdoSchemaElement::Delete() method. This flags the element for deletion when the changes are accepted (generally through FdoIApplySchema), but the element remains a member of all collections until that time.
- Remove the element from the appropriate collection via the FdoSchemaCollection::Remove() or FdoSchemaCollection::RemoveAt() methods. This immediately disassociates the element from the collection.

Schema Element States

All elements within the model maintain a state flag. This flag can be retrieved by calling FdoSchemaElement::GetElementState(), but it cannot be directly set. Instead, its state changes in reaction to the changes made to the model:

- **Unchanged.** When a schema model is retrieved via FdoIDescribeSchema, all elements are initially marked Unchanged.
- **Detached.** Removing an element from an owning collection sets its state to Detached.
- **Deleted.** Calling the Delete() method on an element sets its state to Deleted.
- **Added.** Placing an element within a collection marks the element as Added.
- **Modified.** When adding or removing a sub-element, such as a property element from a class, the class element state will be changed to Modified.

Additionally, when an element that is contained by another element is changed in any way, the containing element is also marked as Modified. So, for example, if a new value is added to the SchemaAttributeDictionary of the "Class3" element in our model, both the "Class3" FdoClass object and the FdoFeatureSchema object would be marked as Modified.

The state flags are maintained until the changes are accepted, that is, when IApplySchema is executed. At that time, all elements marked Deleted are released and all other elements are set to Unchanged.

NOTE When you remove an element from an owning collection, its state is marked as Detached. All collections currently in FDO are owning collections, except for one, the collections FdoClassDefinition::GetIdentityProperties().

Rollback Mechanism

The FdoFeatureSchema contains a mechanism that allows you to "roll back" model changes to the last accepted state. For example, a model retrieved via FdoIDescribeSchema can have classes added, attributes deleted, or names and default values changed. All of these changes are thrown out and the model returned to its unmodified state by calling FdoFeatureSchema::RejectChanges().

The converse of this operation is the FdoFeatureSchema::AcceptChanges() method, which removes all of the elements with a status of Deleted and sets the state flag of all other elements to Unchanged. Generally, this method is only invoked by FDO provider code after it has processed an FdoIApplySchema::Execute() command. Normal FDO clients should not call this method directly.

FDO XML Format

FDO feature schemas can be written to an XML file. The FdoFeatureSchema and FdoFeatureSchemaCollection classes support the FdoXmlSerializable interface. The sample code shows an FdoFeatureSchema object calling the WriteXml() method to generate an XML file containing the feature schema created by the sample code.

FDO feature schemas can also be read from an XML file. The FdoFeatureSchemaCollection class supports the FdoXmlDeserializable interface. The sample code shows an FdoFeatureSchemaCollection object calling the ReadXml() method to read a set of feature schemas into memory from an XML file. The code shows the desired schema being retrieved from the collection and applied to the data store.

The XML format used by FDO is a subset of the Geography Markup Language (GML) standardized by the Open GIS Consortium (OGC). One thing shown in the sample code is a round-trip conversion from FDO feature schema to GML schema back to FDO feature schema. To accomplish this round-trip, the ReadXml() method supports a superset of the GML that is written by the WriteXml() method.

The following table specifies the mapping of FDO feature schema elements to GML elements and attributes. This mapping is sufficient to understand the XML file generated from the schema defined by the sample code. It also provides a guide for writing a GML schema file by hand. This file can then be read in and applied to a data store. For more information, see Example: Creating a Schema Read In from an XML File (page 88).

Another form of round-trip translation would be from a GML schema produced by another vendor's tool to an FDO feature schema, and then back to a GML schema. However, the resemblance the of resulting GML schema to the original

GML schema might vary from only roughly equivalent to being exactly the same.

Map FDO Element to GML Schema Fragment

FDO Element	GML Schema Fragment

FeatureSchema	`<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"`
	`  targetNamespace="http://<customer_url>/<FeatureSchem`
	`aName>"`
	`  xmlns:fdo="http://fdo.osgeo.org/isd/schema"`
	`  xmlns:gml="http://www.opengis.net/gml"`
	`  xmlns:<FeatureSchemaName>="http://<customer_url>/<Fea`
	`tureSchemaName>"`
	`  elementFormDefault="qualified"`
	`  attributeFormDefault="unqualified"`
	`>`
	`  { see <MetaData> }`
	`  {  optional xs:import element to enable schema validation`
	`  <xs:import namespace="http://fdo.osgeo.org/schema"`
	`schemaLocation="<FDO SDK Install Loca`
	`tion>/docs/XmlSchema/FdoDocument.xsd"/>`
	`  }`
	`  { <one xs:element and/or xs:complexType per class> }`
	`</xs:schema>`

ClassDefinition (with identity properties)	`<xs:element name="<className>"`
	`  type="<className>Type"`
	`  abstract="<true | false>"`
	`  substitutionGroup="gml:_Feature"`
	`>`
	`  <xs:key name="<className>Key">`
	`    <xs:selector xpath=".//<className>"/>`
	`    <xs:field xpath="<identityProperty1Name>"/>`
	`    <xs:field xpath="..."/>`
	`    <xs:field xpath="<identityProperty<n>Name>"`
	`  </xs:key>`
	`</xs:element>`

<table>
<tr><th>FDO Element</th><th>GML Schema Fragment</th></tr>
<tr><td>FeatureClass</td><td>

```
<xs:element ...see ClassDefinition (with identity proper
ties)...</xs:element>
<xs:complexType name="<className>Type"
  abstract="<true | false>"/>
  { see FeatureClass.GeometryProperty }
>
  { see <MetaData> }
  <xs:complexContent>
    <xs:extension base="{baseClass} ?
      {baseClass.schema.name}:{baseClass.name} :
      'gml:AbstractFeatureType' "
    >
      <xs:sequence>
        { list of properties; see DataProperty, Geomet
ricProperty }
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

</td></tr>
<tr><td>FeatureClass. GeometryProperty</td><td>

```
<!-- these attributes belong to the xs:complexType element
 -->
fdo:geometryName="<geometryPropertyName>"
fdo:geometricTypes="<list of FdoGeometricTypes>"
fdo:geometryReadOnly="<true | false>"
fdo:hasMeasure="<true | false>"
fdo:hasElevation="<true | false>"
fdo:srsName="<spatialContextName>"/>
```

</td></tr>
</table>

<table>
<tr><th>FDO Element</th><th>GML Schema Fragment</th></tr>
<tr><td>DataProperty (decimal or string)</td><td><pre><!--
  minOccurs attribute generated only if value is 1
  default attribute generated only if a default value exists
  fdo:readOnly attribute generated only if value is true
-->
<xs:element name=”<propertyName>”
  minOccurs=”{isNullable ? 0 : 1}”
  default=”<defaultValue>”
  fdo:readOnly=”<true | false>”
>
  { see <MetaData> }
  <xs:simpleType>
    { see DataType String or DataType Decimal }
  </xs:simpleType>
</xs:element></pre></td></tr>
<tr><td>DataProperty (other type)</td><td><pre><xs:element name=”<propertyName>”
  type=”<datatype>”
  minOccurs=”{isNullable ? 0 : 1}”
  default=”<defaultValue>”
  fdo:readOnly=”<true | false>”
>
  { see <MetaData> }
</xs:element></pre></td></tr>
<tr><td>DataType String</td><td><pre><xs:restriction base=”xs:string”>
  <xs:maxLength value=”<length>”/>
</xs:restriction></pre></td></tr>
<tr><td>DataType Decimal</td><td><pre><xs:restriction base=”xs:decimal”>
  <xs:totalDigits value=”<precision>”/>
  <xs:fractionDigits value=”<scale>”/>
</xs:restriction></pre></td></tr>
</table>

FDO Element	GML Schema Fragment

GeometricProperty (not a defining FeatureClass GeometryProperty)	`<xs:element name="<propertyName>"`
	`  type="gml:AbstractGeometryType"`
	`  fdo:geometryName="<propertyName>"`
	`  fdo:geometricTypes="<list of FdoGeometricTypes>"`
	`  fdo:geometryReadOnly="<true | false>"`
	`  fdo:hasMeasure="<true | false>"`
	`  fdo:hasElevation="<true | false>"`
	`  fdo:srsName="<spatialContextName>"/>`
	`>`
	`  { see <MetaData> }`
	`</xs:element>`

FDO Element	GML Schema Fragment

MetaData	`<!-- the pattern referenced in the xs:schema element for FeatureSchema-->`
	`<xs:annotation>`
	`  <xs:documentation>{description arg to static FdoFeatureSchema::Create()}</xs:documentation>`
	`</xs:annotation>`
	`<!-- the pattern referenced in the xs:element element for DataProperty -->`
	`<xs:annotation>`
	`  <xs:documentation>{description arg to static FdoDataPropertyDefinition::Create()}</xs:documentation>`
	`</xs:annotation>`
	`<!--`
	`  the pattern referenced in the xs:element element for a non-feature-defining`
	`  GeometricProperty`
	`-->`
	`<xs:annotation>`
	`  <xs:documentation>{description arg to static FdoGeometricPropertyDefinition::Create()}</xs:documentation>`
	`</xs:annotation>`
	`<!-- the pattern referenced in the xs:complexType element for FeatureClass -->`
	`<xs:annotation>`
	`  <xs:documentation>{description arg to static FdoFeatureClass::Create()}</xs:documentation>`
	`  <xs:appinfo source="<uri>"/>`
	`  <xs:documentation>{description arg to static FdoGeometricPropertyDefinition::Create()}</xs:documentation>`
	`</xs:annotation>`

Map FDO Datatype to GML Type

FDO Datatype	GML Type
Boolean	xs:boolean
Byte	fdo:Byte
DateTime	xs:dateTime

FDO Datatype	GML Type
Double	xs:double
Int16	fdo:Int16
Int32	fdo:Int32
Int64	fdo:Int64
Single	xs:float
BLOB	xs:base64Binary
CLOB	xs:string

Creating and Editing a GML Schema File

The sample in this section illustrates the creation of a GML schema file containing the definition of an FDO feature schema that contains one feature. The name of this file will have the standard XML schema extension name, *.xsd*. This means that it contains only one schema and that the root element is xs:schema. The ReadXml() method will take a filename argument whose extension is either *.xsd* or *.xml*. In the latter case, the file could contain many schema definitions. If it does, each schema is contained in an xs:schema element, and all xs:schema elements are contained in the fdo:DataStore element. If there is only one schema in the *.xml* file, then the fdo:DataStore element is not used, and the root element is xs:schema.

You may want to validate the schema that you create. To do so, you must include the optional xs:import line specified in the GML schema fragment for FeatureSchema.

The sample feature implements a table definition for the Buildings feature in the Open GIS Consortium document 98-046r1. This table definition is expressed in an XML format on page 14 of the document and is reproduced as follows:

```
<ogc-sfsql-table>
  <table-definition>
    <name>buildings</name>
    <column-definition>
      <name>fid</name>
      <type>INTEGER</type>
      <constraint>NOT NULL</constraint>
      <constraint>PRIMARY KEY</constraint>
    </column-definition>
    <column-definition>
      <name>address</name>
      <type>VARCHAR(64)</type>
    </column-definition>
    <column-definition>
      <name>position</name>
      <type>POINT</type>
    </column-definition>
    <column-definition>
      <name>footprint</name>
      <type>POLYGON</type>
    <column-definition>
  </table-definition>
```

Add GML for the FDO Feature Schema

Start with the skeleton GML for an FDO Feature Schema with the <MetaData> reference replaced by the valid pattern:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://<customer_url>/<FeatureSchemaName>"
  xmlns:fdo="http://fdo.osgeo.org/schema"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:<FeatureSchemaName>="http://<customer_url>/<FeatureSchem
aName>"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
>
  <xs:annotation>
    <xs:documentation>
      {description arg to static FdoFeatureSchema::Create()}
    </xs:documentation>
  </xs:annotation>
  { <one xs:element and/or xs:complexType per class> }
</xs:schema>
```

For <customer_url> substitute “fdo_customer”. For <FeatureSchemaName> substitute “OGC980461FS”, and for {description arg ... } substitute “OGC Simple Features Specification for SQL.”

Add GML for an FDO Feature Class

Start with the GML that is already written and add the skeleton for an FDO Feature Class, which includes the skeleton for a class definition with identity properties. The <MetaData> is replaced with the valid pattern.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://fdo_customer/OGC980461FS"
  xmlns:fdo="http://fdo.osgeo.org/schema"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:OGC980461FS="http://fdo_customer/OGC980461FS"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
>
  <xs:annotation>
    <xs:documentation>OGC Simple Features Specification for
      SQL</xs:documentation>
  </xs:annotation>
  <xs:element name="<className>"
    type="<className>Type"
    abstract="<true | false>"
    substitutionGroup="gml:_Feature"
  >
    <xs:key name="<className>Key">
      <xs:selector xpath=".//<className>"/>
      <xs:field xpath="<identityProperty1Name>"/>
    </xs:key>
  </xs:element>
  <xs:complexType name="<className>Type"
    abstract="<true | false>"/>
    fdo:geometryName="<geometryPropertyName>"
    fdo:geometricTypes="<list of FdoGeometricTypes>"
    fdo:geometryReadOnly="<true | false>"
    fdo:hasMeasure="<true | false>"
    fdo:hasElevation="<true | false>"
    fdo:srsName="<spatialContextName>"/>
  >
    <xs:annotation>
      <xs:documentation>{description arg to static
        FdoFeatureClass::Create()}</xs:documentation>
      <xs:appinfo source="<uri>"/>
      <xs:documentation>{description arg to static
        FdoGeometricPropertyDefinition::Create()}
      </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
    <xs:extension base="{baseClass} ?
      {baseClass.schema.name}:{baseClass.name} :
      'gml:AbstractFeatureType' "
```

```
    >
        <xs:sequence>
         { list of properties; see DataProperty, GeometricProperty
  }
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

You can make the following changes:

- For <className> substitute “buildings”.
- Set the value of the xs:element abstract attribute to false.
- For <identityPropertyName> substitute “fid”. A data property whose name is “fid” will be added.
- Set the value of the xs:complexType abstract attribute to false.
- For <geometryPropertyName> substitute “footprint”.
- For <list of FdoGeometricTypes> substitute “surface”.
- Set the values of fdo:geometryReadOnly, fdo:hasMeasure, and fdo:hasElevation to false.
- For <spatialContextName> substitute “SC_0”.
- For {description arg to FdoFeatureClass::Create()} substitute “OGC 98-046r1 buildings”.
- For <uri> substitute “http://fdo.osgeo.org/schema”.
- For {description arg to FdoGeometricPropertyDefinition::Create()} substitute “a polygon defines a building perimeter”.
- This class has no base class so set the value of the xs:extension base attribute to ‘gml:AbstractFeatureType’.

Add GML for Property Definitions

An integer data property whose name is “fid” will be added. This property is already identified as an identity property in the xs:key element. A string data property whose name is “name” and a geometry property whose name is “position” will also be added.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://fdo_customer/OGC980461FS"
  xmlns:fdo="http://fdo.osgeo.org/schema"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:OGC980461FS="http://fdo_customer/OGC980461FS"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
>
  <xs:annotation>
    <xs:documentation>OGC Simple Features Specification for
    SQL</xs:documentation>
  </xs:annotation>
  <xs:element name="buildings"
    type="buildingsType"
    abstract="false"
    substitutionGroup="gml:_Feature"
  >
    <xs:key name="buildingsKey">
      <xs:selector xpath=".//buildings"/>
      <xs:field xpath="fid"/>
    </xs:key>
  </xs:element>
  <xs:complexType name="buildingsType"
    abstract="false"/>
    fdo:geometryName="footprint"
    fdo:geometricTypes="surface"
    fdo:geometryReadOnly="false"
    fdo:hasMeasure="false"
    fdo:hasElevation="alse"
    fdo:srsName="SC_0"/>
  >
    <xs:annotation>
      <xs:documentation>OGC 98-046r1 buildings
      </xs:documentation>
      <xs:appinfo source="http://fdo.osgeo.org/schema"/>
      <xs:documentation>a polygon defines the perimeter of a
        building</xs:documentation>
    </xs:annotation>
    <xs:complexContent>
    <xs:extension base="gml:AbstractFeatureType"
    >
        <xs:sequence>
          <xs:element name="<propertyName>"
```

```
    type=”<datatype>”
    minOccurs=”{isNullable ? 0 : 1}”
    default=”<defaultValue>”
    fdo:readOnly=”<true | false>”
  >
    <xs:annotation>
      <xs:documentation>{description arg to static
        FdoDataPropertyDefinition::Create()}
        </xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name=”<propertyName>”
    minOccurs=”{isNullable ? 0 : 1}”
    default=”<defaultValue>”
    fdo:readOnly=”<true | false>”
  >
    <xs:annotation>
      <xs:documentation>{description arg to static
        FdoDataPropertyDefinition::Create()}
      </xs:documentation>
    </xs:annotation>
    <xs:simpleType>
      <xs:restriction base=”xs:string”>
        <xs:maxLength value=”<length>”/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
  <xs:element name=”<propertyName>”
    ref=”gml:_Geometry”
    fdo:geometryName=”<propertyName>”
    fdo:geometricTypes=”<list of FdoGeometricTypes>”
    fdo:geometryReadOnly=”<true | false>”
    fdo:hasMeasure=”<true | false>”
    fdo:hasElevation=”<true | false>”
    fdo:srsName=”<spatialContextName>”/>
  >
    <xs:annotation>
      <xs:documentation>{description arg to static
        FdoGeometricPropertyDefinition::Create()}
      </xs:documentation>
    </xs:annotation>
  </xs:element>
</xs:sequence>
```

```
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

You can make the following changes:

- For the first data property <propertyName> substitute “fid”.
- For the first data property <dataType> substitute “fdo:int32”.
- Do not include the minOccurs or default attributes because the value of minOccurs is 0, which is the default, and there is no <defaultValue>.
- Set the fdo:readOnly attribute for “fid” to false.
- Set the content for xs:documentation for “fid” to “feature id”.
- For the second data property <propertyName> substitute “address”.
- Do not include the minOccurs or default attributes because the value of minOccurs is 0, which is the default, and there is no <defaultValue>.
- Set the fdo:readOnly attribute for “name” to false.
- Set the content for xs:documentation for “address” to “address of the building”.
- For <length> substitute “64”.
- For the geometry property <propertyName> substitute “position”.
- For <list of FdoGeometricTypes> substitute “point”.
- Set the values of fdo:geometryReadOnly, fdo:hasMeasure, and fdo:hasElevation to false.
- For <spatialContextName> substitute “SC_0”.
- For {description arg to FdoGeometricPropertyDefinition::Create()} substitute “position of the building”.

The Final Result

After all the required substitutions, the GML for the schema containing the Buildings feature is as follows:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://fdo_customer/OGC980461FS"
  xmlns:fdo="http://fdo.osgeo.org/schema"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:OGC980461FS="http://fdo_customer/OGC980461FS"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
>
  <xs:annotation>
    <xs:documentation>OGC Simple Features Specification for
      SQL</xs:documentation>
  </xs:annotation>
  <xs:element name="buildings"
    type="buildingsType"
    abstract="false"
    substitutionGroup="gml:_Feature"
  >
    <xs:key name="buildingsKey">
      <xs:selector xpath=".//buildings"/>
      <xs:field xpath="fid"/>
    </xs:key>
  </xs:element>
  <xs:complexType name="buildingsType"
    abstract="false"/>
    fdo:geometryName="footprint"
    fdo:geometricTypes="surface"
    fdo:geometryReadOnly="false"
    fdo:hasMeasure="false"
    fdo:hasElevation="false"
    fdo:srsName="SC_0"/>
  >
    <xs:annotation>
      <xs:documentation>OGC 98-046r1 buildings
      </xs:documentation>
      <xs:appinfo source="http://fdo.osgeo.org/schema"/>
      <xs:documentation>a polygon defines the perimeter of a
        building</xs:documentation>
    </xs:annotation>
    <xs:complexContent>
    <xs:extension base="gml:AbstractFeatureType"
    >
        <xs:sequence>
          <xs:element name="fid"
```

```
            type="fdo:int32"
            fdo:readOnly="false"
          >
            <xs:annotation>
              <xs:documentation>feature id
              </xs:documentation>
            </xs:annotation>
          </xs:element>
          <xs:element name="address"
            fdo:readOnly="false"
          >
            <xs:annotation>
              <xs:documentation>address of the building
              </xs:documentation>
            </xs:annotation>
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:maxLength value="64"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:element>
          <xs:element name="position"
            ref="gml:_Geometry"
            fdo:geometryName="position"
            fdo:geometricTypes="point"
            fdo:geometryReadOnly="false"
            fdo:hasMeasure="false"
            fdo:hasElevation="false"
            fdo:srsName="SC_0"/>
          >
            <xs:annotation>
              <xs:documentation>position of the building</xs:docu
mentation>
            </xs:annotation>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

Schema Management Examples

Example: Creating a Feature Schema

The following sample code creates an FdoFeatureSchema object called "SampleFeatureSchema." The schema contains one class, which has three properties. The class and its properties conform to the table definition for the Lakes feature in the Open GIS Consortium document 98-046r1. This table definition is expressed in an XML format on page 10 of the document and is reproduced as follows:

```
<ogc-sfsql-table>
  <table-definition>
    <name>lakes</name>
    <column-definition>
      <name>fid</name>
      <type>INTEGER</type>
      <constgraint>NOT NULL</constraint>
      <constraint>PRIMARY KEY<constraint>
    </column-definition>
    <column-definition>
      <name>name</name>
      <type>VARCHAR(64)</type>
    </column-definition>
    <column-definition>
      <name>shore</name>
      <type>POLYGON</type>
    </column-definition>
  </table-definition>
```

The table definition whose name is "lakes" is mapped to an FdoFeatureClass object called "SampleFeatureClass." The column definition whose name is "fid" is mapped to an FdoDataPropertyDefinition object called "SampleIdentityDataProperty." The column definition whose name is "name" is mapped to an FdoDataPropertyDefinition object called "SampleNameDataProperty." The column definition whose name is "shore" is mapped to an FdoGeometricPropertyDefinition object called "SampleGeometricProperty."

```
// Create the ApplySchema command
GisPtr<FdoIApplySchema> sampleApplySchema;
sampleApplySchema = (FdoIApplySchema *)
  connection->CreateCommand(FdoCommandType_ApplySchema);
// Create the feature schema
GisPtr<FdoFeatureSchema> sampleFeatureSchema;
sampleFeatureSchema = FdoFeatureSchema::Create(L"SampleFea
tureSchema", L"Sample Feature Schema Description");
// get a pointer to the feature schema's class collection
// this object is used to add classes to the schema
GisPtr<FdoClassCollection> sampleClassCollection;
sampleClassCollection = sampleFeatureSchema->GetClasses();
// create a feature class, i.e., a class containing a geometric
// property set some class level properties
GisPtr<FdoFeatureClass> sampleFeatureClass;
sampleFeatureClass = FdoFeatureClass::Create(L"SampleFeatureClass",
 L"Sample Feature Class Description");
sampleFeatureClass->SetIsAbstract(false);
// get a pointer to the feature class's property collection
// this pointer is used to add data and other properties to the
class
GisPtr<FdoPropertyDefinitionCollection> sampleFeatureClassProper
ties;
sampleFeatureClassProperties = sampleFeatureClass->GetProperties();
// get a pointer to the feature schema's class collection
// this object is used to add classes to the schema
GisPtr<FdoClassCollection> sampleClassCollection;
sampleClassCollection = sampleFeatureSchema->GetClasses();
// get a pointer to the feature class's identity property collec
tion
// this property is used to add identity properties to the feature
// class
GisPtr<FdoDataPropertyDefinitionCollection> sampleFeatureClassIden
tityProperties;
sampleFeatureClassIdentityProperties = sampleFeatureClass->GetIden
tityProperties();
// create a data property that is of type Int32 and identifies
// the feature uniquely
GisPtr<FdoDataPropertyDefinition> sampleIdentityDataProperty;
sampleIdentityDataProperty = FdoDataPropertyDefinition::Cre
ate(L"SampleIdentityDataProperty", L"Sample Identity Data Property
 Description");
sampleIdentityDataProperty->SetDataType(FdoDataType_Int32);
```

```
sampleIdentityDataProperty->SetReadOnly(false);
sampleIdentityDataProperty->SetNullable(false);
sampleIdentityDataProperty->SetIsAutoGenerated(false);
// add the identity property to the sampleFeatureClass
sampleFeatureClassProperties->Add(sampleIdentityDataProperty);
sampleFeatureClassIdentityProperties->Add(sampleIdentityDataProp
erty);
// create a data property that is of type String and names the
// feature
GisPtr<FdoDataPropertyDefinition> sampleNameDataProperty;
sampleNameDataProperty = FdoDataPropertyDefinition::Create(L"Sam
pleNameDataProperty", L"Sample Name Data Property Description");
sampleNameDataProperty->SetDataType(FdoDataType_String);
sampleNameDataProperty->SetLength(64);
sampleNameDataProperty->SetReadOnly(false);
sampleNameDataProperty->SetNullable(false);
sampleNameDataProperty->SetIsAutoGenerated(false);
// add the name property to the sampleFeatureClass
sampleFeatureClassProperties->Add(sampleNameDataProperty);
// create a geometric property
GisPtr<FdoGeometricPropertyDefinition> sampleGeometricProperty;
sampleGeometricProperty = FdoGeometricPropertyDefinition::Cre
ate(L"SampleGeometricProperty", L"Sample Geometric Property Descrip
tion");
sampleGeometricProperty->SetGeometryTypes(FdoGeometricType_Sur
face);
sampleGeometricProperty->SetReadOnly(false);
sampleGeometricProperty->SetHasMeasure(false);
sampleGeometricProperty->SetHasElevation(false);
// add the geometric property to the sampleFeatureClass
sampleFeatureClassProperties->Add(sampleGeometricProperty);
// identify it as a geometry property
sampleFeatureClass->SetGeometryProperty(sampleGeometricProperty);
// add the feature class to the schema
sampleClassCollection->Add(sampleFeatureClass);
// point the ApplySchema command at the newly created feature
// schema and execute
sampleApplySchema->SetFeatureSchema(sampleFeatureSchema);
sampleApplySchema->Execute();
```

Example: Describing a Schema and Writing It to an XML File

The following sample code demonstrates describing a schema and writing it to an XML file:

```
// create the DescribeSchema command
GisPtr<FdoIDescribeSchema> sampleDescribeSchema;
sampleDescribeSchema = (FdoIDescribeSchema *)
  connection->CreateCommand(FdoCommandType_DescribeSchema);
// executing the DescribeSchema command returns a feature
// schema collection that is, the set of feature schema which
// reside in the DataStore
GisPtr<FdoFeatureSchemaCollection> sampleFeatureSchemaCollection;
sampleFeatureSchemaCollection = sampleDescribeSchema->Execute();
// find the target feature schema in the collection, write it
// to an xml file, and clear the collection
sampleFeatureSchema = sampleFeatureSchemaCollection->FindItem(L"Sam
pleFeatureSchema");
sampleFeatureSchema->WriteXml(L"SampleFeatureSchema.xml");
sampleFeatureSchemaCollection->Clear();
```

Example: Destroying a Schema

The following sample code demonstrates destroying a schema:

```
// create the DestroySchema command
GisPtr<FdoIDestroySchema> sampleDestroySchema;
sampleDestroySchema = (FdoIDestroySchema *)
  connection->CreateCommand(FdoCommandType_DestroySchema);
// destroy the schema
sampleDestroySchema->SetSchemaName(L"SampleFeatureSchema");
sampleDestroySchema->Execute();
```

Example: Creating a Schema Read In from an XML File

The following sample code demonstrates creating a schema read in from an XML file:

```
sampleFeatureSchemaCollection->ReadXml(L"SampleFeatureSchema.xml");
sampleFeatureSchema = sampleFeatureSchemaCollection->FindItem(L"Sam
pleFeatureSchema");
sampleApplySchema->SetFeatureSchema(sampleFeatureSchema);
sampleApplySchema->Execute();
sampleFeatureSchemaCollection->Clear();
```

SampleFeatureSchema.xml

The following sample XML schema is the contents of the file written out by the WriteXml method belonging to the FdoFeatureSchema class object that was created in the preceding sample code:

```
<?xml version="1.0" encoding="UTF-8" ?>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://fdo_customer/SampleFeatureSchema"
    xmlns:fdo="http://fdo.osgeo.org/schema"
    xmlns:gml="http://www.opengis.net/gml"
    xmlns:SampleFeatureSchema="http://fdo_customer/
      SampleFeatureSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
  <xs:annotation>
    <xs:documentation>Sample Feature Schema Description
      </xs:documentation>
    <xs:appinfo source="http://fdo.osgeo.org/schema" />
  </xs:annotation>
  <xs:element name="SampleFeatureClass"
    type="SampleFeatureSchema:SampleFeatureClassType"
    abstract="false" substitutionGroup="gml:_Feature">
    <xs:key name="SampleFeatureClassKey">
      <xs:selector xpath=".//SampleFeatureClass" />
      <xs:field xpath="SampleIdentityDataProperty" />
    </xs:key>
  </xs:element>
  <xs:complexType name="SampleFeatureClassType"
    abstract="false"
    fdo:geometryName="SampleGeometricProperty"
    fdo:hasMeasure="false"
    fdo:hasElevation="false"
    fdo:srsName="SC_0"
    fdo:geometricTypes="surface">
    <xs:annotation>
      <xs:documentation>Sample Feature Class Description
      </xs:documentation>
      <xs:appinfo source="http://fdo.osgeo.org/schema" />
      <xs:documentation>Sample Geometric Property Descrip
tion</xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="gml:AbstractFeatureType">
        <xs:sequence>
          <xs:element name="SampleIdentityDataProperty"
            default=""
            type="fdo:int32">
            <xs:annotation>
```

```
              <xs:documentation>
                Sample Identity Data Property Description
              </xs:documentation>
            </xs:annotation>
          </xs:element>
          <xs:element name="SampleNameDataProperty"
            default="">
            <xs:annotation>
              <xs:documentation>
                Sample Name Data Property Description
              </xs:documentation>
            </xs:annotation>
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:maxLength value="64" />
              </xs:restriction>
            </xs:simpleType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

8

Data Maintenance

This chapter provides information about using the FDO API to maintain data.

In this chapter

Data Maintenance Operations

The primary operations associated with data maintenance are:

- Inserting
- Updating
- Deleting
- Transactions
- Locking

NOTE Discussion of Transactions and Locking is deferred to a future release of this document.

Inserting Values

Preconditions

In a previous chapter, we created a feature schema and added a feature class to it. The feature class had three properties: an integer data property, a string data property, and a geometric property. We applied this feature schema to the data store. We are now ready to create feature data objects, which are instances of the feature class, and insert them into the data store.

Property Values in General

We can now create feature data objects, which are instances of the feature class, by defining a set of property values corresponding to the properties defined for the class and then inserting them into the data store.

An FDO class correspondends roughly to a table definition in a relational database and a property of a class corresponds roughly to a column definition in a table. Adding the property values corresponds roughly to adding a row in the table.

The main distinction between a data value or geometry value and a property value is the order in which they are created. A data value or geometry value object is created first and is then used to create a property value object. The property value object is then added to the value collection object belonging to the Insert command object. Then, the command is executed.

An insert operation consists of the following steps:

1 Create the insert command object (type FdoIInsert); this object can be reused for multiple insert operations.

2 Point the insert command object at the feature class to which you are adding values (call the SetFeatureClassName(<className>) method).

3 From the insert command object, obtain a pointer using the GetPropertyValues() method to a value collection object (type FdoPropertyValueCollection). You will add property values to the insert command object by adding values to the collection object.

4 Create a data value (type FdoDataValue) or geometry value (type FdoGeometryValue) object. Creating the data value is straightforward; you pass the string or integer value to a static Create() method. Creating the geometry value is described in Geometry Property Values (page 96).

5 Create a property value (type FdoPropertyValue) object, which involves passing the data value or geometry value object as an argument to a static Create() method.

6 Add the property value object to the value collection object.

7 Execute the Insert command.

Data Property Values

A data value object contains data whose type is one of the following:

- Boolean
- Byte
- DateTime
- Decimal
- Double
- Int16
- Int32
- Int64
- Single (another floating point type)
- String

- Binary large object (BLOB)
- Character large object (CLOB)

The data value object is added to the data property value object. The data property value object is added to the property value collection belonging to the Insert command.

Geometry Property Values

A geometry property value object contains a geometry in the form of a byte array. A geometry can be relatively simple, for example, a point (a single pair of ordinates), or quite complex, for example, a polygon (one or more arrays of ordinates). In the latter case, a number of geometry objects are created and then combined together to form the target geometry. Finally, the target geometry is converted to a byte array and incorporated into the geometry property value object.

Creating a geometry value object consists of the following steps:

1. Create a geometry value object (type FdoGeometryValue) using a static Create() method.
2. Create a geometry factory object (type GisAgfGeometryFactory) using a static GetInstance() method. This object is used to create the geometry object or objects which comprise the target geometry.
3. Create the required geometry object or objects using the appropriate Create<geometry> method() belonging to the geometry factory object.
4. Use the geometry factory object to convert the target geometry object to a byte array.
5. Incorporate the byte array into the geometry property value object.

Example: Inserting an Integer, a String, and a Geometry Value

The following sample code shows how to insert an integer, a string, and a geometry value:

```
// create the insert command
GisPtr<FdoIInsert> sampleInsert;
sampleInsert = (FdoIInsert *)
    connection->CreateCommand(FdoCommandType_Insert);
// index returned by the operation which adds a value to the value
// collection
GisInt32 valueCollectionIndex = 0;
// point the Insert command to the target class
// use a fully qualified class name
// whose format is <schemaName>:<className>
sampleInsert-> SetFeatureClassName(L"SampleFeatureSchema:SampleFea
tureClass");
// get the pointer to the value collection used to add properties
// to the Insert command
GisPtr<FdoPropertyValueCollection> samplePropertyValues;
samplePropertyValues = sampleInsert->GetPropertyValues();
// create an FdoDataValue for the identity property value
GisPtr<FdoDataValue> sampleIdentityDataValue;
sampleIdentityDataValue = FdoDataValue::Create(101);
// add the FdoDataValue to the identity property value
GisPtr<FdoPropertyValue> sampleIdentityPropertyValue;
sampleIdentityPropertyValue =
    FdoPropertyValue::Create(L"SampleIdentityDataProperty",
    sampleIdentityDataValue);
// add the identity property value to the value collection
valueCollectionIndex =
    samplePropertyValues->Add(sampleIdentityPropertyValue);
// create an FdoDataValue for the name property value
GisPtr<FdoDataValue> sampleNameDataValue;
sampleNameDataValue = FdoDataValue::Create(L"Blue Lake");
// add the FdoDataValue to the name property value
GisPtr<FdoPropertyValue> sampleNamePropertyValue;
sampleNamePropertyValue =
    FdoPropertyValue::Create(L"SampleNameDataProperty",
    sampleNameDataValue);
// add the name property value to the value collection
valueCollectionIndex =
    samplePropertyValues->Add(sampleNamePropertyValue);
// create an FdoGeometryValue for the geometry property value
// this polygon represents a lake which has an island
// the outer shoreline of the lake is defined as a linear ring
// the shoreline of the island is defined as a linear ring
// the outer shoreline is the external boundary of the polygon
```

```
// the island shoreline is an internal linear ring
// a polygon geometry can have zero or more internal rings
GisPtr<FdoGeometryValue> sampleGeometryValue;
sampleGeometryValue = FdoGeometryValue::Create();
// create an instance of a geometry factory used to create the
// geometry objects
GisPtr<GisAgfGeometryFactory> sampleGeometryFactory;
sampleGeometryFactory = GisAgfGeometryFactory::GetInstance();
// define the external boundary of the polygon, the shoreline of
// Blue Lake
GisPtr<GisILinearRing> exteriorRingBlueLake;
GisInt32 numBlueLakeShorelineOrdinates = 10;
double blueLakeExteriorRingOrdinates[] = {52.0, 18.0, 66.0, 23.0,
    73.0, 9.0, 48.0, 6.0, 52.0, 18.0};
exteriorRingBlueLake = sampleGeometryFactory->CreateLinearRing(
    GisDimensionality_XY, numBlueLakeShorelineOrdinates,
    blueLakeExteriorRingOrdinates);
// define the shoreline of Goose Island which is on Blue Lake
// this is the sole member of the list of interior rings
GisPtr<GisILinearRing> linearRingGooseIsland;
GisInt32 numGooseIslandShorelineOrdinates = 10;
double gooseIslandLinearRingOrdinates[] = {59.0, 18.0, 67.0, 18.0,
    67.0, 13.0, 59.0, 13.0, 59.0, 18.0};
linearRingGooseIsland = sampleGeometryFactory->CreateLinearRing(
    GisDimensionality_XY, numGooseIslandShorelineOrdinates,
    gooseIslandLinearRingOrdinates);
// add the Goose Island linear ring to the list of interior rings
GisPtr<GisLinearRingCollection> interiorRingsBlueLake;
interiorRingsBlueLake = GisLinearRingCollection::Create();
interiorRingsBlueLake->Add(linearRingGooseIsland);
// create the Blue Lake polygon
GisPtr<GisIPolygon> blueLake;
blueLake =
    sampleGeometryFactory->CreatePolygon(exteriorRingBlueLake,
    interiorRingsBlueLake);
// convert the Blue Lake polygon into a byte array
// and set the geometry value to this byte array
GisByteArray * geometryByteArray =
    sampleGeometryFactory->GetAgf(blueLake);
sampleGeometryValue->SetGeometry(geometryByteArray);
// add the Blue Lake FdoGeometryValue to the geometry property
value
GisPtr<FdoPropertyValue> sampleGeometryPropertyValue;
```

```
sampleGeometryPropertyValue =
    FdoPropertyValue::Create(L"SampleGeometryProperty",
    sampleGeometryValue);
// add the geometry property value to the value collection
valueCollectionIndex =
    samplePropertyValues->Add(sampleGeometryPropertyValue);
// do the insertion
// the command returns an FdoIFeatureReader
GisPtr<FdoIFeatureReader sampleFeatureReader;
sampleFeatureReader = sampleInsert->Execute();
```

Updating Values

After inserting (see Inserting Values (page 94)), you can update the values. The update operation involves identifying a feature class ("table"), a feature class object ("row"), and an object property ("column in a row") to be changed, and supplying a new value for the object property to replace the old.

First, create an FdoIUpdate command object and use the command object's SetFeatureClassName() method to identify the feature class. Then, create a filter to identity the feature class object whose properties we want to update, and use the command object's SetFilter() method to attach the command to it. Filters are discussed in Filter and Expression Languages (page 115).

One of the data properties in the example SampleFeatureClass class definition is an identity property, whose name is "SampleIdentityDataProperty" and whose type is fdo:Int32. This means that its value uniquely identifies the feature class object, that is, the "row". Use the name of the identity property in the filter. In the Insert operation, the value of the identity property was set to be '101'. The value of the filter that is needed is "(SampleIdentityDataProperty = 101)".

Finally, create a property value, which contains the new value, attach it to the command object, and then execute the command.

Example: Updating Property Values

The following is an example of updating property values:

```
GisPtr<FdoIUpdate> sampleUpdate;
sampleUpdate =
    (FdoIUpdate *)connection->CreateCommand(FdoCommandType_Update);
GisInt32 numUpdated = 0;
// point the Update command at the target feature class
// use a fully qualified class name
// whose format is <schemaName>:<className>
sampleUpdate-> SetFeatureClassName(L"SampleFeatureSchema:SampleFea
tureClass");
// set the filter to identify which set of properties to update
sampleUpdate->SetFilter(L"( SampleIdentityDataProperty = 101 )");
// get the pointer to the value collection used to add properties
// to the Update command
// we are reusing the samplePropertyValues object that we used
// for the insert operation
samplePropertyValues = sampleUpdate->GetPropertyValues();
// create an FdoDataValue for the name property value
GisPtr<FdoDataValue> sampleNameDataValue;
sampleNameDataValue = FdoDataValue::Create(L"Green Lake");
// set the name and value of the property value
sampleNamePropertyValue->SetName(L"SampleNameDataProperty");
sampleNamePropertyValue->SetValue(sampleNameDataValue);
// add the name property value to the property value collection
// owned by the Update command
samplePropertyValues->Add(sampleNamePropertyValue);
// execute the command
numUpdated = sampleUpdate->Execute();
```

Deleting Values

In addition to inserting (see Inserting Values (page 94)) and updating (see Updating Values (page 99)) values, you can delete the values. The deletion operation involves identifying a feature class ("table") whose feature class objects ("rows") are to be deleted.

First, create an FdoIDelete command object and use the command object's SetFeatureClassName() method to identify the feature class. Then, create a filter to identity the feature class objects that you want to delete, and use the command object's SetFilter() method to attach the filter to it. You can use the same filter that was specified in the preceding section, Updating Values (page 99). Finally, execute the command.

Example: Deleting Property Values

```
GisPtr<FdoIDelete> sampleDelete;
sampleDelete =
    (FdoIDelete *)connection->CreateCommand(FdoCommandType_Delete);
GisInt32 numDeleted = 0;
sampleDelete->
SetFeatureClassName(L"SampleFeatureSchema:SampleFeatureClass");
sampleDelete->SetFilter(L"( SampleIdentityDataProperty = 101 )");
numDeleted = sampleDelete->Execute();
```

Related Class Topics

The following classes are used in the preceding Data Maintenance examples:

- FdoIInsert
- FdoPropertyValueCollection
- FdoDataValue
- FdoPropertyValue
- FdoGeometryValue
- GisAgfGeometryFactory
- GisILinearRing
- GisLinearRingCollection
- GisIPolygon
- GisByteArray
- FdoIDelete
- FdoIUpdate

For more information, see *FDO API Reference Help*.

9

Performing Queries

This chapter describes how to create and perform queries. In the FDO API, you can use queries to retrieve specific features from a data store.

In this chapter

Creating a Query

You create and perform queries using the FdoISelect class, which is a member of the Feature sub-package of the Commands package. Queries are used to retrieve features from the data store, and are executed against one class at a time. The class is specified using the SetFeatureClassName() method in FdoIFeatureCommand. The SetFeatureClassName can be used with feature and non-feature classes.

FdoISelect supports the use of filters to limit the scope of features returned by the command. This is done through one of the SetFilter methods available in the FdoIFeatureCommand class. The filter is similar to the SQL WHERE clause, which specifies the search conditions that are applied to one or more class properties.

Search conditions include spatial and non-spatial conditions. Non-spatial queries create a condition against a data property, such as an integer or string. Basic comparisons (=, <, >, >=, <=, !=), pattern matching (like), and 'In' comparisons can be specified. Spatial queries create a spatial condition against a geometry property. Spatial conditions are enumerated in FdoSpatialCondition and FdoDistanceCondition.

The feature reader (FdoIFeatureReader) is used to retrieve the results of a query for feature and non-feature classes. To retrieve the features from the reader, iterate through the reader using the FdoIFeatureReader.ReadNext method().

Query Example

In the Data Maintenance chapter, we created an instance of the FdoFeatureClass SampleFeatureClass and assigned values to its integer, string, and geometry properties (see Example: Inserting an Integer, a String, and a Geometry Value (page 96)). The sample code in the following query example selects this instance and retrieves the values of its properties. Specifically, the sample code does the following:

1. Creates the select command, and
2. Points the select command at the target FdoFeatureClass SampleFeatureClass, and
3. Creates a filter to identify which instance of SampleFeatureClass to select, and
4. Points the select command at the filter, and

5 Executes the command, which returns an FdoIFeatureReader object, and

6 Loops through the feature reader object, which contains one or more query results depending on the filter arguments. In the sample code provided, there is only one result.

7 Finally, the code extracts the property values from each query result.

```
// we have one FdoFeatureClass object in the DataStore
// create a query that returns this object
// create the select command
GisPtr<FdoISelect> sampleSelect;
sampleSelect = (FdoISelect *)
  connection->CreateCommand(FdoCommandType_Select);
// point the select command at the target FdoFeatureClass
// SampleFeatureClass
sampleSelect->SetFeatureClassName(L"SampleFeatureClass");
// create the filter by
// 1. creating an FdoIdentifier object containing the name of
//    the identity property
GisPtr<FdoIdentifier> queryPropertyName;
queryPropertyName =
  FdoIdentifier::Create(L"SampleIdentityDataProperty");
// 2. creating an FdoDataValue object containing the value of the
//    identity property
GisPtr<FdoDataValue> queryPropertyValue;
queryPropertyValue = FdoDataValue::Create(101);
// 3. calling FdoComparisonCondition::Create() passing in the
//    the queryPropertyName, an enumeration constant signifying an
//    equals comparison operation, and the queryPropertyValue
GisPtr<FdoFilter> filter;
filter = FdoComparisonCondition::Create(queryPropertyName,
  FdoComparisonOperations_EqualTo, queryPropertyValue);
// point the select command at the filter
sampleSelect->SetFilter(filter);
// execute the select command
GisPtr<FdoIFeatureReader> queryResults;
queryResults = sampleSelect->Execute();
// declare variables needed to capture query results
GisPtr<FdoClassDefinition> classDef;
GisPtr<FdoPropertyDefinitionCollection> properties;
GisInt32 numProperties = 0;
FdoPropertyDefinition * propertyDef;
FdoPropertyType propertyType;
FdoDataType dataType;
FdoDataPropertyDefinition * dataPropertyDef;
GisString * propertyName = NULL;
GisPtr<GisByteArray> byteArray;
GisIGeometry * geometry = NULL;
GisGeometryType geometryType = GisGeometryType_None;
GisIPolygon * polygon = NULL;
```

```
GisILinearRing * exteriorRing = NULL;
GisILinearRing * interiorRing = NULL;
GisIDirectPosition * position = NULL;
GisInt32 dimensionality = GisDimensionality_XY;
GisInt32 numPositions = 0;
GisInt32 numInteriorRings = 0;
// loop through the query results
while (queryResults->ReadNext()) {
  // get the feature class object and its properties
  classDef = queryResults->GetClassDefinition();
  properties = classDef->GetProperties();
  // loop through the properties
  numProperties = properties->GetCount();
  for(int i = 0; i < numProperties; i++) {
    propertyDef = properties->GetItem(i);
    // get the property name and property type
    propertyName = propertyDef->GetName();
    propertyType = propertyDef->GetPropertyType();
    switch (propertyType) {
      // it's a data property
      case FdoPropertyType_DataProperty:
        dataPropertyDef =
          dynamic_cast<FdoDataPropertyDefinition *>
          (propertyDef);
        dataType = dataPropertyDef->GetDataType();
        switch (dataType) {
          case FdoDataType_Boolean:
            break;
          case FdoDataType_Int32:
            break;
          case FdoDataType_String:
            break;
          default:
        }
        break;
      // it's a geometric property
      // convert the byte array to a geometry
      // and determine the derived type of the geometry
      case FdoPropertyType_GeometricProperty:
        byteArray = queryResults->GetGeometry(propertyName);
        geometry =
          sampleGeometryFactory->CreateGeometryFromAgf
          (byteArray);
```

```
geometryType = geometry->GetDerivedType();
// resolve the derived type into a list of ordinates
switch (geometryType) {
  case GisGeometryType_None:
    break;
  case GisGeometryType_Point:
    break;
  case GisGeometryType_LineString:
    break;
  case GisGeometryType_Polygon:
    polygon = dynamic_cast<GisIPolygon *>(geometry);
    exteriorRing = polygon->GetExteriorRing();
    dimensionality = exteriorRing-
      >GetDimensionality();
    numPositions = exteriorRing->GetCount();
    double X, Y, Z, M;
    for(int i=0; i<numPositions; i++) {
      position = exteriorRing->GetItem(i);
      if (dimensionality & GisDimensionality_Z &&
        dimensionality & GisDimensionality_M) {
        X = position->GetX();
        Y = position->GetY();
        Z = position->GetZ();
        M = position->GetM();
      else if (dimensionality & GisDimensionality_Z
      && !(dimensionality & GisDimensionality_M)) {
        X = position->GetX();
        Y = position->GetY();
        Z = position->GetZ();
      else {
        X = position->GetX();
        Y = position->GetY();
      }
    }
    numInteriorRings = polygon-
      >GetInteriorRingCount();
    for(int i=0; i<numInteriorRings; i++) {
      interiorRing = polygon->GetInteriorRing(i);
      // do same for interior ring as exterior ring
    }
    break;
  case GisGeometryType_MultiPoint:
    break;
```

```
          case GisGeometryType_MultiLineString:
            break;
          case GisGeometryType_MultiPolygon:
            break;
          case GisGeometryType_MultiGeometry:
            break;
          case GisGeometryType_CurveString:
            break;
          case GisGeometryType_CurvePolygon:
            break;
          case GisGeometryType_MultiCurveString:
            break;
          case GisGeometryType_MultiCurvePolygon:
            break;
          default:
        }
        break;
      default:
    }
  }
}
```

Long Transaction Processing

10

This chapter defines long transactions (LT) and long transaction interfaces, and explains how to implement LT processing in your application.

NOTE For this release, the providers that support long transaction processing are Autodesk FDO Provider for Oracle and OSGeo FDO Provider for ArcSDE.

In this chapter

What Is Long Transaction Processing?

A long transaction (LT) is an administration unit that is used to group conditional changes to objects. Depending on the situation, such a unit can contain conditional changes to one or to many objects. Long transactions are used to modify as-built data in the database without permanently changing the as-built data. Long transactions can be used to apply revisions or alternates to an object.

A root long transaction is a long transaction that represents permanent data and that has descendents. Any long transaction has a root long transaction as an ancestor in its long transaction dependency graph. A leaf long transaction does not have descendents.

For more information about Oracle-specific long transaction versions and locking, see Locking and Long Transactions (page 154).

Supported Interfaces

In the current release of FDO, the following long transaction interfaces are supported:

- FDOIActivateLongTransaction
- FDOIDeactivateLongTransaction
- FDOIRollbackLongTransaction
- FDOICommitLongTransaction
- FDOICreateLongTransaction
- FDOIGetLongTransaction

These interfaces are summarized below. For more information about their usage, supported methods, associated enumerations and readers, see the *FDO API Reference Help*.

FDOIActivateLongTransaction

The FdoIActivateLongTransaction interface defines the ActivateLongTransaction command, which activates a long transaction where feature manipulation and locking commands operate on it. Input to the activate long transaction command is the long transaction name. The Execute operation activates the identified long transaction.

FDOIDeactivateLongTransaction

The FdoIDeactivateLongTransaction interface defines the DeactivateLongTransaction command, which deactivates the active long transaction where feature manipulation and locking commands operate on it. If the active long transaction is the root long transaction, then no long transaction will be deactivated.

FDOIRollbackLongTransaction

The FdoIRollbackLongTransaction interface defines the RollbackLongTransaction command, which allows a user to execute rollback operations on a long transaction. Two different rollback operations are available: Full and Partial.

The operation is executed on all data within a long transaction and on all its descendents. The data is removed from the database and all versions involved in the process deleted.

NOTE If the currently active long transaction is the same as the one being committed or rolled back, then, if the commit or rollback succeeds, the provider resets the current active long transaction to be the root long transaction. If it does not succeed, the active long transaction is left alone and current. If the currently active long transaction is not the same as the one being committed or rolled back, then it is not affected.

FDOICommitLongTransaction

The FdoICommitLongTransaction interface defines the CommitLongTransaction command, which allows a user to execute commit operations on a long transaction. Two different commit operations are available: Full and Partial.

The commit operation can be performed on a leaf long transaction only. A long transaction is a leaf long transaction if it does not have descendents.

FDOICreateLongTransaction

The FdoICreateLongTransaction interface defines the CreateLongTransaction command, which creates a long transaction that is based on the currently active long transaction. There is always an active long transaction. If the user has not activated a user-defined long transaction, then the root long transaction is active.

Input to the CreateLongTransaction command includes a name and description for the new long transaction. The long transaction name submitted to the command has to be unique. If it is not unique, an exception is thrown.

FDOIGetLongTransactions

The FdoIGetLongTransactions interface defines the GetLongTransactions command, which allows the user to retrieve long transaction information. If a long transaction name is submitted, the command returns the information for the named long transaction only. If no long transaction name is given, the command retrieves the names of all available long transactions.

For each returned long transaction, the user has the option to retrieve a list of descendents and/or ancestors.

11

Filter and Expression Languages

This chapter discusses the use of filters and filter expressions. You can use filters and expressions to specify to an FDO provider how to identify a subset of the objects in a data store.

In this chapter

Filters

FDO uses filters through its commands (including provider-specific commands) to select certain features and exclude others.

A filter is a construct that an application specifies to an FDO provider to identify a subset of objects of an FDO data store. For example, a filter may be used to identify all Road type features that have 2 lanes and that are within 200 metres of a particular location. Many FDO commands use filter parameters to specify the objects to which the command applies. For example, a select command takes a filter to identify the objects that the application wants to retrieve or a delete command takes a filter to identify the objects that the application wants to delete from the data store.

When a command executes, the filter is evaluated for each feature instance and that instance is included in the scope of the command only if the filter evaluates to True. Filters may be specified either as text or as an expression tree. Feature providers declare their level of support for filters through the filter capabilities metadata. Query builders should configure themselves based on the filter capabilities metadata in order to provide users with a robust user interface. For more information, see What Is an Expression? (page 21).

Expressions

FDO uses expressions through its commands (including provider-specific commands) to specify input values in order to filter features. In general, commands in FDO do not support the SQL command language (the one exception is the optional SQLCommand). However, to facilitate ease of use for application developers, expressions in FDO can be specified using a textual notation that is based syntactically on expressions and SQL WHERE clauses. In FDO, expressions are not intended to work against tables and columns, but against feature classes, properties, and relationships. For example, an expression to select roads with four or more lanes might look like this:

```
Lanes >= 4
```

An expression is a construct that an application can use to build up a filter. In other words, an expression is a clause of a filter or larger expression. For example, “Lanes >=4 and PavementType = 'Asphalt'” takes two expressions and combines them to create a filter.

Filter and Expression Text

In general, commands in FDO do not support the SQL command language (the one exception is the optional SQLCommand). However, to facilitate ease of use for application developers, expressions and filters in FDO can be specified using a textual notation that is based syntactically on expressions and SQL WHERE clauses. The biggest difference between this approach and SQL is that these clauses are not intended to work against tables and columns, but against feature classes, properties, and relationships. For example, a filter to select roads with four or more lanes might look like:

```
Lanes >= 4
```

Similarly, a filter to select all PipeNetworks that have at least one Pipe in the proposed state might look like:

```
Pipes.state = "proposed"
```

Furthermore, a filter to select all existing parcels whose owner contains the text “Smith” might look like:

```
state = "existing" and owner like "%Smith%"
```

Finally, a filter to select all parcels that are either affected or encroached upon by some change might look like:

```
state in ("affected", "encroached")
```

Language Issues

There are a number of language issues to be considered when working with classes in the Filter, Expression, and Geometry packages:

- Provider-specific constraints on text
- Filter grammar
- Expression grammar
- Filter and Expression keywords
- Data types
- Operators
- Special characters

- Geometry value

Provider-Specific Constraints on Filter and Expression Text

Some providers may have reserved words that require special rules when used with filters and expressions. For more information, see Oracle Reserved Words Used with Filter and Expression Text (page 153).

Filter Grammar

The rules for entering filter expressions are described in the following sections using BNF notation. For more information about BNF notation, see http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html.

The FdoFilter::Parse() method supports the following filter grammar:

```
<Filter> ::= '(' Filter ')'
| <LogicalOperator>
| <SearchCondition>
<LogicalOperator> ::= <BinaryLogicalOperator>
| <UnaryLogicalOperator>
<BinaryLogicalOperator> ::=
<Filter> <BinaryLogicalOperations> <Filter>
<SearchCondition> ::= <InCondition>
| <ComparisonCondition>
| <GeometricCondition>
| <NullCondition>
<InCondition> ::= <Identifier> IN '(' ValueExpressionCollection
')'
<ValueExpressionCollection> ::= <ValueExpression>
| <ValueExpressionCollection> ',' <ValueExpression>
<ComparisonCondition> ::=
<Expression> <ComparisonOperations> <Expression>
<GeometricCondition> ::= <DistanceCondition>
| <SpatialCondition>
<DistanceCondition> ::=
<Identifier> <DistanceOperations> <Expression> <distance>
<NullCondition> ::= <Identifier> NULL
<SpatialCondition> ::=
<Identifier> <SpatialOperations> <Expression>
<UnaryLogicalOperator> ::= NOT <Filter>
<BinaryLogicalOperations> ::= AND | OR
<ComparisionOperations> ::=
= // EqualTo (EQ)
<> // NotEqualTo (NE)
> // GreaterThan (GT)
>= // GreaterThanOrEqualTo (GE)
< // LessThan (LT)
<= // LessThanOrEqualTo (LE)
LIKE // Like
<DistanceOperations> ::= BEYOND | WITHINDISTANCE
<distance> ::= DOUBLE | INTEGER
<SpatialOperations> ::= CONTAINS | CROSSES | DISJOINT
| EQUALS | INTERSECTS | OVERLAPS | TOUCHES | WITHIN | COVEREDBY |
 INSIDE
```

Expression Grammar

The FdoExpression::Parse() method supports the following expression grammar:

```
<Expression> ::= '(' Expression ')'
| <UnaryExpression>
| <BinaryExpression>
| <Function>
| <Identifier>
| <ValueExpression>
<BinaryExpression> ::=
<Expression> '+' <Expression>
| <Expression> '-' <Expression>
| <Expression> '*' <Expression>
| <Expression> '/' <Expression>
<DataValue> ::=
TRUE
| FALSE
| DATETIME
| DOUBLE
| INTEGER
| STRING
| BLOB
| CLOB
| NULL
<Function> ::= <Identifier> '(' <ExpressionCollection> ')'
<ExpressionCollection> ::=
| <Expression>
| <ExpressionCollection> ',' <Expression>
<GeometryValue> ::= GEOMFROMTEXT '(' STRING ')'
<Identifier> ::= IDENTIFIER
<ValueExpression> ::= <LiteralValue> | <Parameter>;
<LiteralValue> ::= <GeometryValue> | <DataValue>
<Parameter> ::= PARAMETER | ':'STRING
<UnaryExpression> ::= '-' <Expression>
```

Expression Operator Precedence

The precedence is shown in YACC notation, that is, the highest precedence operators are at the bottom.

```
%left Add Subtract
%left Multiply Divide
%left Negate
```

Filter and Expression Keywords

The following case-insensitive keywords are reserved in the language, that is, they cannot be used as identifier or function names:

```
AND BEYOND COMPARE CONTAINS COVEREDBY CROSSES DATE
DISJOINT DISTANCE EQUALS FALSE GeomFromText IN INSIDE
INTERSECTS LIKE NOT NULL OR OVERLAPS RELATE SPATIAL TIME
TIMESTAMP TOUCHES TRUE WITHIN WITHINDISTANCE
```

Data Types

The available data types are described in this section.

Identifier

An identifier can be any alphanumeric sequence of characters other than a keyword. Identifiers can be enclosed in double quotes to allow special characters and white space. If you need to include a double quote character inside an identifier, double the character, for example "abc""def".

Parameter

Parameters are defined by a colon followed by alphanumeric characters. The FDO filter language extends SQL to allow for a literal string to follow the colon to allow blanks (and other possibilities), for example, :'Enter Name'.

Determine whether parameters are supported by the FDO Provider you are using by checking SupportParameters on the Connection interface.

String

Strings are literal constants enclosed in single quotes. The FDO filter language also supports the special characters (left and right single quotes) that Microsoft Word uses to automatically replace the single quote character typed from the keyboard. If you need to include a single quote character inside a string you can double the character, for example 'aaa''bbb'.

Integer

Integers allow only decimal characters with an optional unary minus sign. Unary plus is not supported.

(-){[0-9]}

Double

Floating point numbers have a decimal point, can be signed (-), and include an optional exponent (e{[0-9]}).

NOTE If an integer is out of the 32-bit precision range, it is converted to floating point.

Examples:

```
-3.4
12345678901234567
1.2e13
```

DateTime

Date and time are parsed using the standard SQL literal strings:

```
DATE 'YYYY-MM-DD'
TIME 'HH:MM:SS[.sss]'
TIMESTAMP 'YYYY-MM-DD HH:MM:SS[.sss]'
```

For example:

```
DATE '1971-12-24'
TIMESTAMP '2003-10-23 11:00:02'
```

NOTE The BLOB and CLOB strings are currently not supported. If you need to support binary input, use parameters.

Operators

The following operators are special characters common to SQL and most programming languages:

BinaryOperations

These binary operations are available:

+ Add (for compatibility with SQL string concatenation may also be defined using “||”)

- Subtract

* Multiply

/ Divide

UnaryOperations

These unary operation are available:

- Negate

Comparison Operations

These comparison operations are available:

= EqualTo (EQ)

<> NotEqualTo (NE)

> GreaterThan (GT)

>= GreaterThanOrEqualTo (GE)

< LessThan (LT)

<= LessThanOrEqualTo (LE)

Operator Precedence

The following precedence is shown from highest to lowest:

Negate NOT

Multiply Divide

Add Subtract

EQ NE GT GE LT LE

AND

OR

Special Character

The following special characters are used in ExpressionCollections and ValueExpressions to define function arguments and IN conditions:

(Left Parenthesis

, Comma

) Right Parenthesis

The Colon (:) is used in defining parameters and the Dot (.) can be included in real numbers and identifiers.

Geometry Value

Geometry values are handled using a function call GeomFromText('AGF Text string'), as is typical in an SQL query.

The Autodesk extension to WKT, referred to as AGF Text, is a superset of WKT (that is, you can enter WKT as valid AGF Text strings). Dimensionality is optional. It can be XY, XYM, XYZ, or XYZM. If it is not specified, it is assumed to be XY. For more information about AGF, see GisAgfGeometryFactory (page 136).

NOTE Extra ordinates are ignored, rather than generating an error during AGF text parsing. For example, in the string "POINT (10 11 12)", the '12' is ignored because the dimensionality is assumed to be XY.

The following is the grammar definition for AGF Text:

<AGF Text> ::= POINT <Dimensionality> <PointEntity>

| LINESTRING <Dimensionality> <LineString>

| POLYGON <Dimensionality> <Polygon>

| CURVESTRING <Dimensionality> <CurveString>

| CURVEPOLYGON <Dimensionality> <CurvePolygon>

| MULTIPOINT <Dimensionality> <MultiPoint>

| MULTILINESTRING <Dimensionality> <MultiLineString>

| MULTIPOLYGON <Dimensionality> <MultiPolygon>

| MULTICURVESTRING <Dimensionality> <MultiCurveString>

```
| MULTICURVEPOLYGON <Dimensionality> <MultiCurvePolygon>
| GEOMETRYCOLLECTION <GeometryCollection>
<PointEntity> ::= '(' <Point> ')'
<LineString> ::= '(' <PointCollection> ')'
<Polygon> ::= '(' <LineStringCollection> ')'
<MultiPoint> ::= '(' <PointCollection> ')'
<MultiLineString> ::= '(' <LineStringCollection> ')'
<MultiPolygon> ::= '(' <PolygonCollection> ')'
<GeometryCollection : '(' <AGF Collection Text> ')'
<CurveString> ::= '(' <Point> '(' <CurveSegmentCollection> ')' ')'
<CurvePolygon> ::= '(' <CurveStringCollection> ')'
<MultiCurveString> ::= '(' <CurveStringCollection> ')'
<MultiCurvePolygon> ::= '(' <CurvePolygonCollection> ')'
<Dimensionality> ::= // default to XY
| XY
| XYZ
| XYM
| XYZM
<Point> ::= DOUBLE DOUBLE
| DOUBLE DOUBLE DOUBLE
| DOUBLE DOUBLE DOUBLE DOUBLE
<PointCollection> ::= <Point>
| <PointCollection ',' <Point>
<LineStringCollection> ::= <LineString>
| <LineStringCollection> ',' <LineString>
<PolygonCollection> ::= <Polygon>
| <PolygonCollection> ',' <Polygon>
<AGF Collection Text> ::= <AGF Text>
| <AGF Collection Text> ',' <AGF Text>
```

<CurveSegment> ::= CIRCULARARCSEGMENT '(' <Point> ',' <Point> ')'

| LINESTRINGSEGMENT '(' <PointCollection> ')'

<CurveSegmentCollection> ::= <CurveSegment>

| <CurveSegmentCollection> ',' <CurveSegment>

<CurveStringCollection> ::= <CurveString>

| <CurveStringCollection> ',' <CurveString>

<CurvePolygonCollection> ::= <CurvePolygon>

| <CurvePolygonCollection> ',' <CurvePolygon>

The only other token type is DOUBLE, representing a double precision floating point values. Integer (non-decimal point) input is converted to DOUBLE in the lexical analyzer.

Examples of the Autodesk extensions include:

POINT XY (10 11) // equivalent to POINT (10 11)

POINT XYZ (10 11 12)

POINT XYM (10 11 1.2)

POINT XYZM (10 11 12 1.2)

GEOMETRYCOLLECTION (POINT xyz (10 11 12),POINT XYM (30 20 1.8), LINESTRING XYZM(1 2 3 4, 3 5 15, 3 20 20))

CURVESTRING (0 0 (LINESTRINGSEGMENT (10 10, 20 20, 30 40))))

CURVESTRING (0 0 (CIRCULARARCSEGMENT (11 11, 12 12), LINESTRINGSEGMENT (10 10, 20 20, 30 40)))

CURVESTRING (0 0 (ARC (11 11, 12 12), LINESTRINGSEGMENT (10 10, 20 20, 30 40)))

CURVESTRING XYZ (0 0 0 (LINESTRINGSEGMENT (10 10 1, 20 20 1, 30 40 1)))

MULTICURVESTRING ((0 0 (LINESTRINGSEGMENT (10 10, 20 20, 30 40))),(0 0 (ARC (11 11, 12 12), LINESTRINGSEGMENT (10 10, 20 20, 30 40))))

CURVEPOLYGON ((0 0 (LINESTRINGSEGMENT (10 10, 10 20, 20 20), ARC (20 15, 10 10))), (0 0 (ARC (11 11, 12 12), LINESTRINGSEGMENT (10 10, 20 20, 40 40, 90 90))))

MULTICURVEPOLYGON (((0 0 (LINESTRINGSEGMENT (10 10, 10 20, 20 20), ARC (20 15, 10 10))), (0 0 (ARC (11 11, 12 12), LINESTRINGSEGMENT (10 10,

20 20, 40 40, 90 90)))),((0 0 (LINESTRINGSEGMENT (10 10, 10 20, 20 20), ARC (20 15, 10 10))), (0 0 (ARC (11 11, 12 12), LINESTRINGSEGMENT (10 10, 20 20, 40 40, 90 90)))))

The Geometry API

12

This chapter describes the GisGeometry API (hereafter called the “Geometry API”) and explains the different Geometry types and formats.

In this chapter

Description of the Geometry API

The Geometry API supports specific Autodesk applications and APIs, including FDO (Feature Data Objects). This API consists of the following:

- Geometry Type package (all through fully encapsulated interfaces)
- Abstract Geometry Factory
- Concrete Geometry Factory for AGF

You can work with the Geometry API in several different ways:

- AGF
- AGF Text
- Abstract Geometry Factory

AGF

Autodesk Geometry Format (AGF) is Autodesk's extended version of the Well Known Binary Format (WKB).

WKB is a memory layout used to store geometry used by GIS applications. This format was created by the OpenGIS organization to allow the efficient exchange of geometry data between different components in a GIS system. Most pieces of the original specification defining the WKB format are in the document, *99-050.pdf*, the OpenGIS Simple feature specification for OLE/COM that can be found at www.opengis.org.

WKB and AGF

The WKB and AGF formats are differ in only a few significant details:

- WKB defines a byte order of the data in every piece of geometry. This is stored as a byte field, which as a result might change the memory alignment from word to byte. In AGF, only one memory alignment type is supported, which is the same alignment type as used by the .NET framework and Windows (encoded using the little-endian byte order format). As a result, this byte flag does not need to be stored.
- WKB is defined as a 2D format only. This is insufficient to represent 3D points, polylines and polygons. In AGF, the dimension flag has been added.

In particular, a flag is included for each geometry piece to indicate whether the geometry is 2D, 3D, or even 4D (storing a measure value as used by dynamic segmentation.

- In AGF, geometry types are included that are not yet covered by any WKB specification.

Basic / Pure Geometry

In this section, the memory layout of each simple geometry type is described. The format is taken from the OGC specification, built on the memory layout of a C++ struct. All arrays have a computable size and are inline; they do not point to a different location in memory. The actual architecture of this format allows streaming of geometry data.

First, the different data types, their size, and memory layout are discussed

```
// int == 4 byte integer in little endian encoding
// float == 4 byte IEEE floating number in little endian encoding
// double == 8 byte IEEE double number in little endian encoding.
// char == 2 byte unicode character in little endian encoding
// GisInt32 == 4 byte integer in little endian encoding
// double == 8 byte IEEE double number in little endian encoding.
// the type of the geometry
enum GeometryType : int
{
None = 0,
Point = 1,
LineString = 2,
Polygon = 3,
MultiPoint = 4,
MultiLineString = 5,
MultiPolygon = 6,
MultiGeometry = 7,
 = 10,
CurvePolygon = 11,
MultiCurveString = 12,
MultiCurvePolygon = 13
}
```

Coordinate Types

This is a bit field, for example, xym == coordinateDimensionality.XY | CoordinateDimensionality.M. The following sequence defines the type of coordinates used for this object:

```
enum CoordinateDimensionality : int
{
XY = 0,
Z = 1,
M = 2
}
```

Basic Geometry

The following sequence establishes the basic pure geometry:

```
struct Geometry
{
int geomType;
CoordinateDimensionality type;
}
```

Defining a Method for Notation

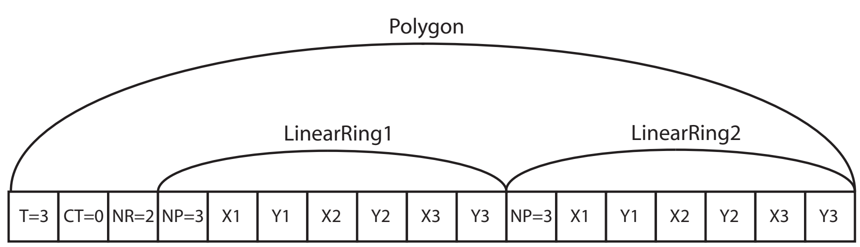
The following sequence defines a method for notation within this specification:

```
// Define a method for notation within this specification
// int PositionSize(geometry)
// {
// if (geometry.type == CoordinateDimensionality.XY |
// CoordinateDimensionality.M ||
// geometry.type == CoordinateDimensionality.XY |
// CoordinateDimensionality.Z)
// return 3;
// if (geometry.type == CoordinateDimensionality.XY |
// CoordinateDimensionality.M | CoordinateDimensionality.Z)
// return 4
// return 2;
// }
struct Point // : Geometry
{
int geomType; // == GeometryType.Point;
CoordinateDimensionality type; // all types allowed
double[] coords; // size = PositionSize(this)
}
struct LineString
{
int geomType;
CoordinateDimensionality type;
int numPts; // >0
double[] coords; // size = numPts* PositionSize(this)
}
struct MultiPoint
{
int geomType;
int numPoints; // > 0
Point[] points; // size = numPoints
}
struct MultiLineString
{
int geomType;
int numLineStrings; // >= 0
LineString[] lineStrings; // size = numLineStrings
}
// building block for polygons, not geometry by itself
struct LinearRing
{
int numPts; // >0
double[] coords; // size = numPts* PositionSize(polygon)
```

```
}
struct Polygon
{
int geomType;
CoordinateDimensionality type;
int numRings; // >= 1 as there has to be at least one ring
LinearRing[] lineStrings; // size = numRings
}
struct MultiPolygon
{
int geomType;
int numPolygons; // >= 0
Polygon[] polygons; // size = numPolygons
}
struct MultiGeometry
{
int geomType;
int numGeom; // >= 0
Geometry[] geometry; // size = numGeom
}
enum CurveElementType : int
{
LineString = 1,
CircularArc = 2
}
struct CurveStringElement
{
int CurveElementType;
}
struct LinearCurveStringElement
{
int CurveElementType;
int length;
double[] coords; // size = this.length * PositionSize (this)
}
struct CircularArcCurveStringElement
{
int CurveElementType; // == CurveElmentType.Arc
double[] coords; // size = 2 * PositionSize(this)
}
struct CurveString
{
int geomType;
```

```
CoordinateDimensionality type; // all types allowed
double[] startPoint; // size = PositionSize(this)
int numElements; // >=0
CurveStringElement[] elements; // size = numElements
}
struct Ring
{
double[] startPoint; // size = PositionSize(this)
int numElements; // >=0
CurveStringElement[] elements; // size = numElements
}
struct MultiCurveString
{
int geomType;
int numCurveStrings; // >= 0
CurveString[] curveStrings; // size = numCurveStrings
}
struct CurvePolygon
{
int geomType; ;
CoordinateDimensionality type;
int numRings; // >=1 as there has to be at least one ring
Ring[] rings; // size = numRings
}
struct MultiCurvePolygon
{
int geomType;
int numPolygons; // >=0
CurvePolygon[] polygons; // size = numElements
}
```

In the following example in the OpenGIS specification, a polygon within the byte array representing the stream is investigated:



T = 3 stands for GeometryType == GeometryType.Polygon

CT = 0 stands for CoordinateDimensionality == CoordinateDimensionality.XY

NR = 2 stands for number of rings = 2

NP =3 stands for number of points = 3

GIS Geometry API

There are two classes in the GIS Geometry API relevant to AGF:

- GisGeometryStreamFactory
- GisAgfGeometryFactory

GisGeometryStreamFactory

The GisGeometryStreamFactory class is a factory (abstract) for classes dealing with serialized geometric data. The data source is defined by the implementation. This is a helper type and does not inherit from GisIGeometry.

GisAgfGeometryFactory

The GisAgfGeometryFactory class is an AGF-based geometry factory, a concrete class that implements all the members from GisGeometryFactoryAbstract.

AGF Text

AGF Text is the textual analogue to the binary AGF format. It is a superset of the OGC WKT format. XY dimensionality is the default, and is optional. AGF Text can be used to represent any geometry value in the Geometry API, whether or not it originates from the AGF geometry factory. Conversions are done with the following methods:

- GisGeometryFactoryAbstract:: CreateGeometry(GisString* text);
- GisIGeometry:: GetText();"

Abstract Geometry Factory

Geometries in AWKB format can be exchanged between software components without depending on the Geometry API itself, because they are not genuine geometry "objects." AWKB content is based on byte arrays. It is handled through a simple GisByteArray class that is not specific to geometry.

Geometry Types

The Geometry types comprise the Global Enum GisGeometryType. The following are Geometry types:

- 0x00 GisGeometryType_None Indicates no specific type; used for "unknown", "do not care" or an incompletely constructed Geometry object.

NOTE GisGeometryType_ None does not represent an instantiable type. An FDO client should not expect an FDO provider to list support for it in its capabilities.

- 01 GisGeometryType_Point Point type (GisIPoint).
- 02 GisGeometryType_LineString LineString type (GisILineString).
- 03 GisGeometryType_Polygon Polygon type (GisIPolygon).
- 04 GisGeometryType_MultiPoint MultiPoint type (GisIMultiPoint).
- 05 GisGeometryType_MultiLineString MultiLineString type (GisIMultiLineString).
- 06 GisGeometryType_MultiPolygon MultiPolygon type (GisIMultiPolygon).
- 07 GisGeometryType_MultiGeometry MultiGeometry type (GisIMultiGeometry).
- 10 GisGeometryType_CurveString CurveString type (GisICurveString).
- 11 GisGeometryType_CurvePolygon CurvePolygon type (GisICurvePolygon).
- 12 GisGeometryType_MultiCurveString MultiCurveString type (GisIMultiCurveString).
- 13 GisGeometryType_MultiCurvePolygon MultiCurvePolygon type (GisIMultiCurvePolygon

Mapping Between Geometry and Geometric Types

The FDO API GeometricType enumeration of GeometricProperty gives the client application some knowledge of which geometry types comprise the geometric property so that it can present the user with an intelligent editor for selecting styles for rendering the geometry. In particular, GeometricType relates to shape dimensionality of geometries allowed in FDO geometric properties. The nearest analogues in the Geometry API are:

- GisDimensionality, which pertains to ordinate (not shape) dimensionality of geometry values.
- GisGeometryType, which has types whose abstract base types map to Geometric Type

The GeometricType enumeration is as follows:

- Point = 0x01, // Point Type Geometry
- Curve = 0x02, // Line and Curve Type Geometry
- Surface = 0x04, // Surface (or Area) Type Geometry
- Solid = 0x08, // Solid Type Geometry

NOTE The enumeration defines a bit mask and the GetGeometricTypes and SetGeometricTypes methods take and return an integer. This is to allow a geometry property to be of more than one type. For example, the call:

```
geometricProperty.SetGeometricTypes(Point | Surface);
```

would allow the geometric property to represent either point type geometry or surface type geometry (polygons).

Spatial Context

Spatial Context is a coordinate system with an identity. Any geometries that are to be spatially related must be in a common spatial context.

Providing an identify for each coordinate system supports separate workspaces, such as schematic diagrams, which are non-georeferenced. However, there are also georeferenced cases. In general, two users may create drawings using the same default spatial parameters (for example, rectangular and 10,000x10,000) that have nothing to do with each other. If their drawings are to be put into a common database, the spatial context capability of FDO preserves the container aspect of the data along wih the spatial parameters.

The FDO Spatial Context Commands are part of the FDO API. They support control over Spatial Contexts in the following ways:

- **Metadata control.** Creates and deletes Spatial Contexts.
- **Active Spatial Context. A s**ession setting to specify which Spatial Context to use by default while storing/retrieving geometries and performing spatial queries.

There is a default Spatial Context for each database. Its attributes (such as coordinate system) are specified when the database is created. This Spatial Context is the active one in any FDO session until a Spatial Context Command is used to change this state. The default Spatial Context's identifier number is 0 (zero).

Spatial contexts have two tolerance attributes: XYTolerance and ZTolerance. The tolerances are in distance units that depend on the coordinate system in use. Geodetic coordinate systems typically have "on the ground" linear distance units instead of the angular (that is, degrees, minutes or seconds) units used for positional ordinates. The meter is the most common unit. Most non-geodetic systems are rectilinear and use the same unit for positional ordinates and distances, for example, meters or feet.

Inserting Geometry Values

For information about geometry property values, see Geometry Property Values (page 96).

See Example: Inserting an Integer, a String, and a Geometry Value (page 96) for a code example that shows how to insert a Geometry value.

Autodesk FDO Provider for Oracle

A

This appendix discusses FDO API development issues that are related to using FDO Provider for Oracle.

In this chapter

What Is FDO Provider for Oracle?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. FDO Provider for Oracle provides FDO with access to an Oracle-based data store.

FDO Provider for Oracle API provides custom commands specifically designed to work with the FDO API. For example, using these commands, you can do the following:

- Gather information about a provider.
- Transmit client services exceptions.
- Get lists of accessible data stores.
- Create connection objects.
- Create and dropping spatial indexes.

FDO Provider for Oracle General Requirements

For Autodesk Map 3D users, a pre-requisite for creating schema and managing long transactions is to include the setting WM_ADMIN_ROLE in the user definition.

If a user definition does not have this setting, use the *FDO User Manager Tool* to delete the user definition and then recreate it to include WM_ADMIN_ROLE.

FDO Provider for Oracle Connection

This information supplements the Establishing a Connection chapter. You connect to a data store directly through FDO Provider for Oracle, and the underlying data source for the data store is an Oracle database.

You can connect to the data store in one step if you already know its name. Otherwise, you must connect in two steps.

The minimum required connection parameters for the initial call to Open() are service, username, and password.

The service parameter is the Oracle Net Service Name of an Oracle instance. An instance could be running on your machine or on some other machine in the network. You can use the Oracle Net Manager to identify which Oracle instances are available to you and what their Net Service Names are. In an Oracle 10*g* installation on a Microsoft Windows XP machine, Net Manager can be accessed with Start ➤ Programs ➤ Oracle ➤ Configuration and Migration Tools ➤ Net Manager. The connection information for the Net Service Name is contained in a file named *tnsnames.ora*, which is located in the Network/Admin folder in either the Oracle instance or the client installation directory.

Multiple users can access the data store. However, access is password-protected.

An Oracle data source, when accessed by FDO Provider for Oracle, may contain more than one data store. For the first call to Open(), a data store name is optional. If successful, the first call to Open() results in the data store parameter becoming a required parameter and a list of the names of the data stores in the data source becoming available. You must choose a data store and call Open() again.

If you know the name of the data store, you can provide it for the first call to Open() and make the connection in one step.

FDO Provider for Oracle and Foreign Schemas

FDO Provider for Oracle supports the creation of foreign schemas. A foreign schema is capable of mapping a table to Oracle instances. This allows users with a pre-existing application (for example, one created with Autodesk GIS Design Server) to map their application to FDO. As a result, both the FDO capability and conventional capability can be used by the same application.

Foreign Schema Settings

In order to use a foreign schema, certain privileges for FDO are required. To support the foreign schema capability, the following settings are required:

- FDO schema instance
- Foreign schema instance
- Oracle identity property

These settings are required for accessing the foreign schema objects (that is, tables, views, and sequences.).

Settings on the FDO Schema Instance

If the foreign schema is on a different Oracle instance, create a PUBLIC database link. A database link is a schema object that enables accessing of objects on another Oracle instance.

Settings on the Foreign Schema Instance

If the foreign schema is on a different Oracle instance, create an FDO user using the same Username and password as on the instance where the FDO schema exists.

NOTE This FDO user does not need to have been granted the f_user_role role.

Grant the select, update, delete, and insert privileges on tables (views, sequences) to the FDO user that is mapped to the FDO schema. Note that if the Foreign Schema tables are enabled for Oracle Workspace Manager, the privileges must be granted for the based table.

Oracle Identity Property

When specifying the main identifier for a feature class with FDO Provider for Oracle, the data type must be Int64. It must also have the following settings: ReadOnly=True and Autogenerated=True. If an identifier is not created with these properties, it will be created by FDO Provider for Oracle. Otherwise, an exception may be raised in certain conditions.

If the foreign schema uses a different data type for identifiers, the user must define the identifier as a fdo int64 type with ReadOnly=True and Autogenerated=True in the XML configuration file. If the identifier in the foreign schema uses an Oracle sequence to generate the unique numbers, the sequencename override must be defined in the override XML file.

Read-Write Privileges

If FDO requires read-write privileges to work with a foreign schema, the owner of the foreign schema must grant these privileges. Also, access to the Oracle metaschema on the tables is required.

NOTE The owner must explicitly grant these privileges. These privileges will vary, according to the schema owner.

Foreign Schema Limitations

This section provides information about known limitations of foreign schemas.

Ensuring Valid Views When Applying a Feature Schema Against a Foreign Schema

The ApplySchema command can creates invalid views when the feature schema is applied against a foreign schema. When you apply against a foreign schema, tables and columns are not automatically created if they do not already exist. A view is created, however, in the connected data store that references these foreign tables and columns. ApplySchema succeeds even if certain tables or columns cannot be obtained for various reasons, such as:

- The database link to the foreign tables is invalid.
- The Oracle instance containing the foreign tables is shut down or unreachable due to network problems.
- The foreign table or column simply does not exist.

When one of these situations occurs, ApplySchema creates invalid (dangling) views because these views reference tables or columns that cannot be reached.

Invalid views can occur regardless of whether schema overrides have been specified because the default schema mappings can also reference unreachable tables or columns. For example, if the feature schema being applied has a Pole class with no table name override, the Pole class is mapped to the POLE table in the foreign schema and a POLE view (referencing the POLE table) is created in the data store that the Oracle provider is currently connected to. If the POLE table does not exist, then the POLE view is dangling, or invalid.

Use one of the following procedures to correct invalid views, depending on whether the correct table name was specified (either through a schema override or the default class to table mapping rule):

Procedure When Table Name Is Correct

If the table name is correct, but it is not reachable for the reasons listed above:

1. Create the table or make it reachable by fixing the database link, fixing network problems, or starting the Oracle instance that contains the table.
2. Recompile the view that references the table.

The Oracle SQL statement for recompiling the view is:

Alter view <view_name> compile;

If there are a number of views to recompile, an alternative method is to use the following procedure for a wrong table name, but only do Steps 1 and 3.

Procedure When Wrong Table Name Is Specified

If the wrong table name is specified, to ensure valid views:

1. Destroy the feature schema. As long as the feature schema maps onto a foreign schema, destroying it does not result in loss of data. No tables or columns are dropped, only the referencing views created by FDO Provider for Oracle are dropped.
2. Fix the schema overrides to supply the proper table name. In some cases, you may need to add a schema override. For example, if a class named "Pole" corresponds to the foreign table "telco_pole", then a table name override must be specified for the Pole class, since the class and table names differ.
3. Re-apply the feature schema.

If your feature schema contains a mixture of classes mapped to foreign tables and classes mapped to non-foreign tables, then the procedure is slightly more complex, especially if any non-foreign table contains data. In this case, the following steps must be done programmatically throughout the FDO API:

1. Describe the feature schema using the DescribeSchema command. Retain this description.

2 Remove every class, except the one with the wrong table name, from the feature schema returned. However, do not delete the classes (that is, do not call FdoClassDefinition::Delete()).

3 Describe the feature schema again (ensure that you retain the feature schema from the first Describe).

4 Delete the class with the wrong table name from the feature schema returned by the second Describe (by calling its FdoClassDefinition::Delete() function).

5 Fix the schema overrides to supply the proper table name.

6 Ensure that FdoIApplySchema::SetIgnoreStates() is set to false, then Apply the feature schema described from Step 3. This deletes the class to repair.

7 Ensure that FdoIApplySchema::SetIgnoreStates() is set to true, then Apply the feature schema described from Step 1, along with the schema overrides. This re-creates the class to repair.

Overrides Capable of Causing Invalid Views

Any schema mapping between a feature schema element and a table or column can create invalid views. This is true for mappings specified through schema overrides or for default mappings. The specific schema mappings that can cause invalid views are as follows:

- Class to Table name
- Class to Geometry Column name
- Data Property to Column name
- Geometric Property to Column name
- Object Property to Table name

Table Name Restrictions When Working with a Foreign Schema

The ApplySchema command does not automatically create tables in foreign schemas. Therefore, the table specified for each class must already exist in the foreign schema. The Schema Overrides, specified through ApplySchema.SetPhysicalMapping, must contain a class to table mapping for each class whose table is named differently from the class. No mapping is required for classes where the table name and class name are the same.

Schema Access on a Different Oracle Instance

The following are Oracle limitations on foreign schema access if the schema is on a different (remote) Oracle instance:

- LOB type columns are not supported.
- Versioning and locking using Oracle Workspace Manager are not supported.

FDO Provider for Oracle and Schema Overrides

Schema overrides are supported through the Overrides API that is specific to FDO Provider for Oracle. This API is published as part of the FDO SDK.

Schema Override Set

A schema override set is the set of schema overrides for a particular Feature Schema and FDO Provider.

The top level of a schema override set is very similar to the Feature Schema, itself. There is a root class (OraclePhysicalSchemaMapping), with a list of classes and a list of relations. These lists are subsets of the lists in the corresponding Feature Schema. It is not necessary to list every class and relation; list only the ones for which overrides are being specified. OraclePhysicalSchemaMapping provides the Oracle-specific implementation of FdoPhysicalSchemaMapping.

The methods for these MetaClasses are stripped down from the methods on the corresponding Feature Schema MetaClasses. In the Schema Override set, only name and physical properties are specified. For example, the names for schema objects can be specified, but not the descriptions, since the descriptions cannot be overridden. Name cannot be overidden either, but each object needs a name for identification, so it must be specified.

The Schema Override Set is used to specify schema-wide overrides such as:

- Oracle Database for all tables for classes and object properties in the schema. Defaults to the current Oracle Database for the current connection.
- Oracle Owner for all tables for classes and object properties in the schema. Defaults to the current Oracle Schema for the current connection.
- Tablespace for all tables for classes and object properties in the schema. Defaults to the default table space for the Oracle Owner.
- Default table mapping type for all classes in the schema. If not specified, the default table mapping type is Concrete.

These schema-wide overrides can themselves be overridden on an element-by-element basis. For example, there are overrides available for class table, object property, and geometric property.

Class Table Overrides

The RDBMS table for storing class properties can be specified by adding a table to the class. The table specifies the table name and table primary key name. By default, the table name is set to be the same as the class name.

Data Property Overrides

The physical representation for a data property can be overridden by attaching a column to it. The column specifies the name of the property's corresponding column in the FDO database. If Column is not specified, then the column names default to Name (the property name).

Object Property Overrides

The type of an Object Property is a class in a Feature Schema. This class can be considered the referenced class. This referenced class has properties, so a home for each property must be provided in the RDBMS data store. There are are two different ways to store these properties. The Mapping Definition for each Object Property is specified by setting its MappingDefinition to an object of one of the following classes:

- **PropertyMappingSingle.** The referenced class properties are embedded in the containing class's table. The containing class is the class containing the Object Property.
- **PropertyMappingConcrete.** The Object Property is not stored in the containing class's table. A separate table is automatically generated for it.

Geometric Property Overrides

The column for a Geometric property can be overridden by attaching a Geometric column to it. Only the column name can be specified. The column type must always be mdsys.sdo_geometry.

The default column depends on whether the F_Geometry_0 table is present and whether the Geometric Property is also the GeometryProperty for its containing class.

If it is the GeometryProperty and F_Geometry_0 exists, then the table for this property is F_Geometry_<n>, where <n> is the ID of the Spatial Context Group for the associated Spatial Context. If the Geometric Property is not associated with a Spatial Context, then <n> is the ID of the active Spatial Context group, at the time the geometric property is created. The column for the property is always RDBMS_GEOM.

Otherwise, the column name is assumed to be the same as the property name. The column table is assumed to be the table for the containing class.

Oracle-Specific Schema Creation Restrictions

This section describes the restrictions that apply when creating schema(s) using FDO Provider for Oracle.

FDOFeatureClass

An FdoFeatureClass must have an identity.

Classes

- Class names must be unique across the data store.
- FdoFeatureClass must define or inherit at least one IdentityProperty.

Properties

Restrictions apply to specific types of properties.

Data Properties

- The default value must not be specified.
- A non-nullable data property cannot be added to a class that already has data.

Identity Properties

- Identity properties cannot be nullable.
- Read-only Identity properties must be autogenerated.

String Properties

String property length must be between 1 and 4000 bytes inclusive.

Decimal Properties

- Decimal property precision must be between 0 and 38 inclusive.
- Decimal property scale must be between -84 and 127 inclusive.

Geometric Properties

- Only FdoFeatureClass can have geometric properties. A feature class can have multiple geometric properties; main geometry is not mandatory. HasMeasure and HasElevation are supported.
- If the geometric property values are stored in a feature geometry system table (F_GEOMETRY_<n>), then HasMeasure must be false.

Object Properties

- The object property class must be an FdoClass. (FdoFeatureClass is not allowed.)
- IdentityProperty is mandatory if ObjectType is not FdoObjectType_Value and the object property class has no identity properties.

Oracle-Specific Schema Modification Restrictions

This section describes restrictions that apply when modifying schema(s) using FDO Provider for Oracle.

Almost all modifications are disallowed, with the exception of those that follow.

Schema Element Descriptions

- Any schema element description is allowed.
- Any schema attribute dictionary (entries can be added, deleted, or modified) is allowed.

Data Properties

The read-only setting for a data property can be modified if the property is not autogenerated.

Oracle-Specific Deletion Restrictions

This section describes restrictions that apply when performing deletion in a schema while using FDO Provider for Oracle.

FDOClassDefinition

FdoClassDefinition cannot be deleted if it has data (objects).

FdoClassDefinition cannot be deleted if another class in the data store has it as its base class.

FDOClass

FdoClass cannot be deleted if it is referenced by any object property in the data store.

Property

- A data property cannot be deleted if it has any non-null values.
- An object property cannot be deleted if it has data.
- A geometric property cannot be deleted if its containing class has data.

Oracle Reserved Words Used with Filter and Expression Text

When using a filter string with Oracle reserved words, the string within the expression must be encapsulated inside single quotes (following the same convention used with the SQL language). Failure to do so will result in a parsing error because the parser cannot determine any difference between the value and the keyword.

Example of a filter string:

```
AND='linetype'
```

This FDO constraint appliesto the Oracle reserved words:

- AND
- DATE
- IN
- LIKE
- NOT
- OR

Locking and Long Transactions

The purpose of this section is two-fold. First, it illustrates ways of understanding the subtleties of the interactions between locking and long transactions in an Oracle context. Secondly, it provides concrete examples of those subtleties.

An FDO long transaction version is called a workspace in an Oracle context. In this discussion, the FDO phrase “long transaction version” is shortened to “long transaction”. A key phrase in the example is “root,” which represents permanent data. Any long transaction has a root long transaction as an ancestor. The Oracle Workspace Manager (OWM) name for the FDO root long transaction is “LIVE”.

OWM and FDO Lock Types

The following table shows the names of the Oracle Workspace Manager locks used to implement each FDO lock:

FDO Lock Type	Oracle Lock Type
Shared	Shared

FDO Lock Type	Oracle Lock Type
Exclusive	Workspace Exclusive
Long Transaction Exclusive	Version Exclusive
All Long Transaction Exclusive	Exclusive

Example: AllLongTransactionExclusiveLock

The following is a proven example using the AllLongTransactionExclusiveLock type with multiple users and the Update command. When you connect to an Oracle data store, you are placed in the already-activated, default root long transaction. If a long transaction is created in root, it is considered a child of root. When the new long transaction is activated (for example, as LT1), the subsequent actions take place in the context of LT1. If another long transaction is subsequently created (for example, as LT2), it is created as a child of LT1.

NOTE When using FDO Provider for Oracle long transactions and locking, the combination of Oracle Workspace Manager capabilities and, potentially, other third-party applications can introduce many variables and combinations. The possible resulting conflicts in locking and long transactions can be similarly wide and varied.

This example considers two closely related cases. The same set of actions are taken in both cases, but in slightly different sequences, yielding different results. User1 creates a long transaction in the context of root and it is activated. User1 applies an AllLongTransactionExclusiveLock to a feature object in a data store. User1 updates that feature object in the data store. User2 attempts to update the same object, in the same data store, in the context of root. In the first case, User2 succeeds, and in the second case User2 fails (that is, a lock conflict is reported).

More specifically, the sequence of events for both cases is captured in the accompanying Long Transaction and Locking Sequencing Example diagram. For this example, all events occur in a single data store. The sequence of events are:

1. User1 creates LT1, activates LT1, and updates feature object “a” in LT1.
2. User2 successfully updates object “a” in root.

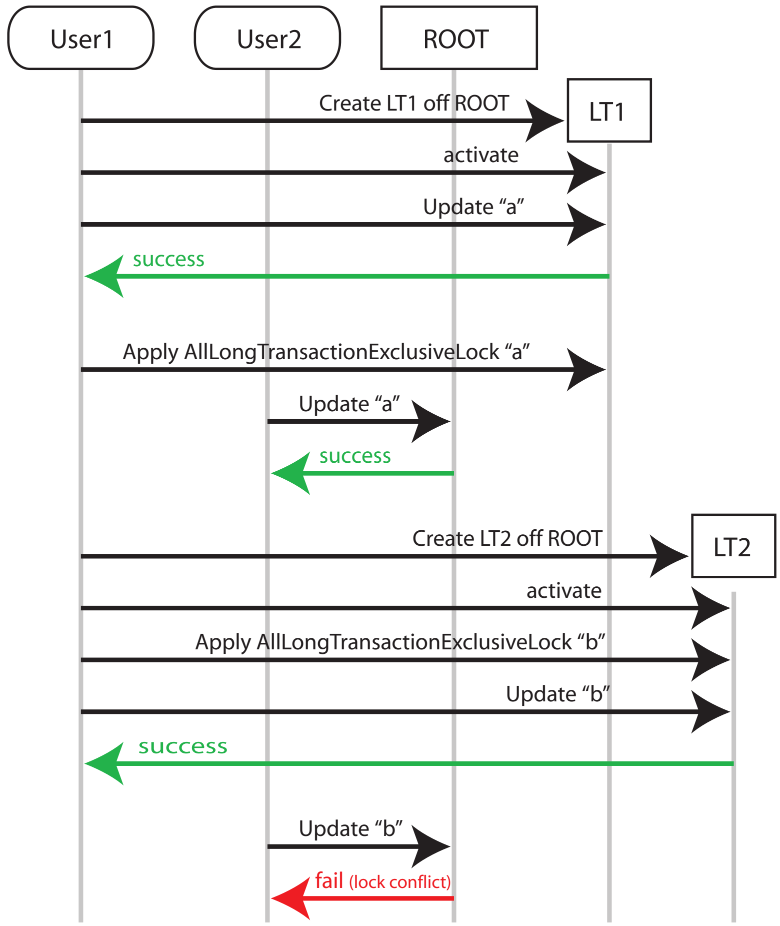
3 User1 creates LT2, activates LT2, and updates feature object “b” in LT2.

4 User2 fails to update object “b” in root.

The key difference is that, in LT1, User1 updates feature object “a” before the lock is applied, and, in LT2, User1 applies the lock to feature object “b” before it is updated. Prior to update, a copy of object "b" has not been made in LT2. This causes the lock to be applied to the copy of the object in root, because there is not yet a copy in LT2.

Therefore, if User1 intends to prevent anyone from modifying the object from the root level, User1 must apply the lock to the object *before* updating it.

For more information about Oracle Workspace Manager and its lock management, see the Oracle documentation.



Long Transaction and Locking Sequencing Example

FDO Provider for Oracle Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Filters

- Expressions
- Geometry
- Raster

Connection Capabilities

Capability	Oracle
Spatial Content Extent Types	
Static	Y
Dynamic	
Lock Types	
Shared	Y
Exclusive	Y
Transaction	Y
All Long Transaction Exclusive	Y
Long Transaction Exclusive	Y
Thread Types	
Single- Threaded	
Per Connection Threaded	Y
Per Command Threaded	
Multi-threaded	

Capability	Oracle
Supports	
Locking	Y
Timeout	
Transactions	Y
Long Transactions	Y
SQL	Y
Configuration	

Schema Capabilities

Capability	Oracle
Class Types	
Class	Y
Feature Class	Y
Data Types	
Boolean	Y
Byte	Y
DateTime	Y
Decimal	Y
Double	Y

Capability	Oracle
Int16	Y
Int32	Y
Int64	Y
Single	Y
String	Y
BLOB	Y
CLOB	Y
Auto-Generated Data Types Supported (Int64)	Y
Supports	
Inheritance	Y
Multiple Schemas	Y
Object Properties	Y
Association Properties	Y
Schema Overrrides	Y
Network Model	
Auto Id Generation	Y
Data Store Scope Unique Id Generation	Y

Capability	Oracle
Schema Modification	Y
Constraints	
Inclusive Value Range Constraints	Y
Exclusive Value Range Constraints	
Value Constraints List	
Null Value Constraints	Y
Unique Value Constraints	
Composite Unique Value Constraints	

Command Capabilities

Capability	Oracle
Feature Commands	
Select	Y
Select Aggregates	Y
Insert	Y
Delete	Y
Update	Y
Schema Commands	
Describe Schema	Y

Capability	Oracle
Describe Schema Mapping	Y
Apply Schema	Y
Destroy Schema	Y
Spatial Context Commands	
Activate Spatial Context	Y
Create Spatial Context	Y
Destroy Spatial Context	Y
Get Spatial Contexts	Y
Data Store Commands	
Create Data Store	Y
Destroy Data Store	Y
List Data Stores	Y
Measure Unit Commands	
Create Measure Unit	
Destroy Measure Unit	
Get Measure Units	
SQL Command	Y
Locking Commands	

Capability	Oracle
Acquire Lock	Y
Get Lock Info	Y
Get Locked Objects	Y
Get Lock Owners	Y
Release Lock	Y
Long Transaction Commands	
Activate Long Transaction	Y
Deactivate Long Transaction	Y
Commit Long Transaction	Y
Create Long Transaction	Y
Get Long Transactions	Y
Freeze Long Transaction	
Rollback Long Transaction	Y
Activate Long Transaction Checkpoint	
Create Long Transaction Checkpoint	
Get Long Transaction Checkpoints	
Rollback Long Transaction Checkpoint	
Change Long Transaction Privileges	

Capability	Oracle
Get Long Transaction Privileges	
Change Long Transaction Set	
Get Long Transaction In Set	
RDBMS Custom Commands	
Create Spatial Index	Y
Destroy Spatial Index	Y
Get Spatial Indexes	Y
Supports	
Parameters	
Timeout	
Select Expressions	Y
Select Functions	Y
Select Distinct	Y
Select Ordering	Y
Select Grouping	Y

Filter Capabilities

Capability	Oracle
Condition Types	

Capability	Oracle
Comparsion	Y
Like	Y
In	Y
Null	Y
Spatial	Y
Distance	Y
Spatial Operations	
Contains	
Crosses	
Disjoint	
Equals	
Intersects	Y
Overlaps	
Touches	
Within	
Covered By	Y
Inside	Y
Envelope Intersects	Y

Capability	Oracle
Distance Operations	
Beyond	
Within	Y
Supports	
Geodesic Distance	
Non-Literal Geometric Operations	

Expression Capabilities

Capability	Oracle
Expression Types	
Basic	Y
Function	Y
Parameter	Y
Functions	
Double Avg(Double)	Y
Int64 Ceil(Int64)	Y
String Concat(String)	Y
Int64 Count(Int64)	Y
Int64 Floor(Int64)	Y

Capability	Oracle
String Lower(String)	Y
Double Min(Double)	Y
Double Max(Double)	Y
Double StdDev(Double)	
Double Sum(Double)	Y
String Upper(String)	Y
geomValue SpatialExtents (<Fdo-DataType>)	Y

Geometry Capabilities

Capability	Oracle
Geometry Types	
Point	Y
Line String	Y
Polygon	Y
Multi-Point	Y
Multi-Line String	Y
Multi-Polygon	Y
Multi-Geometry	

Capability	Oracle
Curve String	Y
Curve Polygon	Y
Multi-Curve String	Y
Multi-Curve Polygon	Y
Geometry Component Types	
Linear Ring	Y
Line String Segment	Y
Circular Arc Segment	Y
Ring	Y
Dimensionalities	
XY	Y
Z	Y
M	Y

Raster Capabilities

Capability	Oracle
Supports	
Raster	
Stitching	

Capability	Oracle
Subsampling	
Bitonal Data Model	
Gray Data Model	
RGB Data Model	
RGBA Data Model	
Palette Data Model	

OSGeo FDO Provider for ArcSDE

B

This appendix discusses FDO API development issues that are related to OSGeo FDO Provider for ArcSDE.

In this chapter

What Is FDO Provider for ArcSDE?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. ESRI® ArcSDE® (Spatial Database Engine) is part of the ArcGIS 9 system. ArcSDE manages the exchange of information between an (ArcGIS 9 Desktop) application and a relational database management system. FDO Provider for ArcSDE provides FDO with access to an ArcSDE 9-based data store, which, in this case, must be Oracle 9*i* (9.2.0.6).

FDO Provider for ArcSDE Software Requirements

Installed Components

FDO Provider for ArcSDE dynamically linked libraries are installed with the FDO SDK. They are located in <FDO SDK Install Location>\FDO\bin. You do not have to do anything to make these DLLs visible.

External Dependencies

The operation of FDO Provider for ArcSDE is dependent on the presence of ArcSDE 9 and a supported data source, such as Oracle 9*i*, in the network environment. The host machine running FDO Provider for ArcSDE must also have the required DLLs present, which are available by installing either an ArcGIS 9.1 Desktop application or the ArcSDE SDK. For example, the required DLLs are present if either ArcView®, ArcEditor®, or ArcInfo® are installed. For more information about ArcGIS 9.1 Desktop applications and the ArcSDE SDK, refer to the ESRI documentation.

Specifically, in order for FDO Provider for ArcSDE to run, three dynamically linked libraries, sde91.dll, sg91.dll, and pe91.dll, are required and you must ensure that the PATH environment variable references the local folder containing these DLLs. For example, in Microsoft Windows, if ArcGIS 9.1 Desktop is installed to C:\Program Files\ArcGIS, then the required ArcSDE binaries are located at C:\Program Files\ArcGIS\ArcSDE\bin. Similarly, if the

ArcSDE SDK is installed to the default location, then the required ArcSDE binaries are located at C:\ArcGis\ArcSDE\bin. The absence of this configuration may cause the following exception message "The ArcSDE runtime was not found.".

FDO Provider for ArcSDE Limitations

The FDO Provider for ArcSDE is based on a subset of the ArcSDE API. This subset does not include the following:

- Raster functionality
- Native ArcSDE metadata
- The annotation data, with the exception of the ANNO_TEXT column

ArcSDE Limitations

FDO Provider for ArcSDE must abide by limitations of the ArcSDE technology to which it connects. This section discusses these limitations.

Relative to ArcObjects API and ArcGIS Server API

The ArcSDE API does not support the following advanced functionality found in the ArcObjects API and the newer ArcGIS Server API:

- Advanced geometries, such as Bezier curves and ellipses
- Relationships
- Topology
- Networks
- Analysis
- Linear referencing

Curved Segments

If ArcSDE encounters curved segments, it will automatically tessellate them. This means that if you create a geometry containing an arc segment in an ArcSDE table using ArcObjects API and then you try to read that geometry back using the ArcSDE API, you will get a series of line segments that approximate the original arc segment. That is, you get an approximation of the original geometry.

Locking and Versioning

ArcSDE permits row locks or table versioning provided that the ID column, which uniquely identifies the row, is maintained by ArcSDE. If there is no ID column or the ID column is maintained by the user, ArcSDE does not permit row locking or table versioning to be enabled.

NOTE In ArcSDE you can either lock rows in a table or version a table, but you cannot do both at the same time. To do either, you must alter the table's registration.

The following sections illustrate these three steps:

1. The creation of a table.
2. The alteration of the table registration to identify one of the column definitions as the row ID column and to enable row locking.
3. The alteration of the table registration to disable row locking and to enable versioning.

Table Creation

The command is:

```
sdetable -o create -t hassdemaintainedrowid -d "name string(20),
fid integer(9)" -u t_user -p test
```

The output of the describe registration command (sdetable -o describe_reg) for this table is as follows:

NOTE The Row Lock has no value and the value of Dependent Objects is None.

```
Table Owner             : T_USER
Table Name              : HASSDEMAINTAINEDROWID
Registration Id         : 18111
Row ID Column           :
Row ID Column Type      :
Row Lock                :
Minimum Row ID          :
Dependent Objects       : None
Registration Date       : 02/24/05 13:08:02
Config. Keyword         : DEFAULTS
User Privileges         : SELECT, UPDATE, INSERT, DELETE
Visibility              : Visible
```

Identity Row ID Column and Enable Row Locking

The command is:

```
sdetable -o alter_reg -t hassdemaintainedrowid -c fid -C sde -L
on -u t_user -p test
```

The output of the describe registration command (sdetable -o describe_reg) for this table is as follows.

NOTE The Row ID Column value is FID, the Row ID Column Type value is SDE Maintained, and the Row Lock value is Enable.

```
Table Owner             : T_USER
Table Name              : HASSDEMAINTAINEDROWID
Registration Id         : 18111
Row ID Column           : FID
Row ID Column Type      : SDE Maintained
Row ID Allocation       : Many
Row Lock                : Enable
Minimum Row ID          : 1
Dependent Objects       : None
Registration Date       : 02/24/05 13:08:02
Config. Keyword         : DEFAULTS
User Privileges         : SELECT, UPDATE, INSERT, DELETE
Visibility              : Visible
```

Disable Row Locking and Enable Versioning

The command is:

```
sdetable -o alter_reg -t hassdemaintainedrowid -L off -V MULTI -u
 t_user -p test
```

The output of the describe registration command (sdetable -o describe_reg) for this table is as follows:

NOTE The “Row Lock” is “Not Enable” and “Dependent Objects” is “Multiversion Table”.

```
Table Owner            : T_USER
Table Name             : HASSDEMAINTAINEDROWID
Registration Id        : 18111
Row ID Column          : FID
Row ID Column Type     : SDE Maintained
Row ID Allocation      : Many
Row Lock               : Not Enable
Minimum Row ID         : 1
Dependent Objects      : Multiversion Table
Dependent Object Names : A18111, D18111
Registration Date      : 02/24/05 13:08:02
Config. Keyword        : DEFAULTS
User Privileges        : SELECT, UPDATE, INSERT, DELETE
Visibility             : Visible
```

FDO Provider for ArcSDE Connection

This information supplements the Establishing a Connection chapter. You connect to an ArcSDE data store indirectly through the ArcSDE server. The underlying data source for the data store is a database, such as Oracle. The ArcSDE server is connected to the data source and mediates the requests that you send it.

You can connect to FDO Provider for ArcSDE in one step if you already know the name of the data store that you want to use. Otherwise, you must connect in two steps.

The minimum required connection properties for the initial call to Open() are server, instance, username, and password. Multiple users can access the data store. However, access is password-protected. The server property is the name of the machine hosting the ArcSDE server. The instance property acts as an index into an entry in the services file. An entry contains port and protocol information used to connect to the ArcSDE server. On a Windows machine, the services file is located in C:\WINDOWS\system32\drivers\etc. Assuming that the instance name is "esri_sde", an entry would look something like this: "esri_sde 5151/tcp #ArcSDE Server Listening Port".

An ArcSDE data source may contain more than one data store. For the first call to Open(), a data store name is optional. If successful, the first call to Open() results in the data store parameter becoming a required parameter and a list of the names of the data stores in the data source becoming available. You must choose a data store and call Open() again.

If the data source supports multiple data stores, the list returned by the first call to Open() will contain a list of all of the data stores resident in the data source. Otherwise, the list will contain one entry: "Default Data Store".

If you know the name of the data store, you can provide it for the first call to Open() and make the connection in one step.

Data Type Mappings

This section shows the mappings from FDO data types to ArcSDE data types to Oracle data types:

FDO DataType	sdetable Column Definition	Oracle Column Type
FdoDataType_Boolean	Not supported	Not supported
FdoDataType_Byte	Not supported	Not supported
FdoDataType_DateTime	date	DATE
FdoDataType_Decimal	Not supported	Not supported
FdoDataType_Double	double(38,8)	NUMBER(38,8)
FdoDataType_Int16	integer(4)	NUMBER(4)
FdoDataType_Int32	integer(10)	NUMBER(10)

FDO DataType	sdetable Column Definition	Oracle Column Type
FdoDataType_Int64	Not supported	Not supported

FdoDataType_Single	float(6,2) // typical
	float(0<n<=6, o<m<DBMSLimit)) // possible
		NUMBER(6,2)
		NUMBER(n,8)

FdoDataType_String	string(<length>)	VARCHAR2(<length>)
FdoDataType_BLOB	blob	LONG RAW
FdoDataType_CLOB	Not supported	Not supported
FdoDatatype_UniqueID	Not supported	Not supported

Creating a Feature Schema

This section describes the creation of the SampleFeatureSchema, which is the example feature schema described in the Schema Management chapter. It also describes the creation of the OGC980461FS schema, which is the schema defined in the OpenGIS project document 98-046r1.

FDO Provider for ArcSDE does not support the creation or destruction of feature schema (that is, does not support the FdoIApplySchema and FdoIDestroySchema commands.) However, it does support the FdoIDescribeSchema command. The intended use of FDO Provider for ArcSDE is to operate on already existing feature schemas. FDO Provider for ArcSDE supports inserting, selecting, updating, and deleting data in existing schemas.

You can use FDO Provider for ArcSDE to operate on a new feature schema. However, you must create the schema using ArcSDE tools. In particular you use the sdetable and sdelayer commands, which can be used to create a schema in any of the data store technologies used by ArcSDE. This part of the description is generic. Other parts of the description are specific to Oracle and to Windows XP because Oracle is the data store technology and Windows XP is the operating system for this exercise.

First, you must create an Oracle username for the feature schema (that is, the name of the Oracle user is the name of the feature schema.) To do this, you connect as system administrator to the Oracle instance used by the ArcSDE server. The following command creates the user and grants to that user the privileges necessary for the ArcSDE tool commands to succeed:

```
grant connect,resource to <schemaName> identified by <password>
```

Secondly, you must log in to the host where the ArcSDE server is running. ArcSDE tools are on the host machine where the ArcSDE server resides.

TIP NetMeeting can be used to remotely login to where the ArcSDE Server is running and launch a command window (that is, in the Run dialog box, enter cmd) The ArcSDE tool commands can be executed through the command window. Do not use C:\WINDOWS\SYSTEM32\COMMAND.COM because the line buffer is too short to contain the entire text of some of the SDE tool command strings.

Finally, execute the sdetable and sdelayer commands in a command window to create each of the classes. Since you are executing these commands on the host where the ArcSDE server is located, you can omit the server name option. If the ArcSDE server is connected to only one data store, you can omit the service option. For more information about all of the ArcSDE commands, consult the ArcSDE Developer Help Guide.

SampleFeatureSchema

In this sample a feature schema called SampleFeatureSchema is created, which contains one feature class called SampleFeatureClass. This feature class has the following three properties:

- An Int32 called SampleIdentityDataProperty.
- A string called SampleNameDataProperty.
- A polygon geometry called SampleGeometricProperty.

First, use the sdetable -o create command to add the integer and string properties to SampleFeatureClass. Then, use the sdetable -o alter_reg command to identify the SampleIdentityDataProperty as an identity property. Finally, use the sdelayer -o add command to add the geometric property to SampleFeatureClass. This assumes that only one ArcSDE server service is running so that the -i option is optional. The -i option takes a service name as an argument.

The sdetable -o create command can be invoked as follows:

```
sdetable -o create -t SampleFeatureClass -d “SampleIdentityDataProp
erty INTEGER(10), SampleNameDataProperty STRING(64)” -u SampleFea
tureSchema -p test.
```

The -o option takes the command option name. The -d option takes the column definitions, which is a quoted list of column name/column type pairs delimited by commas. The -u option takes an Oracle database user name, which becomes the feature schema name. The -p option takes a password.

The sdetable -o alter_reg command is invoked as follows:

```
sdetable -o alter_reg -t SampleFeatureClass -c SampleIdentityDat
aProperty -C USER -u SampleFeatureSchema -p test
```

The -c option identifies the column name that will be the identity property. The -C option indicates whether SDE is supposed to generate the value or obtain it from the user. You will be prompted to confirm that you want to alter the registration of the table.

The sdelayer command is invoked as follows:

```
sdelayer -o add -l SampleFeatureClass,SampleGeometricProperty -E
0,0,100,50 -e a -u SampleFeatureSchema -p test
```

The -o option takes the command option name. The -l option identifies the table and column. The -E option identifies the extents; the arguments are <xmin,ymin,xmax,ymax>. The -e option identifies the geometry type with 'a' indicating an area shape.

OGC980461FS

This schema contains the ten classes defined in the OpenGIS Project Document *980946r1*. The types of the properties belonging to the classes is similar to that of SampleFeatureClass, namely, an integer, a string, and a geometry. One difference is that the geometry in three of the classes is multipart. Two of them have MULTIPOLYGON geometries, and one of them has a MULTILINESTRING geometry. A multipart geometry is indicated by adding a '+' to the entity argument to the -e option in the sdelayer command. A MULTIPOLYGON geometry is indicated by "-e a+", and a MULTILINESTRING geometry is indicated by "-e l+".

An ArcSDE table cannot have two geometries. This restriction impacts the definition of the buildings class, which has a POLYGON and a POINT geometry. We have chosen to add the POINT geometry. The OpenGIS 98-046r1 document defines one query that references building objects, and the POINT geometry supports this query.

NOTE The use of -E option in the sdelayer command defines the extents. The arguments are <xmin,ymin,xmax,ymax>. The values provided below ensure that you will not receive any "ordinate out of bounds" errors when inserting the 98046r1 data.

ArcSDE Commands That Define the OGC980461FS Classes

```
sdetable -o create -t lakes -d "fid integer(10), name string(64)"
 -u OGC980461FS -p test
sdetable -o alter_reg -t lakes -c fid -C user -u OGC980461FS -p
test
sdelayer -o add -l lakes,shore -E 0,0,100,50 -e a -u OGC980461FS
-p test
sdetable -o create -t road_segments -d "fid integer(10), name
string(64), aliases string(64), num_lanes integer(10)" -u
OGC980461FS -p test
sdetable -o alter_reg -t road_segments -c fid -C user -u
OGC980461FS -p test
sdelayer -o add -l road_segments,centerline -E 0,0,100,50 -e l -u
 OGC980461FS -p test
sdetable -o create -t divided_routes -d "fid integer(10), name
string(64), num_lanes integer(10)" -u OGC980461FS -p test
sdetable -o alter_reg -t divided_routes -c fid -C user -u
OGC980461FS -p test
sdelayer -o add -l divided_routes,centerlines -E 0,0,100,50 -e l+
 -u OGC980461FS -p test
sdetable -o create -t forests -d "fid integer(10), name string(64)"
 -u OGC980461FS -p test
sdetable -o alter_reg -t forests -c fid -C user -u OGC980461FS -p
 test
sdelayer -o add -l forests,boundary -E 0,0,100,50 -e a+ -u
OGC980461FS -p test
sdetable -o create -t bridges -d "fid integer(10), name string(64)"
 -u OGC980461FS -p test
sdetable -o alter_reg -t bridges -c fid -C user -u OGC980461FS -p
 test
sdelayer -o add -l bridges,position -E 0,0,100,50 -e p -u
OGC980461FS -p test
sdetable -o create -t streams -d "fid integer(10), name string(64)"
 -u OGC980461FS -p test
sdetable -o alter_reg -t streams -c fid -C user -u OGC980461FS -p
 test
sdelayer -o add -l streams,centerline -E 0,0,100,50 -e l -u
OGC980461FS -p test
sdetable -o create -t buildings -d "fid integer(10), address
string(64)" -u OGC980461FS -p test
sdetable -o alter_reg -t buildings -c fid -C user -u OGC980461FS
-p test
sdelayer -o add -l buildings,position -E 0,0,100,50 -e p -u
OGC980461FS -p test
```

```
sdetable -o create -t ponds -d "fid integer(10), name string(64),
 type string(64)" -u OGC980461FS -p test
sdetable -o alter_reg -t ponds -c fid -C user -u OGC980461FS -p
test
sdelayer -o add -l ponds,shores -E 0,0,100,50 -e a+ -u OGC980461FS
 -p test
sdetable -o create -t named_places -d "fid integer(10), name
string(64)" -u OGC980461FS -p test
sdetable -o alter_reg -t named_places -c fid -C user -u OGC980461FS
 -p test
sdelayer -o add -l named_places,boundary -E 0,0,100,50 -e a -u
OGC980461FS -p test
sdetable -o create -t map_neatlines -d "fid integer(10)" -u
OGC980461FS -p test
sdetable -o alter_reg -t map_neatlines -c fid -C user -u
OGC980461FS -p test
sdelayer -o add -l map_neatlines,neatline -E 0,0,100,50 -e a -u
OGC980461FS -p test
```

FDO Provider for ArcSDE Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Capability	ArcSDE
Spatial Content Extent Types	

Capability	ArcSDE
Static	Y
Dynamic	
Lock Types	
Shared	
Exclusive	Y
Transaction	
All Long Transaction Exclusive	
Long Transaction Exclusive	
Thread Types	
Single- Threaded	
Per Connection Threaded	Y
Per Command Threaded	
Multi-threaded	
Supports	
Locking	Y
Timeout	
Transactions	Y
Long Transactions	Y

Capability	ArcSDE
SQL	Y
Configuration	

Schema Capabilities

Capability	ArcSDE
Class Types	
Class	Y
Feature Class	Y
Data Types	
Boolean	
Byte	
DateTime	Y
Decimal	
Double	Y
Int16	Y
Int32	Y
Int64	
Single	Y
String	Y

Capability	ArcSDE
BLOB	Y
CLOB	
Auto-Generated Data Types Supported (Int32)	Y
Supports	
Inheritance	
Multiple Schemas	Y
Object Properties	
Association Properties	
Schema Overrrides	
Network Model	
Auto Id Generation	Y
Data Store Scope Unique Id Generation	
Schema Modification	
Constraints	
Inclusive Value Range Constraints	
Exclusive Value Range Constraints	
Value Constraints List	

Capability	ArcSDE
Null Value Constraints	Y
Unique Value Constraints	Y
Composite Unique Value Constraints	Y

Command Capabilities

Capability	ArcSDE
Feature Commands	
Select	Y
Select Aggregates	Y
Insert	Y
Delete	Y
Update	Y
Schema Commands	
Describe Schema	Y
Describe Schema Mapping	
Apply Schema	
Destroy Schema	
Spatial Context Commands	
Activate Spatial Context	Y

Capability	ArcSDE
Create Spatial Context	Y
Destroy Spatial Context	Y
Get Spatial Contexts	Y
Data Store Commands	
Create Data Store	
Destroy Data Store	
List Data Stores	Y
Measure Unit Commands	
Create Measure Unit	
Destroy Measure Unit	
Get Measure Units	
SQL Command	Y
Locking Commands	
Acquire Lock	Y
Get Lock Info	Y
Get Locked Objects	Y
Get Lock Owners	Y
Release Lock	Y

Capability	ArcSDE
Long Transaction Commands	
Activate Long Transaction	Y
Deactivate Long Transaction	Y
Commit Long Transaction	Y
Create Long Transaction	Y
Get Long Transactions	Y
Freeze Long Transaction	
Rollback Long Transaction	Y
Activate Long Transaction Checkpoint	
Create Long Transaction Checkpoint	
Get Long Transaction Checkpoints	
Rollback Long Transaction Checkpoint	
Change Long Transaction Privileges	
Get Long Transaction Privileges	
Change Long Transaction Set	
Get Long Transaction In Set	
RDBMS Custom Commands	
Create Spatial Index	

Capability	ArcSDE
Destroy Spatial Index	
Get Spatial Indexes	
Supports	
Parameters	Y
Timeout	
Select Expressions	
Select Functions	Y
Select Distinct	Y
Select Ordering	Y
Select Grouping	

Filter Capabilities

Capability	ArcSDE
Condition Types	
Comparsion	Y
Like	Y
In	Y
Null	Y
Spatial	Y

Capability	ArcSDE
Distance	Y
Spatial Operations	
Contains	Y
Crosses	Y
Disjoint	Y
Equals	Y
Intersects	Y
Overlaps	Y
Touches	Y
Within	Y
Covered By	Y
Inside	Y
Envelope Intersects	Y
Distance Operations	
Beyond	Y
Within	Y
Supports	
Geodesic Distance	

Capability	ArcSDE
Non Literal Geometric Operations	

Expression Capabilities

Capability	ArcSDE
Expression Types	
Basic	Y
Function	Y
Parameter	
Functions	
Double Avg(Double)	Y
Int64 Ceil(Int64)	
String Concat(String)	
Int64 Count(Int64)	Y
Int64 Floor(Int64)	
String Lower(String)	
Double Min(Double)	Y
Double Max(Double)	Y
Double StdDev(Double)	Y
Double Sum(Double)	Y

Capability	ArcSDE
String Upper(String)	
geomValue SpatialExtents (TBD - Fdo-DataType)	

Geometry Capabilities

Capability	ArcSDE
Geometry Types	
Point	Y
Line String	Y
Polygon	Y
Multi-Point	Y
Multi-Line String	Y
Multi-Polygon	Y
Multi-Geometry	
Curve String	
Curve Polygon	
Multi-Curve String	
Multi-Curve Polygon	
Geometry Component Types	

Capability	ArcSDE
Linear Ring	Y
Line String Segment	Y
Circular Arc Segment	
Ring	
Dimensionalities	
XY	Y
Z	Y
M	Y

Raster Capabilities

Capability	ArcSDE
Supports	
Raster	
Stitching	
Subsampling	
Bitonal Data Model	
Gray Data Model	
RGB Data Model	
RGBA Data Model	

Capability	ArcSDE
Palette Data Model	

OSGeo FDO Provider for MySQL

C

This appendix discusses FDO API development issues that are related to OSGeo FDO Provider for MySQL.

In this chapter

What Is FDO Provider for MySQL?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The FDO Provider for MySQL provides FDO with access to a MySQL-based data store.

The FDO Provider for MySQL API provides custom commands that are specifically designed to work with the FDO API. For example, using these commands, you can do the following:

- Gather information about a provider.
- Transmit client services exceptions.
- Get lists of accessible data stores.
- Create connection objects.
- Create and execute spatial queries.

The MySQL architecture supports different storage engines. Choose an engine as needed, depending on its characteristics and capabilities, such as the following:

- MyISAM is a disk-based storage engine. It does not support transactions.
- InnoDB is a disk-based storage engine. It has full ACID transaction capability.
- Memory (Heap) is a storage engine utilizing only RAM. It is very fast.
- NDB is the MySQL Cluster storage engine.
- MERGE is a variation of MyISAM. A MERGE table is a collection of identical MyISAM tables, which means that all tables have the same columns, column types, indexes, and so on.

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf) and the *OSGeo FDO Provider for MySQL API Reference Help* (MySQL_Provider_API.chm).

FDO Provider for MySQL Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Capability	MySQL
Spatial Content Extent Types	
Static	Y
Dynamic	
Lock Types	
Shared	
Exclusive	
Transaction	
All Long Transaction Exclusive	
Long Transaction Exclusive	
Thread Types	

Capability	MySQL
Single- Threaded	
Per Connection Threaded	Y
Per Command Threaded	
Multi-threaded	
Supports	
Locking	
Timeout	
Transactions	Y
Long Transactions	
SQL	Y
Configuration	

Schema Capabilities

Capability	MySQL
Class Types	
Class	Y
Feature Class	Y
Data Types	
Boolean	Y

Capability	MySQL
Byte	Y
DateTime	Y
Decimal	Y
Double	Y
Int16	Y
Int32	Y
Int64	Y
Single	Y
String	Y
BLOB	
CLOB	
Auto-Generated Data Types Supported (Int64)	Y
Supports	
Inheritance	Y
Multiple Schemas	Y
Object Properties	Y
Association Properties	Y

Capability	MySQL
Schema Overrrides	Y
Network Model	
Auto Id Generation	Y
Data Store Scope Unique Id Generation	Y
Schema Modification	Y
Constraints	
Inclusive Value Range Constraints	Y
Exclusive Value Range Constraints	
Value Constraints List	
Null Value Constraints	Y
Unique Value Constraints	Y
Composite Unique Value Constraints	Y

Command Capabilities

Capability	MySQL
Feature Commands	
Select	Y
Select Aggregates	Y
Insert	Y

Capability	MySQL
Delete	Y
Update	Y
Schema Commands	
Describe Schema	Y
Describe Schema Mapping	Y
Apply Schema	Y
Destroy Schema	Y
Spatial Context Commands	
Activate Spatial Context	Y
Create Spatial Context	Y
Destroy Spatial Context	Y
Get Spatial Contexts	Y
Data Store Commands	
Create Data Store	Y
Destroy Data Store	Y
List Data Stores	Y
Measure Unit Commands	
Create Measure Unit	

Capability	MySQL
Destroy Measure Unit	
Get Measure Units	
SQL Command	Y
Locking Commands	
Acquire Lock	
Get Lock Info	
Get Locked Objects	
Get Lock Owners	
Release Lock	
Long Transaction Commands	
Activate Long Transaction	
Deactivate Long Transaction	
Commit Long Transaction	
Create Long Transaction	
Get Long Transactions	
Freeze Long Transaction	
Rollback Long Transaction	
Activate Long Transaction Checkpoint	

Capability	MySQL
Create Long Transaction Checkpoint	
Get Long Transaction Checkpoints	
Rollback Long Transaction Checkpoint	
Change Long Transaction Privileges	
Get Long Transaction Privileges	
Change Long Transaction Set	
Get Long Transaction In Set	
RDBMS Custom Commands	
Create Spatial Index	Y
Destroy Spatial Index	Y
Get Spatial Indexes	Y
Supports	
Parameters	Y
Timeout	
Select Expressions	Y
Select Functions	Y
Select Distinct	Y
Select Ordering	Y

Capability	MySQL
Select Grouping	Y

Filter Capabilities

Capability	MySQL
Condition Types	
Comparsion	Y
Like	Y
In	Y
Null	Y
Spatial	Y
Distance	Y
Spatial Operations	
Contains	
Crosses	
Disjoint	
Equals	
Intersects	Y
Overlaps	
Touches	

Capability	MySQL
Within	
Covered By	Y
Inside	Y
Envelope Intersects	Y
Distance Operations	
Beyond	
Within	Y
Supports	
Geodesic Distance	
Non Literal Geometric Operations	

Expression Capabilities

Capability	MySQL
Expression Types	
Basic	Y
Function	Y
Parameter	Y
Functions	
Double Avg(Double)	Y

Capability	MySQL
Int64 Ceil(Int64)	Y
String Concat(String)	Y
Int64 Count(Int64)	Y
Int64 Floor(Int64)	Y
String Lower(String)	Y
Double Min(Double)	Y
Double Max(Double)	Y
Double StdDev(Double)	Y
Double Sum(Double)	Y
String Upper(String)	Y
geomValue SpatialExtents (TBD - FdoDataType)	

Geometry Capabilities

Capability	MySQL
Geometry Types	
Point	Y
Line String	Y
Polygon	Y

Capability	MySQL
Multi-Point	Y
Multi-Line String	Y
Multi-Polygon	Y
Multi-Geometry	
Curve String	Y
Curve Polygon	Y
Multi-Curve String	Y
Multi-Curve Polygon	Y
Geometry Component Types	
Linear Ring	Y
Line String Segment	Y
Circular Arc Segment	Y
Ring	Y
Dimensionalities	
XY	Y
Z	
M	

Raster Capabilities

Capability	MySQL
Supports	
Raster	
Stitching	
Subsampling	
Bitonal Data Model	
Gray Data Model	
RGB Data Model	
RGBA Data Model	
Palette Data Model	

OSGeo FDO Provider for ODBC

D

This appendix discusses FDO API development issues that are related to OSGeo FDO Provider for ODBC.

In this chapter

What Is FDO Provider for ODBC?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The FDO Provider for ODBC provides FDO with access to an ODBC-based data store.

The FDO Provider for ODBC can access simple x, y, z feature objects that can run in a multi-platform environment, including Windows, Linux, and UNIX.

The FDO Provider for ODBC has the following characteristics:

- The FDO Provider for ODBC supports the definition of one or more feature classes in terms of any relational database table that contains an X, Y, and optionally, Z columns.
- Metadata, which maps the table name, and X, Y, and optionally, Z columns to a feature class, is maintained outside the database in a configuration file. This information, in conjunction with the table structure in the database, provides the definition of the feature class.
- The x, y, and z locations of objects are stored in separate properties in the primary object definition of a feature, but are accessible through a single class property 'Geometry'.
- Read-only access is provided to pre-existing data defined and populated through 3rd party applications (that is, FDO Provider for ODBC will not be responsible for defining the physical schema of the data store nor for populating the object data).
- The schema configuration of the data store is provided to the FDO Provider for ODBC through an optional XML file containing the Geographic Markup Language (GML) definition of the schema that maps 'tables' and 'columns' in the data store to feature classes and property mappings in the FDO data model.

NOTE Microsoft Excel (must have at least one named range; do not use DATABASE or other reserved words as a range name).

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf) and the *OSGeo FDO Provider for ODBC API Reference Help* (ODBC_Provider_API.chm).

FDO Provider for ODBC Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Capability	ODBC
Spatial Content Extent Types	
Static	Y
Dynamic	
Lock Types	
Shared	
Exclusive	
Transaction	
All Long Transaction Exclusive	
Long Transaction Exclusive	
Thread Types	

Capability	ODBC
Single- Threaded	
Per Connection Threaded	Y
Per Command Threaded	
Multi-threaded	
Supports	
Locking	
Timeout	
Transactions	
Long Transactions	
SQL	Y
Configuration	

Schema Capabilities

Capability	ODBC
Class Types	
Class	Y
Feature Class	Y
Data Types	
Boolean	Y

Capability	ODBC
Byte	Y
DateTime	Y
Decimal	Y
Double	Y
Int16	Y
Int32	Y
Int64	Y
Single	Y
String	Y
BLOB	
CLOB	
Auto-Generated Data Types Supported (Int16, Int32, Int64)	Y
Supports	
Inheritance	Y
Multiple Schemas	Y
Object Properties	
Association Properties	

Capability	ODBC
Schema Overrrides	Y
Network Model	
Auto Id Generation	Y
Data Store Scope Unique Id Generation	
Schema Modification	
Constraints	
Inclusive Value Range Constraints	
Exclusive Value Range Constraints	
Value Constraints List	
Null Value Constraints	Y
Unique Value Constraints	
Composite Unique Value Constraints	

Command Capabilities

Capability	ODBC
Feature Commands	
Select	Y
Select Aggregates	Y
Insert	Y

Capability	ODBC
Delete	Y
Update	Y
Schema Commands	
Describe Schema	Y
Describe Schema Mapping	Y
Apply Schema	
Destroy Schema	
Spatial Context Commands	
Activate Spatial Context	
Create Spatial Context	
Destroy Spatial Context	
Get Spatial Contexts	Y
Data Store Commands	
Create Data Store	
Destroy Data Store	
List Data Stores	
Measure Unit Commands	
Create Measure Unit	

Capability	ODBC
Destroy Measure Unit	
Get Measure Units	
SQL Command	Y
Locking Commands	
Acquire Lock	Y
Get Lock Info	Y
Get Locked Objects	Y
Get Lock Owners	Y
Release Lock	Y
Long Transaction Commands	
Activate Long Transaction	Y
Deactivate Long Transaction	Y
Commit Long Transaction	Y
Create Long Transaction	Y
Get Long Transactions	Y
Freeze Long Transaction	
Rollback Long Transaction	Y
Activate Long Transaction Checkpoint	

Capability	ODBC
Create Long Transaction Checkpoint	
Get Long Transaction Checkpoints	
Rollback Long Transaction Checkpoint	
Change Long Transaction Privileges	
Get Long Transaction Privileges	
Change Long Transaction Set	
Get Long Transaction In Set	
RDBMS Custom Commands	
Create Spatial Index	
Destroy Spatial Index	
Get Spatial Indexes	
Supports	
Parameters	Y
Timeout	
Select Expressions	
Select Functions	Y
Select Distinct	Y
Select Ordering	Y

Capability	ODBC
Select Grouping	

Filter Capabilities

Capability	ODBC
Condition Types	
Comparsion	Y
Like	Y
In	Y
Null	Y
Spatial	Y
Distance	
Spatial Operations	
Contains	
Crosses	
Disjoint	
Equals	
Intersects	Y
Overlaps	
Touches	

Capability	ODBC
Within	Y
Covered By	
Inside	Y
Envelope Intersects	Y
Distance Operations	
Beyond	
Within	
Supports	
Geodesic Distance	
Non Literal Geometric Operations	

Expression Capabilities

Capability	ODBC
Expression Types	
Basic	Y
Function	Y
Parameter	
Functions	
Double Avg(Double)	Y

Capability	ODBC
Int64 Ceil(Int64)	Y
String Concat(String)	Y
Int64 Count(Int64)	Y
Int64 Floor(Int64)	Y
String Lower(String)	Y
Double Min(Double)	Y
Double Max(Double)	Y
Double StdDev(Double)	Y
Double Sum(Double)	Y
String Upper(String)	Y
geomValue SpatialExtents (TBD - Fdo-DataType)	

Geometry Capabilities

Capability	ODBC
Geometry Types	
Point	Y
Line String	
Polygon	

Capability	ODBC
Multi-Point	
Multi-Line String	
Multi-Polygon	
Multi-Geometry	
Curve String	
Curve Polygon	
Multi-Curve String	
Multi-Curve Polygon	
Geometry Component Types	
Linear Ring	
Line String Segment	
Circular Arc Segment	
Ring	
Dimensionalities	
XY	Y
Z	Y
M	

Raster Capabilities

Capability	ODBC
Supports	
Raster	
Stitching	
Subsampling	
Bitonal Data Model	
Gray Data Model	
RGB Data Model	
RGBA Data Model	
Palette Data Model	

Autodesk FDO Provider for Raster

E

This appendix discusses FDO API development issues that are related to Autodesk FDO Provider for Raster.

In this chapter

What Is FDO Provider for Raster?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The Autodesk FDO Provider for Raster is a stand-alone file format that supports GIS data.

The FDO Provider for Raster has the following characteristics:

- The FDO Provider for Raster supports georeferenced file-based raster images and file-based grid coverages. Raster images are pixel-based images, such as digital photographs (satellite images, for example). Raster images are very useful as background images underneath your vector data, for example, an aerial photograph of a city with a layer of streets overlaying it.
- The FDO Provider for Raster can run in a multi-platform environment, including Windows and Linux.

Supported Formats

The following list shows the raster image file formats that are supported, along with their acronyms and file extensions:

- JPEG (.jpg, .jpeg) - Joint Photographic Experts Group
- JPG2K (.jp2, .j2k) - Joint Photographic Experts Group
- MrSID (.sid) - Multi-Resolution Seamless Image Database
- PNG (.png) - Portable Network Graphic
- TIFF (.tif, .tiff) - Tagged Image File Format
- DEM (.dem) - USGS Format Digital Elevation Model
- ECW (.ecw) - Enhanced Compressed Wavelet
- DTED (.dt0, .dt1, dt2) - Digital Terrain Elevation Data
- ESRI ASCII GRID (.asc) - ESRI Surface
- ESRI Binary GRID (.adf) - ESRI Surface

Supported Data Models

The following are the data models supported:

ModelType	BitsPerPixel	Organization	DataType
Bitonal	1	Pixel	Unsigned Integer
Grey	8	Pixel	Unsigned Integer
RGB	24	Pixel	Unsigned Integer
RGBA	32	Pixel	Unsigned Integer
Pallete	8	Pixel	Unsigned Integer
Data	1	Pixel	Unsigned Integer
Data	8	Pixel	Unsigned Integer
Data	8	Pixel	Signed Integer
Data	16	Pixel	Unsigned Integer
Data	16	Pixel	Signed Integer
Data	32	Pixel	Unsigned Integer
Data	32	Pixel	Signed Integer
Data	32	Pixel	Float

NOTE Only DEM, TIFF, and ECW images support the 'Data' ModelType.

NOTE All 2- and 4-BitsPerPixel images are promoted to 8 BitsPerPixel as per the underlying ATIL behavior.

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf) and the *Autodesk FDO Provider for Raster API Reference Help* (Raster_Provider_API.chm).

FDO Provider for Raster Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Capability	Raster
Spatial Content Extent Types	
Static	Y
Dynamic	
Lock Types	
Shared	
Exclusive	
Transaction	
All Long Transaction Exclusive	

Capability	Raster
Long Transaction Exclusive	
Thread Types	
Single- Threaded	
Per Connection Threaded	Y
Per Command Threaded	
Multi-threaded	
Supports	
Locking	Y
Timeout	
Transactions	
Long Transactions	
SQL	
Configuration	Y

Schema Capabilities

Capability	Raster
Class Types	
Class	
Feature Class	Y

Capability	Raster
Data Types	
Boolean	
Byte	
DateTime	
Decimal	
Double	
Int16	
Int32	
Int64	
Single	
String	Y
BLOB	Y
CLOB	
Auto-Generated Data Types Supported	
Supports	
Inheritance	Y
Multiple Schemas	Y
Object Properties	

Capability	Raster
Association Properties	
Schema Overrrides	Y
Network Model	
Auto Id Generation	
Data Store Scope Unique Id Generation	
Schema Modification	
Constraints	
Inclusive Value Range Constraints	
Exclusive Value Range Constraints	
Value Constraints List	
Null Value Constraints	
Unique Value Constraints	
Composite Unique Value Constraints	

Command Capabilities

Capability	Raster
Feature Commands	
Select	Y
Select Aggregates	Y

Capability	Raster
Insert	
Delete	
Update	
Schema Commands	
Describe Schema	Y
Describe Schema Mapping	Y
Apply Schema	
Destroy Schema	
Spatial Context Commands	
Activate Spatial Context	
Create Spatial Context	
Destroy Spatial Context	
Get Spatial Contexts	Y
Data Store Commands	
Create Data Store	
Destroy Data Store	
List Data Stores	
Measure Unit Commands	

Capability	Raster
Create Measure Unit	
Destroy Measure Unit	
Get Measure Units	Y
SQL Command	
Locking Commands	
Acquire Lock	
Get Lock Info	
Get Locked Objects	
Get Lock Owners	
Release Lock	
Long Transaction Commands	
Activate Long Transaction	
Deactivate Long Transaction	
Commit Long Transaction	
Create Long Transaction	
Get Long Transactions	
Freeze Long Transaction	
Rollback Long Transaction	

Capability	Raster
Activate Long Transaction Checkpoint	
Create Long Transaction Checkpoint	
Get Long Transaction Checkpoints	
Rollback Long Transaction Checkpoint	
Change Long Transaction Privileges	
Get Long Transaction Privileges	
Change Long Transaction Set	
Get Long Transaction In Set	
RDBMS Custom Commands	
Create Spatial Index	
Destroy Spatial Index	
Get Spatial Indexes	
Supports	
Parameters	
Timeout	
Select Expressions	
Select Functions	
Select Distinct	

Capability	Raster
Select Ordering	
Select Grouping	

Filter Capabilities

Capability	Raster
Condition Types	
Comparsion	
Like	
In	Y
Null	
Spatial	Y
Distance	
Spatial Operations	
Contains	
Crosses	
Disjoint	
Equals	
Intersects	Y
Overlaps	

Capability	Raster
Touches	
Within	Y
Covered By	
Inside	Y
Envelope Intersects	Y
Distance Operations	
Beyond	
Within	
Supports	
Geodesic Distance	
Non Literal Geometric Operations	

Expression Capabilities

Capability	Raster
Expression Types	
Basic	Y
Function	Y
Parameter	
Functions	

Capability	Raster
BLOB MOSAIC(BLOB raster)	Y
BLOB CLIP(BLOB raster, Double minX, Double minY, Double maxX, Double maxY)	Y
BLOB RESAMPLE(BLOB raster, Double minX, Double minY, Double maxX, Double maxY, Int32 height, Int32 width)	Y
Double Avg(Double)	
Int64 Ceil(Int64)	
String Concat(String)	
Int64 Count(Int64)	
Int64 Floor(Int64)	
String Lower(String)	
Double Min(Double)	
Double Max(Double)	
Double StdDev(Double)	
Double Sum(Double)	Y
String Upper(String)	
geomValue SpatialExtents (TBD - Fdo-DataType)	

Geometry Capabilities

Capability	Raster
Geometry Types	
Point	
Line String	
Polygon	
Multi-Point	
Multi-Line String	
Multi-Polygon	
Multi-Geometry	
Curve String	
Curve Polygon	
Multi-Curve String	
Multi-Curve Polygon	
Geometry Component Types	
Linear Ring	
Line String Segment	
Circular Arc Segment	
Ring	

Capability	Raster
Dimensionalities	
XY	Y
Z	
M	

Raster Capabilities

Capability	Raster
Supports	
Raster	Y
Stitching	Y
Subsampling	Y
Bitonal Data Model	Y
Gray Data Model	Y
RGB Data Model	Y
RGBA Data Model	Y
Palette Data Model	Y

OSGeo FDO Provider for SDF

F

This appendix discusses FDO API development issues that are related to OSGeo FDO Provider for SDF.

In this chapter

What Is FDO Provider for SDF?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The FDO Provider for SDF is a standalone file format that supports GIS data.

The FDO Provider for SDF uses Autodesk's spatial database format, which is a file-based personal geodatabase that supports multiple features/attributes, spatial indexing, interoperability, file-locking, and high performance for large data sets.

The SDF file format has the following characteristics:

- SDF files can be read on different platforms.
- The SDF file has its own spatial indexing.
- SDF files can store geometric and non-geometric data with minimum overhead.
- Although it does not support concurrency control (locking), the SDF file format is a valid alternative to RDBMS.

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf) and the *OSGeo FDO Provider for SDF API Reference Help* (SDF_Provider_API.chm).

FDO Provider for SDF Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Capability	SDF
Spatial Content Extent Types	
Static	
Dynamic	Y
Lock Types	
Shared	
Exclusive	
Transaction	
All Long Transaction Exclusive	
Long Transaction Exclusive	
Thread Types	
Single- Threaded	
Per Connection Threaded	Y
Per Command Threaded	
Multi-threaded	
Supports	
Locking	
Timeout	

Capability	SDF
Transactions	
Long Transactions	
SQL	
Configuration	

Schema Capabilities

Capability	SDF
Class Types	
Class	Y
Feature Class	Y
Data Types	
Boolean	Y
Byte	Y
DateTime	Y
Decimal	Y
Double	Y
Int16	Y
Int32	Y
Int64	Y

Capability	SDF
Single	Y
String	Y
BLOB	
CLOB	
Auto-Generated Data Types Supported (Int32)	Y
Supports	
Inheritance	
Multiple Schemas	
Object Properties	
Association Properties	
Schema Overrrides	
Network Model	
Auto Id Generation	Y
Data Store Scope Unique Id Generation	
Schema Modification	
Constraints	
Inclusive Value Range Constraints	

Capability	SDF
Exclusive Value Range Constraints	Y
Value Constraints List	
Null Value Constraints	Y
Unique Value Constraints	
Composite Unique Value Constraints	

Command Capabilities

Capability	SDF
Feature Commands	
Select	Y
Select Aggregates	Y
Insert	Y
Delete	Y
Update	Y
Schema Commands	
Describe Schema	Y
Describe Schema Mapping	
Apply Schema	Y
Destroy Schema	

Capability	SDF
Spatial Context Commands	
Activate Spatial Context	
Create Spatial Context	Y
Destroy Spatial Context	
Get Spatial Contexts	Y
Data Store Commands	
Create Data Store	Y
Destroy Data Store	Y
List Data Stores	
Measure Unit Commands	
Create Measure Unit	
Destroy Measure Unit	
Get Measure Units	
SQL Command	Y
Locking Commands	
Acquire Lock	Y
Get Lock Info	Y
Get Locked Objects	Y

Capability	SDF
Get Lock Owners	Y
Release Lock	Y
Long Transaction Commands	
Activate Long Transaction	Y
Deactivate Long Transaction	Y
Commit Long Transaction	Y
Create Long Transaction	Y
Get Long Transactions	Y
Freeze Long Transaction	
Rollback Long Transaction	Y
Activate Long Transaction Checkpoint	
Create Long Transaction Checkpoint	
Get Long Transaction Checkpoints	
Rollback Long Transaction Checkpoint	
Change Long Transaction Privileges	
Get Long Transaction Privileges	
Change Long Transaction Set	
Get Long Transaction In Set	

Capability	SDF
RDBMS Custom Commands	
Create Spatial Index	
Destroy Spatial Index	
Get Spatial Indexes	
Supports	
Parameters	Y
Timeout	
Select Expressions	
Select Functions	Y
Select Distinct	Y
Select Ordering	Y
Select Grouping	

Filter Capabilities

Capability	SDF
Condition Types	
Comparsion	Y
Like	Y
In	Y

Capability	SDF
Null	Y
Spatial	Y
Distance	
Spatial Operations	
Contains	Y
Crosses	
Disjoint	Y
Equals	
Intersects	Y
Overlaps	
Touches	
Within	Y
Covered By	
Inside	
Envelope Intersects	Y
Distance Operations	
Beyond	
Within	

Capability	SDF
Supports	
Geodesic Distance	
Non Literal Geometric Operations	

Expression Capabilities

Capability	SDF
Expression Types	
Basic	Y
Function	Y
Parameter	
Functions	
Double Avg(Double)	Y
Int64 Ceil(Int64)	
String Concat(String)	Y
Int64 Count(Int64)	Y
Int64 Floor(Int64)	
String Lower(String)	
Double Min(Double)	Y
Double Max(Double)	Y

Capability	SDF
Double StdDev(Double)	Y
Double Sum(Double)	Y
String Upper(String)	
geomValue SpatialExtents (TBD - FdoDataType)	Y

Geometry Capabilities

Capability	SDF
Geometry Types	
Point	Y
Line String	Y
Polygon	Y
Multi-Point	Y
Multi-Line String	Y
Multi-Polygon	Y
Multi-Geometry	Y
Curve String	Y
Curve Polygon	Y
Multi-Curve String	Y

Capability	SDF
Multi-Curve Polygon	Y
Geometry Component Types	
Linear Ring	Y
Line String Segment	Y
Circular Arc Segment	Y
Ring	Y
Dimensionalities	
XY	Y
Z	Y
M	Y

Raster Capabilities

Capability	SDF
Supports	
Raster	
Stitching	
Subsampling	
Bitonal Data Model	
Gray Data Model	

Capability	SDF
RGB Data Model	
RGBA Data Model	
Palette Data Model	

G

OSGeo FDO Provider for SHP

This appendix discusses FDO API development issues that are related to OSGeo FDO Provider for SHP.

In this chapter

What Is FDO Provider for SHP?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The FDO Provider for SHP provides FDO with access to an SHP-based data store.

The FDO Provider for SHP uses a standalone file format that supports GIS data. The FDO Provider for SHP (Shape) has the following characteristics:

- Read-only access is provided to pre-existing spatial and attribute data from an Environmental Systems Research Institute (ESRI) Shape file (SHP).
- The FDO Provider for SHP can run in a multi-platform environment, including Windows and Linux.
- A Shape file consists of three separate files: SHP (shape geometry), SHX (shape index), and DBF (shape attributes in dBASE format).
- The FDO Provider for SHP accesses the information in each of the three separate files, and treats each SHP, and its associated DBF file, as a feature class with a single geometry property, and optionally, with data attribute properties.
- Schema configuration of the data store is provided to the FDO Provider for SHP through an XML file containing the Geographic Markup Language (GML) definition of the schema that maps SHP and DBF data in the data store to feature classes and property mappings in the FDO data model.
- Although it does not support concurrency control (locking), the SHP file format is a valid alternative to RDBMS.

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf) and the *OSGeo FDO Provider for SHP API Reference Help* (SHP_Provider_API.chm).

FDO Provider for SHP Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands

- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Capability	SHP
Spatial Content Extent Types	
Static	Y
Dynamic	
Lock Types	
Shared	
Exclusive	
Transaction	
All Long Transaction Exclusive	
Long Transaction Exclusive	
Thread Types	
Single- Threaded	
Per Connection Threaded	Y
Per Command Threaded	
Multi-threaded	

Capability	SHP
Supports	
Locking	
Timeout	
Transactions	
Long Transactions	
SQL	
Configuration	Y

Schema Capabilities

Capability	SHP
Class Types	
Class	Y
Feature Class	Y
Data Types	
Boolean	
Byte	
DateTime	Y
Decimal	Y
Double	

Capability	SHP
Int16	
Int32	Y
Int64	
Single	
String	Y
BLOB	
CLOB	
Auto-Generated Data Types Supported (Int32)	Y
Supports	
Inheritance	
Multiple Schemas	Y
Object Properties	
Association Properties	
Schema Overrrides	Y
Network Model	
Auto Id Generation	Y
Data Store Scope Unique Id Generation	

Capability	SHP
Schema Modification	Y
Constraints	
Inclusive Value Range Constraints	
Exclusive Value Range Constraints	
Value Constraints List	
Null Value Constraints	Y
Unique Value Constraints	
Composite Unique Value Constraints	

Command Capabilities

Capability	SHP
Feature Commands	
Select	Y
Select Aggregates	Y
Insert	Y
Delete	Y
Update	Y
Schema Commands	
Describe Schema	Y

Capability	SHP
Describe Schema Mapping	Y
Apply Schema	Y
Destroy Schema	Y
Spatial Context Commands	
Activate Spatial Context	
Create Spatial Context	
Destroy Spatial Context	
Get Spatial Contexts	Y
Data Store Commands	
Create Data Store	
Destroy Data Store	
List Data Stores	
Measure Unit Commands	
Create Measure Unit	
Destroy Measure Unit	
Get Measure Units	
SQL Command	
Locking Commands	

Capability	SHP
Acquire Lock	
Get Lock Info	
Get Locked Objects	
Get Lock Owners	
Release Lock	
Long Transaction Commands	
Activate Long Transaction	
Deactivate Long Transaction	
Commit Long Transaction	
Create Long Transaction	
Get Long Transactions	
Freeze Long Transaction	
Rollback Long Transaction	
Activate Long Transaction Checkpoint	
Create Long Transaction Checkpoint	
Get Long Transaction Checkpoints	
Rollback Long Transaction Checkpoint	
Change Long Transaction Privileges	

Capability	SHP
Get Long Transaction Privileges	
Change Long Transaction Set	
Get Long Transaction In Set	
RDBMS Custom Commands	
Create Spatial Index	
Destroy Spatial Index	
Get Spatial Indexes	
Supports	
Parameters	Y
Timeout	
Select Expressions	Y
Select Functions	Y
Select Distinct	Y
Select Ordering	Y
Select Grouping	

Filter Capabilities

Capability	SHP
Condition Types	

Capability	SHP
Comparsion	Y
Like	Y
In	Y
Null	Y
Spatial	Y
Distance	
Spatial Operations	
Contains	
Crosses	
Disjoint	
Equals	
Intersects	
Overlaps	
Touches	
Within	Y
Covered By	
Inside	Y
Envelope Intersects	Y

Capability	SHP
Distance Operations	
Beyond	
Within	
Supports	
Geodesic Distance	
Non Literal Geometric Operations	

Expression Capabilities

Capability	SHP
Expression Types	
Basic	Y
Function	Y
Parameter	
Functions	
Double Avg(Double)	Y
Int64 Ceil(Int64)	Y
String Concat(String)	Y
Int64 Count(Int64)	Y
Int64 Floor(Int64)	Y

Capability	SHP
String Lower(String)	Y
Double Min(Double)	Y
Double Max(Double)	Y
Double StdDev(Double)	Y
Double Sum(Double)	Y
String Upper(String)	Y
geomValue SpatialExtents (TBD - Fdo-DataType)	

Geometry Capabilities

Capability	SHP
Geometry Types	
Point	Y
Line String	Y
Polygon	Y
Multi-Point	Y
Multi-Line String	Y
Multi-Polygon	Y
Multi-Geometry	

Capability	SHP
Curve String	
Curve Polygon	
Multi-Curve String	
Multi-Curve Polygon	
Geometry Component Types	
Linear Ring	Y
Line String Segment	Y
Circular Arc Segment	
Ring	
Dimensionalities	
XY	Y
Z	Y
M	Y

Raster Capabilities

Capability	SHP
Supports	
Raster	
Stitching	

Capability	SHP
Subsampling	
Bitonal Data Model	
Gray Data Model	
RGB Data Model	
RGBA Data Model	
Palette Data Model	

Autodesk FDO Provider for SQL Server

H

This appendix discusses FDO API development issues that are related to Autodesk FDO Provider for SQL Server.

In this chapter

What Is FDO Provider for SQL Server?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The FDO Provider for SQL Server provides FDO with access to a Microsoft SQL Server-based data store.

The Autodesk FDO Provider for SQL Server API provides custom commands that are specifically designed to work with the FDO API. For example, using these commands, you can do the following:

- Read and create schema.
- Read and write geospatial and non-geospatial data.

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf) and the *Autodesk FDO Provider for SQL Server API Reference Help* (SQLServer_Provider_API.chm).

FDO Provider for SQL Server Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Capability	SQL Server
Spatial Content Extent Types	

Capability	SQL Server
Static	Y
Dynamic	
Lock Types	
Shared	
Exclusive	
Transaction	
All Long Transaction Exclusive	
Long Transaction Exclusive	
Thread Types	
Single- Threaded	
Per Connection Threaded	Y
Per Command Threaded	
Multi-threaded	
Supports	
Locking	
Timeout	
Transactions	Y
Long Transactions	

Capability	SQL Server
SQL	Y
Configuration	

Schema Capabilities

Capability	SQL Server
Class Types	
Class	Y
Feature Class	Y
Data Types	
Boolean	Y
Byte	Y
DateTime	Y
Decimal	Y
Double	Y
Int16	Y
Int32	Y
Int64	Y
Single	Y
String	Y

Capability	SQL Server
BLOB	
CLOB	
Auto-Generated Data Types Supported (Int64)	Y
Supports	
Inheritance	Y
Multiple Schemas	Y
Object Properties	Y
Association Properties	Y
Schema Overrrides	Y
Network Model	
Auto Id Generation	Y
Data Store Scope Unique Id Generation	Y
Schema Modification	Y
Constraints	
Inclusive Value Range Constraints	Y
Exclusive Value Range Constraints	Y
Value Constraints List	Y

Capability	SQL Server
Null Value Constraints	Y
Unique Value Constraints	Y
Composite Unique Value Constraints	Y

Command Capabilities

Capability	SQL Server
Feature Commands	
Select	Y
Select Aggregates	Y
Insert	Y
Delete	Y
Update	Y
Schema Commands	
Describe Schema	Y
Describe Schema Mapping	Y
Apply Schema	Y
Destroy Schema	Y
Spatial Context Commands	
Activate Spatial Context	Y

Capability	SQL Server
Create Spatial Context	Y
Destroy Spatial Context	Y
Get Spatial Contexts	Y
Data Store Commands	
Create Data Store	Y
Destroy Data Store	Y
List Data Stores	Y
Measure Unit Commands	
Create Measure Unit	
Destroy Measure Unit	
Get Measure Units	
SQL Command	Y
Locking Commands	
Acquire Lock	Y
Get Lock Info	Y
Get Locked Objects	Y
Get Lock Owners	Y
Release Lock	Y

Capability	SQL Server
Long Transaction Commands	
Activate Long Transaction	Y
Deactivate Long Transaction	Y
Commit Long Transaction	Y
Create Long Transaction	Y
Get Long Transactions	Y
Freeze Long Transaction	
Rollback Long Transaction	Y
Activate Long Transaction Checkpoint	
Create Long Transaction Checkpoint	
Get Long Transaction Checkpoints	
Rollback Long Transaction Checkpoint	
Change Long Transaction Privileges	
Get Long Transaction Privileges	
Change Long Transaction Set	
Get Long Transaction In Set	
RDBMS Custom Commands	
Create Spatial Index	Y

Capability	SQL Server
Destroy Spatial Index	Y
Get Spatial Indexes	Y
Supports	
Parameters	
Timeout	
Select Expressions	Y
Select Functions	Y
Select Distinct	Y
Select Ordering	Y
Select Grouping	Y

Filter Capabilities

Capability	SQL Server
Condition Types	
Comparsion	Y
Like	Y
In	Y
Null	Y
Spatial	Y

Capability	SQL Server
Distance	Y
Spatial Operations	
Contains	
Crosses	
Disjoint	
Equals	
Intersects	Y
Overlaps	
Touches	
Within	Y
Covered By	Y
Inside	Y
Envelope Intersects	Y
Distance Operations	
Beyond	
Within	
Supports	
Geodesic Distance	

Capability	SQL Server
Non Literal Geometric Operations	

Expression Capabilities

Capability	SQL Server
Expression Types	
Basic	Y
Function	Y
Parameter	
Functions	
Double Avg(Double)	Y
Int64 Ceil(Int64)	Y
String Concat(String)	Y
Int64 Count(Int64)	Y
Int64 Floor(Int64)	Y
String Lower(String)	Y
Double Min(Double)	Y
Double Max(Double)	Y
Double StdDev(Double)	Y
Double Sum(Double)	Y

Capability	SQL Server
String Upper(String)	Y
geomValue SpatialExtents (TBD - FdoDataType)	

Geometry Capabilities

Capability	SQL Server
Geometry Types	
Point	Y
Line String	Y
Polygon	Y
Multi-Point	Y
Multi-Line String	Y
Multi-Polygon	Y
Multi-Geometry	
Curve String	Y
Curve Polygon	Y
Multi-Curve String	Y
Multi-Curve Polygon	Y
Geometry Component Types	

Capability	SQL Server
Linear Ring	Y
Line String Segment	Y
Circular Arc Segment	Y
Ring	Y
Dimensionalities	
XY	Y
Z	Y
M	Y

Raster Capabilities

Capability	SQL Server
Supports	
Raster	
Stitching	
Subsampling	
Bitonal Data Model	
Gray Data Model	
RGB Data Model	
RGBA Data Model	

Capability	SQL Server
Palette Data Model	

OSGeo FDO Provider for WFS

This appendix discusses FDO API development issues that are related to OSGeo FDO Provider for WFS.

In this chapter

What Is FDO Provider for WFS?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The FDO Provider for WFS provides FDO with access to a WFS-based data store.

An OGC Web Feature Service (WFS) provides access to geographic features that are stored in an opaque data store in a client/server environment. A client uses WFS to retrieve geospatial data that is encoded in Geography Markup Language (GML) from a single or multiple Web Feature Service. The communication between client and server is encoded in XML. If the WFS response includes feature geometries, it is encoded in Geography Markup Language (GML), which is specified in the OpenGIS Geographic Markup Language Implementation Specification.

Using FDO Provider for WFS data manipulation operations, you can do the following:

- Query features based on spatial and non-spatial constraints.
- Create new feature instances.
- Delete feature instances.
- Update feature instances.
- Lock feature instances.

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf).

NOTE There is no public API documentation for the FDO Provider for WFS; functionality is available through the main FDO API.

FDO Provider for WFS Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Expressions

- Filters
- Geometry
- Raster

Connection Capabilities

Capability	WFS
Spatial Content Extent Types	
Static	Y
Dynamic	
Lock Types	
Shared	
Exclusive	
Transaction	
All Long Transaction Exclusive	
Long Transaction Exclusive	
Thread Types	
Single- Threaded	
Per Connection Threaded	Y
Per Command Threaded	
Multi-threaded	

Capability	WFS
Supports	
Locking	
Timeout	
Transactions	
Long Transactions	
SQL	
Configuration	

Schema Capabilities

Capability	WFS
Class Types	
Class	Y
Feature Class	Y
Data Types	
Boolean	Y
Byte	Y
DateTime	Y
Decimal	Y
Double	Y

Capability	WFS
Int16	Y
Int32	Y
Int64	Y
Single	Y
String	Y
BLOB	
CLOB	
Auto-Generated Data Types Supported	
Supports	
Inheritance	Y
Multiple Schemas	Y
Object Properties	Y
Association Properties	Y
Schema Overrrides	
Network Model	
Auto Id Generation	
Data Store Scope Unique Id Generation	
Schema Modification	

Capability	WFS
Constraints	
Inclusive Value Range Constraints	
Exclusive Value Range Constraints	
Value Constraints List	
Null Value Constraints	
Unique Value Constraints	
Composite Unique Value Constraints	

Command Capabilities

Capability	WFS
Feature Commands	
Select	Y
Select Aggregates	
Insert	
Delete	
Update	
Schema Commands	
Describe Schema	Y
Describe Schema Mapping	

Capability	WFS
Apply Schema	
Destroy Schema	
Spatial Context Commands	
Activate Spatial Context	
Create Spatial Context	
Destroy Spatial Context	
Get Spatial Contexts	Y
Data Store Commands	
Create Data Store	
Destroy Data Store	
List Data Stores	
Measure Unit Commands	
Create Measure Unit	
Destroy Measure Unit	
Get Measure Units	
SQL Command	
Locking Commands	
Acquire Lock	

Capability	WFS
Get Lock Info	
Get Locked Objects	
Get Lock Owners	
Release Lock	
Long Transaction Commands	
Activate Long Transaction	
Deactivate Long Transaction	
Commit Long Transaction	
Create Long Transaction	
Get Long Transactions	
Freeze Long Transaction	
Rollback Long Transaction	
Activate Long Transaction Checkpoint	
Create Long Transaction Checkpoint	
Get Long Transaction Checkpoints	
Rollback Long Transaction Checkpoint	
Change Long Transaction Privileges	
Get Long Transaction Privileges	

Capability	WFS
Change Long Transaction Set	
Get Long Transaction In Set	
RDBMS Custom Commands	
Create Spatial Index	
Destroy Spatial Index	
Get Spatial Indexes	
Supports	
Parameters	
Timeout	
Select Expressions	
Select Functions	
Select Distinct	
Select Ordering	
Select Grouping	

Filter Capabilities

Capability	WFS
Condition Types	
Comparsion	

Capability	WFS
Like	
In	
Null	
Spatial	
Distance	
Spatial Operations	
Contains	
Crosses	
Disjoint	
Equals	
Intersects	
Overlaps	
Touches	
Within	
Covered By	
Inside	
Envelope Intersects	
Distance Operations	

Capability	WFS
Beyond	
Within	
Supports	
Geodesic Distance	
Non Literal Geometric Operations	

Expression Capabilities

Capability	WFS
Expression Types	
Basic	Y
Function	
Parameter	
Functions	
Double Avg(Double)	
Int64 Ceil(Int64)	
String Concat(String)	
Int64 Count(Int64)	
Int64 Floor(Int64)	
String Lower(String)	

Capability	WFS
Double Min(Double)	
Double Max(Double)	
Double StdDev(Double)	
Double Sum(Double)	
String Upper(String)	
geomValue SpatialExtents (TBD - Fdo-DataType)	

Geometry Capabilities

Capability	WFS
Geometry Types	
Point	Y
Line String	Y
Polygon	Y
Multi-Point	Y
Multi-Line String	Y
Multi-Polygon	Y
Multi-Geometry	Y
Curve String	Y

Capability	WFS
Curve Polygon	Y
Multi-Curve String	Y
Multi-Curve Polygon	Y
Geometry Component Types	
Linear Ring	Y
Line String Segment	Y
Circular Arc Segment	Y
Ring	Y
Dimensionalities	
XY	Y
Z	Y
M	Y

Raster Capabilities

Capability	WFS
Supports	
Raster	
Stitching	
Subsampling	

Capability	WFS
Bitonal Data Model	
Gray Data Model	
RGB Data Model	
RGBA Data Model	
Palette Data Model	

OSGeo FDO Provider for WMS

J

This appendix discusses FDO API development issues that are related to OSGeo FDO Provider for WMS.

In this chapter

What Is FDO Provider for WMS?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The FDO Provider for WMS provides FDO with access to a WMS-based data store.

An Open Geospatial Consortium (OGC) Web Map Service (WMS) produces maps of spatially referenced data dynamically from geographic information. This international standard defines a "map" to be a portrayal of geographic information as a digital image file suitable for display on a computer screen. A map is not the data itself. Maps by WMS are generally rendered in a pictorial format, such as PNG, GIF or JPEG, or occasionally as vector-based graphical elements in Scalable Vector Graphics (SVG) or Web Computer Graphics Metafile (WebCGM) formats.

The FDO Provider for WMS has the following characteristics:

- The FDO Provider for WMS serves up map information originating from an OGC Basic Web Map Service that provides pictorially formatted images, such as PNG, GIF, or JPEG.
- WMS map data is exposed through an FDO feature schema whose classes contain an FDO Raster property definition. The FDO schema exposed from the FDO Provider for WMS conforms to a pre-defined FDO schema that is specific to WMS and that acts as the basis for all FDO interaction with WMS data, regardless of the originating source of the WMS images.
- WMS data manipulation operations are limited to querying features based on spatial and non-spatial constraints. Schema manipulation operations are not supported.

The FDO Provider for WMS can run in a multi-platform environment, including Windows and Linux.

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf) and the *OSGeo FDO Provider for WMS API Reference Help* (WMS_Provider_API.chm).

FDO Provider for WMS Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection

- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Capability	WMS
Spatial Content Extent Types	
Static	Y
Dynamic	
Lock Types	
Shared	
Exclusive	Y
Transaction	
All Long Transaction Exclusive	
Long Transaction Exclusive	
Thread Types	
Single- Threaded	
Per Connection Threaded	Y

Capability	WMS
Per Command Threaded	
Multi-threaded	
Supports	
Locking	
Timeout	
Transactions	
Long Transactions	
SQL	
Configuration	Y

Schema Capabilities

Capability	WMS
Class Types	
Class	
Feature Class	Y
Data Types	
Boolean	
Byte	
DateTime	

Capability	WMS
Decimal	
Double	
Int16	
Int32	
Int64	
Single	
String	Y
BLOB	Y
CLOB	
Auto-Generated Data Types Supported	
Supports	
Inheritance	Y
Multiple Schemas	
Object Properties	
Association Properties	
Schema Overrrides	Y
Network Model	
Auto Id Generation	

Capability	WMS
Data Store Scope Unique Id Generation	
Schema Modification	
Constraints	
Inclusive Value Range Constraints	
Exclusive Value Range Constraints	
Value Constraints List	
Null Value Constraints	
Unique Value Constraints	
Composite Unique Value Constraints	

Command Capabilities

Capability	WMS
Feature Commands	
Select	Y
Select Aggregates	
Insert	
Delete	
Update	
Schema Commands	

Capability	WMS
Describe Schema	Y
Describe Schema Mapping	Y
Apply Schema	
Destroy Schema	
Spatial Context Commands	
Activate Spatial Context	
Create Spatial Context	
Destroy Spatial Context	
Get Spatial Contexts	Y
Data Store Commands	
Create Data Store	
Destroy Data Store	
List Data Stores	
Measure Unit Commands	
Create Measure Unit	
Destroy Measure Unit	
Get Measure Units	
SQL Command	

Capability	WMS
Locking Commands	
Acquire Lock	
Get Lock Info	
Get Locked Objects	
Get Lock Owners	
Release Lock	
Long Transaction Commands	
Activate Long Transaction	
Deactivate Long Transaction	
Commit Long Transaction	
Create Long Transaction	
Get Long Transactions	
Freeze Long Transaction	
Rollback Long Transaction	
Activate Long Transaction Checkpoint	
Create Long Transaction Checkpoint	
Get Long Transaction Checkpoints	
Rollback Long Transaction Checkpoint	

Capability	WMS
Change Long Transaction Privileges	
Get Long Transaction Privileges	
Change Long Transaction Set	
Get Long Transaction In Set	
RDBMS Custom Commands	
Create Spatial Index	
Destroy Spatial Index	
Get Spatial Indexes	
Supports	
Parameters	
Timeout	
Select Expressions	
Select Functions	
Select Distinct	
Select Ordering	
Select Grouping	

Filter Capabilities

Capability	WMS
Condition Types	
Comparsion	
Like	
In	
Null	
Spatial	
Distance	
Spatial Operations	
Contains	
Crosses	
Disjoint	
Equals	
Intersects	
Overlaps	
Touches	
Within	
Covered By	

Capability	WMS
Inside	
Envelope Intersects	
Distance Operations	
Beyond	
Within	
Supports	
Geodesic Distance	
Non Literal Geometric Operations	

Expression Capabilities

Capability	WMS
Expression Types	
Basic	
Function	Y
Parameter	
Functions	
BLOB RESAMPLE(BLOB raster, Double minX, Double minY, Double maxX, Double maxY, Int32 height, Int32 width)	Y

Capability	WMS
BLOB CLIP(BLOB raster, Double minX, Double minY, Double maxX, Double maxY)	Y
Double Avg(Double)	
Int64 Ceil(Int64)	
String Concat(String)	
Int64 Count(Int64)	
Int64 Floor(Int64)	
String Lower(String)	
Double Min(Double)	
Double Max(Double)	
Double StdDev(Double)	
Double Sum(Double)	
String Upper(String)	
geomValue SpatialExtents (TBD - Fdo-DataType)	

Geometry Capabilities

Capability	WMS
Geometry Types	
Point	

Capability	WMS
Line String	
Polygon	
Multi-Point	
Multi-Line String	
Multi-Polygon	
Multi-Geometry	
Curve String	
Curve Polygon	
Multi-Curve String	
Multi-Curve Polygon	
Geometry Component Types	
Linear Ring	
Line String Segment	
Circular Arc Segment	
Ring	
Dimensionalities	
XY	Y
Z	

Capability	WMS
M	

Raster Capabilities

Capability	WMS
Supports	
Raster	Y
Stitching	
Subsampling	Y
Bitonal Data Model	Y
Gray Data Model	Y
RGB Data Model	Y
RGBA Data Model	Y
Palette Data Model	Y

Index

E

F

G

H

I

K

L

M